Frédéric Benhamou (Ed.)

Principles and Practice of Constraint Programming - CP 2006

12th International Conference, CP 2006
Nantes, France, September 25-29, 2006
Proceedings

Springer

Volume Editor

Frédéric Benhamou
Laboratoire d'Informatique de Nantes-Atlantique
2, rue de la Houssinière, 44322 Nantes, Cedex 03, France
E-mail: Frederic.Benhamou@univ-nantes.fr

Library of Congress Control Number: 2006933296

CR Subject Classification (1998): D.1, D.3.2-3, I.2.3-4, F.3.2, I.2.8, F.4.1, J.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-46267-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-46267-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11889205 06/3142 5 4 3 2 1 0

Preface

The 12th International Conference on the Principles and Practice of Constraint Programming (CP 2006) was held in Nantes, France, September 24–29, 2006. Information about the conference can be found on the Web at http://www.sciences.univ-nantes.fr/cp06/. Information about past conferences in the series can be found at http://www.cs.ualberta.ca/~ai/cp/.

The CP conference series is the premier international conference on constraint programming and is held annually. The conference is concerned with all aspects of computing with constraints, including: algorithms, applications, environments, languages, models and systems. This series of conferences was created to bring together researchers from different disciplines, interested in high-level modeling as well as sound and efficient resolution of complex optimization and satisfaction problems. Based on this interdisciplinary culture, the CP series of conferences is widely open to different communities of researchers interested in constraint satisfaction, SAT, mathematical programming, problem modeling, system design and implementation, etc.

This year, we received 142 submissions. All of the submitted papers received at least three reviews and were discussed in much detail during an online Program Committee meeting. As a result, the Program Committee chose to publish 42 full papers and 21 short papers in the proceedings (the selection rate for this year is 0.30 for full papers and 0.46 for all accepted contributions). Following the standard format for CP, the full papers were presented at the conference in two parallel tracks and the short papers were presented as posters during a dedicated session. This year, one paper was selected by a subcommittee of the Program Committee to receive a best paper award. The subcommittee was composed of Francesca Rossi, Mark Wallace and myself. To illustrate the variety of the topics addressed in the conference, as well as the industrial impact of constraint programming, the conference program also included four invited talks by Shabbir Ahmed, Giuseppe F. Italiano, Jean-Pierre Merlet and Helmut Simonis. An additional invited talk was given by the recipient of the second "Award for Research Excellence in Constraint Programming" given by the Association for Constraint Programming during the conference. Finally, the core of the technical program included three excellent tutorials: "Soft Constraint Solving" by Javier Larrosa and Thomas Schiex, "Constraint Satisfaction for Stimuli Generation for Hardware Verification" by Yehuda Naveh and "Constraint-Based Local Search in Comet" by Laurent Michel and Pascal Van Hentenryck.

Many other scientific contributions made the program of CP 2006 a rich mix of tradition and innovation. CP 2006 continued the tradition of the CP doctoral program, in which PhD students presented their work, listened to tutorials on career and ethical issues, and discussed their work with senior researchers via a mentoring scheme. As usual, the Doctoral Program was a big success with the participation of 38 young researchers. Also traditional and very important to demonstrate the practical impact of the field was the systems demonstration session. Finally, the CP series are now well known for their first day of satellite workshops, mixing well-established meetings and new scientific events dedicated to the most recent evolutions of CP. We had ten workshops this year, each with their own proceedings, and the second solver competition was also organized during this workshop day.

CP 2006 also hosted for the first time two important events, addressing crucial aspects of our field: practical impact and future orientations of CP. The first event, dedicated to CP systems, was CPTools'06, the first international day on constraint programming tools, organized by Laurent Michel, Christian Schulte and Pascal Van Hentenryck. One of the main reasons of the success of CP is the ability of CP systems to address both expressiveness and efficiency issues. This successful meeting was the perfect place for researchers and practitioners to exchange on all aspects of system design and usability. The second forum, organized by Lucas Bordeaux, Barry O'Sullivan and Pascal Van Hentenryck, was a half-day plenary workshop called "The Next Ten Years of Constraint Programming." This event was an important occasion to share ideas on the future of our discipline during exciting and lively discussion and debates.

On behalf of the CP community, I would like to take this opportunity to warmly thank the many people who participated by their hard work and constant commitment to the organization of CP 2006.

My first and warmest thanks are for Narendra Jussien, General Co-chair of CP 2006, who did an amazing job on the organization front. It was a real pleasure to work with him on this project. The different organization chairs were particularly efficient, professional and friendly. Barry O'Sullivan, Workshop and Tutorial Chair, helped us all along the process and was instrumental in many aspects of the organization. I want to thank him here warmly for his efficiency and his patience. Thank you very much to Zeynep Kiziltan and Brahim Hnich, the Doctoral Program Chairs, for their efficiency and reactivity on this difficult task. Laurent Michel was Chair for poster and demo presentations and took care of these important aspects of the conference in a very smooth way. Many thanks to him. Christian Schulte and Mikael Lagerkvist, Publicity Chairs, had the crucial mission of advertising the conference. I thank them here very much for their friendly and competent participation. It was a great pleasure to work with the CP 2006 Program Committee on the scientific program. The commitment and reactivity of its members as well as the amazing amount of work they invested in reviewing and discussing the submitted papers was truly impressive and I would like to thank them all for their hard work. Thank you to Francesca Rossi and Mark Wallace for their help on the Best Paper Award Committee. Locally, nothing could have been done without the commitment and hard work of the Local Organizing Committee members. Let me warmly thank here Charlotte, Christophe, Christophe, Frédéric, Guillaume, Hadrien, Jean-Marie, Marco, Philippe, Romuald, Sophie, Thierry, Thomas and Xavier.

The final thanks go to the institutions that helped sponsor the conference and particularly to support the doctoral program and to bring in invited speakers. These institutions and companies are: the Région des pays de la Loire, Nantes Métropole, ILOG, the École des Mines de Nantes, the Association for Constraint Programming, the Cork Constraint Computation Center, the Intelligent Information Systems Institute, the Laboratoire d'Informatique de Nantes Atlantique, the Université de Nantes, the Conseil Général de Loire Atlantique and the Association Française pour la Programmation par Contraintes.

September 2006 Frédéric Benhamou

Organization

Conference Organization

Conference Chairs	Frédéric Benhamou, Univ. of Nantes, France
	Narendra Jussien, EMN, France
Program Chair	Frédéric Benhamou, Univ. of Nantes, France
Workshop and Tutorial Chair	Barry O'Sullivan, Univ. College Cork, Ireland
Poster and Demo Chair	Laurent Michel, Univ. of Connecticut, USA
Publicity Chairs	Christian Schulte, KTH, Sweden
	Mikael Lagerkvist, KTH, Sweden
Doctoral Program Chairs	Zeynep Kiziltan, Univ. of Bologna, Italy
	Brahim Hnich, Univ. College Cork, Ireland

Program Committee

Pedro Barahona, New U. of Lisbon, Portugal
Nicolas Beldiceanu, EMN, France
Christian Bessiere, LIRMM-CNRS, France
David Cohen, Royal Holloway, Great Britain
Andrew Davenport, IBM, USA
Boi Faltings, EPFL, Switzerland
Carla Gomes, Cornell, USA
Laurent Granvilliers, U. of Nantes, France
John Hooker, CMU, USA
Peter Jonsson, Linköping U., Sweden
Irit Katriel, U. of Aarhus, Denmark
Zeynep Kiziltan, U. of Bologna, Italy
Luc Jaulin, ENSIETA Brest, France
Jimmy Lee, CUHK, Hong Kong
Michael Maher, NICTA, Australia
Pedro Meseguer, IIIA-CSIC, Spain
Laurent Michel, U. of Connecticut, USA
Michela Milano, U. of Bologna, Italy
Barry O'Sullivan, 4C, Ireland
Gilles Pesant, Polytech Montréal, Canada
Jean-François Puget, ILOG, France
Jean-Charles Régin, ILOG, France
Francesca Rossi, U. of Padova, Italy
Louis-Martin Rousseau, Polytech Montréal, Canada
Michel Rueher, U. of Nice, France
Thomas Schiex, INRA Toulouse, France
Christian Schulte, KTH, Sweden
Helmut Simonis, CrossCore Optimization, Great Britain
Barbara Smith, 4C, Ireland
Peter Stuckey, U. of Melbourne, Australia
Peter van Beek, U. of Waterloo, Canada
Pascal Van Hentenryck, Brown U. USA
Mark Wallace, Monash U., Australia
Toby Walsh, NICTA and UNSW, Australia
Roland Yap, NUS, Singapore
Makoto Yokoo, Kyushu U., Japan
Weixiong Zhang, Washington U., USA

Referees

Anbulagan
Carlos Ansotegui
Christian Artigues
Francisco Azevedo
Pedro Barahona
Vincent Barichard
Roman Bartak
Joe Bater
Nicolas Beldiceanu
Belaïd Benhamou
Christian Bessiere
Ateet Bhalla
Stefano Bistarelli
Simon Boivin
Lucas Bordeaux
Eric Bourreau
Sebastian Brand
Marc Brisson
Ismel Brito
Ken Brown
Hadrien Cambazard
Tom Carchrae
Mats Carlsson
Amedeo Cesta
Gilles Chabert
Nicolas Chapados
Kenil Cheng
Dave Cohen
Hélène Collavizza
Martin Cooper
Marie-Claude Coté
Jorge Cruz
Andrew Davenport
Romuald Debruyne
Sophie Demassey
Iván Dotú
Gregory J. Duck
Nizar El Hachemi
Boi Faltings
Alex Ferguson
Spencer K.L. Fung
Maria Garcia de la Banda
Jonathan Gaudreault
Marco Gavanelli
Ian Gent
Simon de Givry
Alexandre Goldsztejn
Carla Gomes
Laurent Granvilliers
Martin Green
Justin W. Hart
Warwick Harvey
Emmanuel Hebrard
Federico Heras
Katsutoshi Hirayama
John Hooker
Holger Hoos
Luc Jaulin
Peter Jeavons
Christopher Jefferson
Peter Jonsson
Sun Jun
Irit Katriel
Tom Kelsey
Zeynep Kiziltan
R. Baker Kearfott
András Kovács
Ludwig Krippahl
Andrei Krokhin
Fredrik Kuivinen
Frédéric Lardeux
Javier Larrosa
Yat-Chiu Law
Yahia Lebbah
Jimmy Lee
Olivier Lhomme
Wei Li
Chavalit Likitvivatanavong
Santiago Macho González
Michael Maher
Felip Manya
Deepak Mehta
Luc Mercier
Pedro Meseguer
Laurent Michel
Ian Miguel
Michela Milano
Belaid Moa
Jay Modi
Eric Monfroy
João Moura Pires
Yehuda Naveh
Bertrand Neveu
Peter Nightingale
Gustav Nordh
Tomas Eric Nordlander
Barry O'Sullivan
Gilles Pesant
Thierry Petit
Karen Petrie
Cédric Pralet
Benoit Pralong
Nicolas Prcovic
Steven Prestwich
Patrick Prosser
Gregory M. Provan
Jakob Puchinger
Jean-François Puget
Claude-Guy Quimper
Stefan Ratschan
Igor Razgon
Jean-Charles Régin
Guillaume Rochart
Andrea Roli
Emma Rollon
Colva Roney-Dougal
Francesca Rossi
Louis-Martin Rousseau
Michel Rueher
Ashish Sabharwal
Jamila Sam-Haroud
Frédéric Saubion
Thomas Schiex
Joachim Schimpf
Tom Schrijvers
Christian Schulte

Andrew See
Meinolf Sellmann
Paul Shaw
Marius Silaghi
Helmut Simonis
Charles Siu
Barbara Smith
Fred Spiessens
Peter Stuckey
Radoslaw Szymanek
Guido Tack
Vincent Tam
Cyril Terrioux
Tan Tiow Seng
Dave Tompkins
Charlotte Truchet
Chris Unsworth
Peter van Beek
M.R.C. van Dongen
Maarten van Emden
Pascal Van Hentenryck
Willem-Jan van Hoeve
K. Brent Venable
Gérard Verfaillie
Guillaume Verger
Xuan-Ha Vu
Mark Wallace
Richard Wallace
Toby Walsh
Nic Wilson
Roland Yap
Makoto Yokoo
Neil Yorke-Smith
Alessandro Zanarini
Bruno Zanuttini
Weixiong Zhang
Yuanlin Zhang
Xing Zhao
Neng-Fa Zhou
Kenny Zhu
Mikael Z. Lagerkvist
Matthias Zytnicki

Sponsoring Institutions

Association Française pour la Programmation par Contraintes
Conseil Général de Loire-Atlantique
Cork Constraint Computation Centre
École des Mines de Nantes
Ilog
Intelligent Information System Institute, Cornell Univ.
Laboratoire d'Informatique de Nantes-Atlantique
Nantes Métropole
Région Pays de la Loire
Université de Nantes

Table of Contents

Global Optimization of Probabilistically Constrained Linear Programs

Shabbir Ahmed

H. Milton Stewart School of Industrial & Systems Engineering
Georgia Institute of Technology
sahmed@isye.gatech.edu

Various applications in reliability and risk management give rise to optimization problems where certain constraints involve stochastic parameters and are required to be satisfied with a pre-specified probability threshold. In this talk we address such probabilistically constrained linear programs involving stochastic right-hand-sides. These problems involve non-convex feasible sets, and are extremely difficult to optimize.

In the first part of the talk, we reveal monotonicity properties inherent in the problem, and exploit these to develop a global optimization algorithm. The proposed approach is a branch-and-bound algorithm that searches for a global optimal solution by successively partitioning the non-convex feasible region and by using bounds on the objective function to fathom inferior partition elements. This basic algorithm is enhanced by domain reduction and cutting plane strategies to reduce the size of the partition elements and hence tighten bounds. The algorithm, which requires solving deterministic linear programming subproblems, is proved to converge to a global optimal solution.

In the second part of the talk, we address probabilistically constrained linear programs under discrete distribution of the right-hand-side parameters. These problems can be reformulated as mixed integer linear programs. We perform a detailed polyhedral study of the convex hull of the set of feasible solutions of such problems, and develop families of strong valid inequalities. In the case of a single probabilistic constraint, these inequalities suffice to describe the convex hull of the feasible set. In the general case, the inequalities, when used within a branch-and-cut scheme, serve to significantly improve relaxation bounds, and expedite convergence.

The first part of the talk is based on joint work Myun-Seok Cheon and Faiz Al-Khayyal, and the second part of the talk is based on joint work with James Luedtke and George Nemhauser.

Algorithms and Constraint Programming[⋆]

Fabrizio Grandoni[1] and Giuseppe F. Italiano[2]

[1] Dipartimento di Informatica, Università di Roma "La Sapienza", via Salaria 113, 00198 Roma, Italy
grandoni@di.uniroma1.it

[2] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy
italiano@disp.uniroma2.it

Abstract. Constraint Programming is a powerful programming paradigm with a great impact on a number of important areas such as logic programming [45], concurrent programming [42], artificial intelligence [12], and combinatorial optimization [46]. We believe that constraint programming is also a rich source of many challenging algorithmic problems, and cooperations between the constraint programming and the algorithms communities could be beneficial to both areas.

1 Introduction

Given a set of variables $\mathcal{X}$, and a set of constraints $\mathcal{C}$ forbidding some partial assignments of variables, the NP-hard *Constraint Satisfaction Problem* (CSP) is to find an assignment of all the variables which satisfies all the constraints [37].

One of the most common ways to solve CSPs is via *backtracking*: given a partial assignment of variables, extend it by instantiating some other variable in a way compatible with the previous assignments. If this is not possible, backtrack and try a different partial assignment. This standard approach can be improved in several ways:

- **(improved search)** instead of backtracking to the previously instantiated variable, one can backtrack to the variable generating the conflict (*backjumping*), and try to avoid such conflict later (*backchecking* and *backmarking*).
- **(domain filtering)** consistency properties which feasible assignments need to satisfy can be used to filter out part of the values in the domains, thus reducing the search space. This can be done in a preprocessing step, or during the search, both for specific and for arbitrary sets of constraints.

⋆ This work has been partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet") and by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT ("Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments").

- **(variables/values ordering)** The order in which variables and values are considered during the search can heavily affect the time needed to find a solution. There are several heuristics to find a convenient, static or dynamic, ordering of variables and values (*fail-first*, *succeed-first*, *most-constrained*, etc.).

In this paper we focus on the last two strategies, and we show how algorithmic techniques can be helpful. In Section 3 we will describe two polynomial-time filtering algorithms. The first one, which is based on matching, can be used for filtering of the well-known `alldifferent` constraint. The second one uses techniques from dynamic algorithms to speed up the filtering based on *inverse-consistency*, a consistency property which can be applied to arbitrary sets of binary constraints.

In Section 4 we will present an exact (exponential-time) algorithm to solve any CSP asymptotically faster than with trivial enumeration. As we will see, the improved running time is heavily based on the way variables and values are instantiated. However, in this case the approach is not heuristic: the running time is guaranteed on any instance.

2 Preliminaries

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of n variables. Given $x \in \mathcal{X}$, by $D(x)$ we denote the domain of x. From now on we will assume that each domain is discrete and finite.

An *assignment* of a variable $x \in \mathcal{X}$ is a pair (x, a), with $a \in D(x)$, whose meaning is that x is assigned value a. A *constraint* C is a set of assignments of different variables:

$$C = \{(x_{i(1)}, a_1), (x_{i(2)}, a_2) \ldots (x_{i(h)}, a_h)\}.$$

Constraint C is said to be *satisfied* by an assignment of the variables if there exists one variable $x_{i(j)}$ such that $x_{i(j)} \neq a_j$, and it is said to be *violated* otherwise.

Remark 1. For ease of presentation, in this paper we use the explicit representation of constraints above. However, implicit representations are more common in practice.

Given $\mathcal{X}$ and a set $\mathcal{C}$ of constraints, the *Constraint Satisfaction Problem* (CSP) is to find an assignment of values to variables (*solution*) such that all the constraints are satisfied. We only mention that there are two relevant variants of this problem:

- list all the solutions;
- find the best solution according to some objective function.

Some of the techniques we are going to describe partially extend to such cases.

A (d, p)-CSP is a CSP where each domain contains at most d values, and each constraint involves at most p variables. Without loss of generality, we can consider $(d, 2)$-CSPs only (also called *binary* CSPs), as the following simple lemma shows.

Lemma 1. *Each (d, p)-CSP instance can be transformed into an equivalent $(\max\{d, p\}, 2)$-CSP instance in polynomial time.*

Proof. Duplicate the variables $\mathcal{X}$ and add a variable c for each constraint $C = \{(x_{i(1)}, a_1), (x_{i(2)}, a_2), \ldots, (x_{i(h)}, a_h)\}$, $h \leq p$. The domain of c is

$$D(c) = \{(x_{i(1)} \neq a_1), (x_{i(2)} \neq a_2), \ldots, (x_{i(h)} \neq a_h)\}.$$

Intuitively, assigning the value $(x_{i(j)} \neq a_j)$ to c means that constraint C is satisfied thanks to the fact that $x_{i(j)} \neq a_j$. For each such c we also add h constraints $C_1, C_2, \ldots, C_h$ of the following form:

$$C_j = \{(c, (x_{i(j)} \neq a_j)), (x_{i(j)}, a_j)\}.$$

The intuitive explanation of C_j is that it cannot happen that $x_{i(j)} = a_j$ and, at the same time, constraint C is satisfied thanks to the fact that $x_{i(j)} \neq a_j$. It is not hard to see that the new problem is satisfiable if and only if the original problem is. □

Remark 2. Dealing with non-binary constraints directly, without passing through their binary equivalent, might be convenient in some applications [44].

It is also worth to mention that there is a nice duality between variables and constraints.

Lemma 2. *Each (d, p)-CSP on n variables and m constraints can be transformed in polynomial-time into an equivalent (p, d)-CSP on m variables and n constraints.*

Proof. For each constraint $C = \{(x_{i(1)}, a_1), (x_{i(2)}, a_2) \ldots (x_{i(h)}, a_h)\}$, $h \leq p$, create a variable c of domain:

$$D(c) = \{(x_{i(1)} \neq a_1), (x_{i(2)} \neq a_2), \ldots, (x_{i(h)} \neq a_h)\}.$$

The interpretation of c is as in Lemma 1. Now consider any variable x of the original problem. If there exists $a \in D(x)$ such that the assignment (x, a) does not conflict with any constraint, do not add any constraint for x. Note that in such case, if there is a solution, there is a solution with $x = a$. Otherwise, for each $i \in D(x) = \{1, 2, \ldots, d(x)\}$, take a variable c_i such that $(x \neq i) \in D(c_i)$. Add the constraint

$$X = \{(c_1, (x \neq 1)), (c_2, (x \neq 2)), \ldots, (c_{d(x)}, (x \neq d(x)))\}.$$

The intuitive explanation of X is that x must take some value in its domain. It is not hard to see that the original problem is satisfiable if and only if the original problem is. □

Note that each $(d, 2)$-CSP can be represented via a *consistency graph* which has a node for each possible assignment (x, a) and an edge between each pair of compatible assignments (anti-edges correspond to constraints or to multiple assignments of the same variable). Any solution corresponds to an n-clique in such graph (see Figure 1).

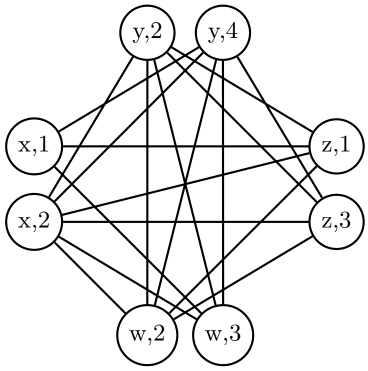


Fig. 1. Example of consistency-graph. A solution is given by the assignments $\{(x,2),(y,2),(z,1),(w,2)\}$. The assignment $(x,1)$ is arc consistent, while it is not path-inverse consistent.

3 Polynomial Algorithms and Domain Filtering

An assignment (x,a) is *consistent* if it belongs to some solution, and *inconsistent* otherwise. Deciding whether an assignment is inconsistent is an NP-hard problem (otherwise one could solve CSP in polynomial time [37]). However, it is sometimes possible to filter out (part of the) inconsistent assignments efficiently. In this section we give two examples of how algorithmic techniques can be used in the filtering process. In Section 3.1 we discuss the filtering of the well-known (non-binary) `alldifferent` constraint, which makes use of matching algorithms. In Section 3.2 we consider the filtering based on *ℓ-inverse-consistent*, suitable for any set of binary constraints, and we present a faster algorithm employing standard techniques from dynamic algorithms.

3.1 Alldifferent Filtering Via Matching

In this paper we focus on binary constraints. However, there are families of non-binary constraints which appear very frequently in the applications. So it makes sense to design faster and more accurate filtering algorithms for them.

A relevant example is the `alldifferent` constraint, which requires that a set of variables take values different from each other. The `alldifferent` constraint is very powerful. For example with only 3 such constraints on a proper set of variables one can naturally model the well-known *n-queens problem*: place n queens on a $n \times n$ chessboard such that no two queens threaten each other (a queen threatens any other queen on the same row, column, diagonal and anti-diagonal).

The `alldifferent` constraint has the great advantage that the consistency of the assignments can be decided in polynomial time, with the following procedure by Regin [40]. Consider the bipartite graph B, which has the variables on the left side, the values on the right side, and one edge between x and a for each possible assignment (x, a). Without loss of generality, let us assume the values available are at least as many as the variables (otherwise the problem has trivially no solution). Then any feasible solution to the original problem corresponds to a perfect bipartite matching in B, that is a matching where all the variables are matched. Luckily, we do not need to compute explicitly all the perfect bipartite matchings to determine whether a given assignment (x, a) belongs to any one of them. In fact, let M be any perfect bipartite matching. Such matching can be computed in time $O(m'\sqrt{n'})$, where n' is the number of nodes and m' the number of edges of B [29]. Let us direct all the edges in M from right to left, and all the other edges from left to right. Then an edge $(x, a) \notin M$ belongs to some other perfect matching M' if and only if

- (x, a) belongs to an oriented cycle or
- it belongs to an even-length oriented path, starting in a free node on the right side.

We can check the two properties above for all the edges in linear time $O(n'+m')$. Altogether, we can find the subset of consistent assignments in time $O(m'\sqrt{n'})$.

In many applications the variables range over intervals. If such intervals are very large, the approach above becomes unpractical. However, there is a convenient alternative in such case: computing the largest subinterval for each variable such that both endpoints correspond to consistent assignments (*narrowing* of the intervals). Puget [38] observed that the bipartite graph B corresponding to the `alldifferent` constraint is *convex* if the variables range over intervals. Thus one can compute a perfect bipartite matching in $O(n \log n)$ time via the matching algorithm by Glover [25] for convex bipartite graphs. Puget use this observation to narrow the `alldifferent` constraint within the same time bound. Later Mehlhorn and Thiel [34] described an algorithm which takes linear time plus the time to sort the intervals endpoints. Their algorithm makes use of the union-find data structure by Gabow and Tarjan [24]. This improves on the result by Puget in all the cases where sorting can be done in linear time.

3.2 Inverse Consistency and Decremental Clique Problem

Most of the filtering techniques designed for arbitrary binary constraints are based on some kind of *local consistency property* $\mathcal{P}$, which all the consistent assignments need to satisfy. Enforcing $\mathcal{P}$-consistency is a typical dynamic process: an assignment (x, a) which is initially $\mathcal{P}$-consistent may become inconsistent because of the removal of some other assignment (y, b). Thus the same (x, a) might be checked several times. Using the information gathered during the previous consistency-checks can speed up the following checks. This is typically what happens in dynamic algorithms, and so it makes sense trying to apply the techniques developed in that area to speed up the filtering process (for references on dynamic algorithms, see e.g. [13]).

Maybe the simplest and most studied local consistency property is *arc-consistency* [33]. An assignment (x, a) is arc-consistent if, for every other variable y, there is at least one assignment (y, b) compatible with (x, a). The assignment (y, b) is a *support* for (x, a) on variable y. Clearly, if an assignment is not arc-consistent, it cannot be consistent (unless x is the unique variable). Arc-consistency can be naturally generalized. An assignment (x, a) is *path-inverse consistent* [17] if it is arc-consistent and, for every two other distinct variables y and z, there are assignments (y, b) and (z, c) which are mutually compatible and compatible with (x, a). We say that $\{(y, b), (z, c)\}$ is a *support* for (x, a) on $\{y, z\}$ The ℓ-inverse consistency [17] is the natural generalization of arc-consistency ($\ell = 2$) and path-inverse consistency ($\ell = 3$) to arbitrary (fixed) values of $\ell \leq n$.

There is a long series of papers on arc-consistency [3,4,5,33,35]. The currently fastest algorithm has running time $O(e\,d^2)$, where e denotes the number of distinct pairs of variables involved in some constraint. For long time the fastest known ℓ-inverse-consistency-based filtering algorithm, for $\ell \geq 3$, was the $\mathrm{O}(e\,n^{\ell-2}d^{\ell})$ algorithm by Debruyne [10].

Remark 3. The quantity $e\,n^{\ell-2}$, corresponding to the number of distinct subsets of ℓ pairwise constrained variables, can be replaced by the tighter $e^{\ell/2}$, given by a combinatorial lemma by Erdős [19].

The algorithm in [10] is based on a rather simple dynamic strategy to check whether an assignment is ℓ-inverse consistent. Roughly, the idea is to sort the candidate supports of any assignment (x, a) on any subset of other $(\ell-1)$ distinct variables in an arbitrary way, and then follow such order while searching for supports for (x, a). Moreover, the last supports found are maintained: this way, if a support for (x, a) is deleted and a new one is needed, the already discarded candidates are not reconsidered any more.

The algorithm in [10] was conjectured to be asymptotically the fastest possible. In this section we review an asymptotically faster algorithm for $\ell \geq 3$ [27], based on standard techniques from dynamic algorithms. For the sake of simplicity, let us consider the case of path-inverse-consistency ($\ell = 3$). The same approach extends to larger values of ℓ. Consider any triple of pairwise constrained variables $\{x, y, z\}$, and let $G_{x,y,z}$ be the graph whose nodes are the assignments of x, y and z, and whose edges are the pairs of compatible assignments (i.e. $G_{x,y,z}$ is the restriction of the consistency graph to variables x, y, and z). Any assignment (x, a) is path-inverse-consistent with respect to variables y and z if and only if (x, a) belongs to at least one triangle of $G_{x,y,z}$. More precisely, the number of supports for (x, a) on $\{y, z\}$ is exactly the number of triangles of $G_{x,y,z}$ which contain (x, a).

Thus a natural approach to enforce path-inverse-consistency is to count all the supports for (x, a) on $\{y, z\}$ initially, and then update the count each time a support is deleted. If we scan all the candidate supports, the initial counting costs $O(d^3)$. Since there can be at most $O(d)$ deletions, and listing all the triangles that contain a given deleted value costs $O(d^2)$, the overall cost of this approach is $O(d^3)$. Since the graphs $G_{x,y,z}$ are $O(e^{1.5})$, this approach has cost $O(e^{1.5}d^3)$, the same as with Debruyne's algorithm.

We next show how to speed up both the initial counting and the later updating by using fast matrix multiplication and lazy updating, two techniques which are widely used in dynamic algorithms [14,15]. By A we denote the adjacency matrix of G. Given an assignment $i = (x, a)$, the number of triangles in which i is contained is $t(i) = (1/2)A^3[i, i]$. Hence, as first observed by Itai and Rodeh [31], the quantities $t(i)$'s can be computed in time $O(d^\omega)$, where $\omega < 2.376$ is fast square matrix multiplication exponent [8].

It remains to show how to maintain the counting under deletion of nodes. We use the following idea. In time $O(d^\omega)$ we can also compute for each edge $\{i, j\}$, the number $t(i, j)$ of triangles which contain $\{i, j\}$. The number of triangles $t(i)$ containing i is one half times the sum of the $t(i, j)$'s over all the edges $\{i, j\}$ incident to i. Now suppose we remove a neighbor j of i. Then, in order to update $t(i)$, it is sufficient to subtract $t(i, j)$. Suppose that we later remove another neighbor k of i. This time subtracting $t(i, k)$ is not correct any more, since we could subtract the triangle $\{i, j, k\}$ twice. However, we can subtract $t(i, k)$ and then add one if $\{i, j, k\}$ is a triangle. This argument can be generalized. Suppose we already deleted a subset of nodes D, and now we wish to delete a neighbor j of i. Then, in order to update $t(i)$, we first subtract $t(i, j)$ and then we add one for each $k \in D$ such that $\{i, j, k\}$ is a triangle. This costs $O(|D|)$ for each update. Altogether maintaining the counting costs $O(d|D|)$ per deletion of node.

When $|D|$ becomes too large, say $|D| \geq d^\epsilon$ for some $\epsilon \in (0, 1)$, this approach is not convenient any more. However in that case we can update all the $t(i, j)$'s, and empty D. In order to update the $t(i, j)$'s, we need to compute all the 2-length paths passing through a node in D. This costs $O(d^{\omega\epsilon+2(1-\epsilon)})$, that is the time to multiply a $d \times d^\epsilon$ matrix by a $d^\epsilon \times d$, where the multiplication is performed by decomposing the rectangular matrices in square pieces, and using square matrix multiplication. Since we perform such update every d^ϵ deletions, the amortized cost per deletion is $O(d^{2+\epsilon(\omega-3)})$. Balancing the terms $O(d^{2+\epsilon(\omega-3)})$ and $O(d^{1+\epsilon})$, one obtains an overall $O(d^{1+1/(4-\omega)}) = O(d^{1.616})$ amortized cost per deletion. Using more sophisticated rectangular matrix multiplication algorithms [30] the running time can be reduced to $O(d^{2.575})$. This leads to the following result.

Theorem 1. *Path-inverse consistency can be enforced in time* $O(e^{1.5}d^{2.575})$.

Remark 4. Depending on the size and density of the matrices involved, it might be convenient in practice to use matrix-multiplication algorithms different from the fastest asymptotic ones.

Another important consistency property is *max-restricted path consistency.* The same basic approach as above allows one to reduce the time to enforce max-restricted path consistency from $O(e^{1.5}d^3)$ [11] to $O(e^{1.5}d^{2.575})$ [26].

4 Exact Algorithms in Variables/Values Ordering

The classical approach to solve (exactly) NP-hard problems is via heuristics. Although heuristics are very useful in practice, they suffer from few drawbacks. First, they do not guarantee worst-case running times (better than the trivial bounds). For example, the worst-case running time to solve a $(d, 2)$-CSP instance on n variables is (implicitly) assumed to be $\Omega(d^n)$, that is the time bound achieved with exhaustive search. This can be problematic in critical applications, where the worst-case running time matters. Moreover, since the relative performance of heuristics can be assessed only empirically, it is often difficult to compare different heuristic approaches for the same problem.

A potential way to overcome those limits is offered by the design of *exact algorithms*, an area which attracted growing interest in the last decade. The aim of exact algorithms is to solve NP-hard problems in the minimum possible (exponential) worst-case running time. Exact algorithms have several merits:

- The measure of performance is theoretically well-defined: comparing different algorithms is easy.
- The running time is guaranteed on any input, not only on inputs tested experimentally.
- A reduction of the base of the exponential running time, say, from $O(2^n)$ to $O(2^{0.9\,n})$, increases the size of the instances solvable within a given amount of time by a constant *multiplicative* factor; running a given exponential algorithm on a faster computer can enlarge the mentioned size only by a constant *additive* factor.
- The design and analysis of exact algorithms leads to a deeper insight in NP-hard problems, with a positive long-term impact on the applications.

There are exact algorithms faster than trivial approaches for a number of problems such as: TSP [18,28], maximum independent set [1,22,41], minimum dominating set [20], coloring [2,7], satisfiability [6,9], maximum cut [47], feedback vertex set [39], Steiner tree [16,36], treewidth [23], and many others. For more references, see e.g. [21,32,43,48,49].

To show the potentialities of exact algorithms, we will describe an exact deterministic algorithm which solves any $(d, 2)$-CSP on n variables in time $O((1 + \lfloor d/3 \rfloor 1.3645)^n)$, thus breaking the $\Omega(d^n)$ barrier given by trivial enumeration.

In order to achieve such running time, we first describe a faster algorithm to solve $(3, 2)$-CSPs, and we later show how to use it to speed up the solution of arbitrary $(d, 2)$-CSPs. Note that $(3, 2)$-CSP is an interesting problem in its own, since it includes as special cases important problems like 3-coloring and 3-SAT via Lemma 2.

We need the following two observations.

Lemma 3. (reduction) *[2] Consider a variable x of a $(d, 2)$-CSP such that $|D(x)| \leq 2$. Then there is a polynomial-time computable equivalent $(d, 2)$-CSP with one less variable.*

Proof. If $D(x) = \{a\}$, it is sufficient to remove variable x, and each value conflicting with (x, a) in the domains of the other variables. So, let us assume $D(x) = \{a, b\}$. In such case remove x and add the following set of constraints: for every (y, a') conflicting with (x, a) and for every (z, b') conflicting with (x, b), $y \neq z$, add the constraint $\{(y, a'), (z, b')\}$. In fact, setting $y = a'$ and $z = b'$ would rule out any possible assignment for x. On the other direction, any solution to the new problem can be extended to a solution for the original problem by assigning either value a or b to x. □

Lemma 4. (domination) *Consider any $(d, 2)$-CSP. Let $a, b \in D(x)$, for some variable x, and let A and B be the set of assignments of other variables conflicting with (x, a) and (x, b), respectively. If $A \subseteq B$, then an equivalent CSP is obtained by removing b from $D(x)$.*

Proof. Suppose there is a solution where $x = b$. Then, by switching x to a, the solution remains feasible. □

Remark 5. The two properties above cannot be applied if the aim is to compute all the solutions, or the best solution according to some objective function.

We are now ready to describe our improved algorithm for $(3, 2)$-CSP, which consists of the following steps.

1. **(filtering)** Exhaustively apply arc-consistency and domination to reduce the domains.
2. **(base)** If there a variable with an empty domain, return *no*. Otherwise, if there is at most one variable, return *yes*.
3. **(reduction)** If there is a domain $D(x)$ of cardinality at most 2, remove variable x according to Lemma 3, and branch one the problem obtained.
4. **(branch 1)** If there is a constraint $\{(x, a), (y, b)\}$, where (y, b) is not involved in other constraints, branch by either *selecting* a (i.e., restricting $D(x)$ to $\{a\}$) or *discarding* a (i.e., removing a from $D(x)$).
5. **(branch 2)** Otherwise, take a pair (x, a) involved in constraints with the maximum possible number of distinct variables. Branch by either selecting or discarding a.

By *branching* on a set of subproblems, we mean solve them recursively, and return *yes* if and only if the answer of any one of the subproblems is *yes*.

Lemma 5. *The algorithm above solves any $(3, 2)$-CSP instance in worst-case time $O(1.466^n)$.*

Proof. We define an instance *ground* if the algorithm solves it without branching on two subproblems (hence in polynomial-time). By $P(\mathcal{X}, \mathcal{C})$ we denote the total number of ground instances solved to solve a given $(3, 2)$-CSP instance $(\mathcal{X}, \mathcal{C})$. Let $P(n)$ be the maximum of $P(\mathcal{X}, \mathcal{C})$ over all the instances on n variables. We will show that $P(n) \leq 1.4656^n$. The claim follows by observing that generating each subproblem costs only polynomial time, excluding the cost of the recursive calls, and the total number of subproblems generated is within a polynomial from $P(n)$. Hence the total running time is $O(1.4656^n\, n^{O(1)}) = O(1.466^n)$.

We proceed by induction on n. Trivially, $P(0) = P(1) = 1$ satisfy the claim. Now assume the claim is true up to $(n - 1) \geq 1$ variables, and consider any instance $(\mathcal{X}, \mathcal{C})$ on $n \geq 2$ variables. We distinguish different cases, depending on the step where the algorithm branches:

(base). We do not generate any subproblem

$$P(\mathcal{X},\mathcal{C}) \leq 1 \leq 1.4656^n.$$

(reduction). We generate exactly one subproblem with one less variable:

$$P(\mathcal{X},\mathcal{C}) \leq P(n-1) \leq 1.4656^{n-1} \leq 1.4656^n.$$

(branch 1). In both subproblems variable x is removed by Step 3. When we select a, we remove b from $D(y)$ by arc-consistency, and hence variable y by Step 3. When we discard a, we remove a value $c \in D(x) \setminus \{b\}$ by dominance, being c dominated by b. So also in that case y is later removed by Step 3. Altogether

$$P(\mathcal{X},\mathcal{C}) \leq P(n-2) + P(n-2) \leq 2 \cdot 1.4656^{n-2} \leq 1.4656^n.$$

(branch 2). By basically the same arguments as above, if (x,a) is involved in constraints with at least two other variables y and z, when we branch by discarding a, we remove at least variable x, while when we branch by selecting a, we remove at least variables x, y and z. Hence

$$P(\mathcal{X},\mathcal{C}) \leq 1.4656^{n-1} + 1.4656^{n-3} \leq 1.4656^n.$$

Otherwise, by Steps 1 and 4, and by a simple case analysis, there must be a set of 6 constraints involving x and y of the following form: $\{(x,a),(y,b')\}$, $\{(x,a),(y,c')\}$, $\{(x,b),(y,b')\}$, $\{(x,c),(y,c')\}$, $\{(x,b),(y,a')\}$, and $\{(x,c),(y,a')\}$. When we select a, we remove variable x, values b' and c' from $D(y)$ by arc-consistency, and later variable y by Step 3. When we discard a, we remove variable x, value a' by dominance, and later variable y by Step 3. Thus

$$P(\mathcal{X},\mathcal{C}) \leq 2 \cdot 1.4656^{n-2} \leq 1.4656^n.$$

The claim follows. □

By using similar, but more sophisticated, arguments, Beigel and Eppstein showed that any $(3,2)$-CSP on n variables can be solved in worst-case time $O(1.36443^n)$ [2].

Consider now the following algorithm to solve any $(d,2)$-CSP, which can be interpreted as a derandomization of a result in [2]. For each variable x, partition $D(x)$ in $\lceil d/3 \rceil$ subsets of cardinality at most 3, and branch by restricting the domain of x to each one of the mentioned subsets. When all the domains are restricted in that way, solve the instance obtained with the mentioned $O(1.36443^n)$ algorithm for $(3,2)$-CSP. Return *yes* if and only if one of the subproblems generated is feasible.

Theorem 2. *Any $(d,2)$-CSP, $d \geq 3$, can be solved in $O(\alpha^n)$ worst-case time, where $\alpha = \alpha(d) = \min\{\lceil d/3 \rceil 1.3645, 1 + \lfloor d/3 \rfloor 1.3645\}$.*

Proof. For the sake of simplicity, assume all the domains have size d. This can be achieved by adding dummy variables. Consider the algorithm above. Its running time is trivially

$$O(\lceil d/3 \rceil^n \, 1.36443^n n^{O(1)}) = O(\lceil d/3 \rceil^n \, 1.3645^n).$$

When d is not a multiple of 3 a better time bound is achieved by observing that the partition of each $D(x)$ contains $\lfloor d/3 \rfloor$ sub-domains of size 3, and one sub-domain of size $d \pmod 3 \in \{1,2\}$. Hence the algorithm generates $\binom{n}{i}$ problems containing i domains of cardinality at most 2, and $n-i$ domains of cardinality 3. The variables corresponding to the i small domains can be removed without branching. Hence the running time is

$$O(\sum_i \binom{n}{i} \lfloor d/3 \rfloor^{n-i} 1.36443^{n-i} n^{O(1)}) = O((1 + \lfloor d/3 \rfloor 1.3645)^n).$$

□

Remark 6. A different algorithm is obtained by partitioning the domains in sub-domains of size 2 instead of 3, and then branching as in the algorithm above. Since each subproblem created can be solved in polynomial time, the overall running time is $O(\lceil d/2 \rceil^n n^{O(1)})$. This improves on the previous result for $d = 4$ and $d = 10$.

References

1. R. Beigel. Finding maximum independent sets in sparse and general graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 856–857, 1999.
2. R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms*, 54:168–204, 2005.
3. C. Bessière. Arc-consistency and arc-consistency again. In *National Conference on Artificial Intelligence (AAAI)*, pages 179–190, 1994.
4. C. Bessière and M. O. Cordier. Arc-consistency and arc-consistency again. In *National Conference on Artificial Intelligence (AAAI)*, pages 108–113, 1993.
5. C. Bessière, E. Freuder, and J. C. Regin. Using inference to reduce arc consistency computation. In *International Joint Conference on Artificial Intelligence*, pages 592–598, 1995.
6. T. Brueggemann and W. Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329:303–313, 2004.
7. J. M. Byskov. Enumerating maximal independent sets with applications to graph colouring. *Operations Research Letters*, 32:547–556, 2004.
8. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.
9. E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.
10. R. Debruyne. A property of path inverse consistency leading to an optimal PIC algorithm. In *European Conference on Artificial Intelligence*, pages 88–92, 2000.

11. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Principles and Practice of Constraint Programming (CP)*, pages 312–326, 1997.
12. R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.
13. C. Demetrescu, I. Finocchi, G. F. Italiano, "Dynamic Graphs", Chapter 22, *Handbook of Data Structures and Applications*, CRC Press, 2004.
14. C. Demetrescu, G. F. Italiano, "Trade-Offs for Fully Dynamic Transitive Closure on DAGs: Breaking Through the $O(n^2)$ Barrier", *Journal of the ACM*, vol. 52, no. 2, March 2005, 147–156.
15. C. Demetrescu, G. F. Italiano, "Dynamic Shortest Paths and Transitive Closure: Algorithmic Techniques and Data Structures", *Journal of Discrete Algorithms*, vol 4 (3), September 2006.
16. S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1971/72.
17. C. D. Elfe and E. C. Freuder. Neighborhood inverse consistency preprocessing. In *National Conference on Artificial Intelligence (AAAI)/Innovative Applications of Artificial Intelligence*, volume 1, pages 202–208, 1996.
18. D. Eppstein. The traveling salesman problem for cubic graphs. In *Workshop on Algorithms and Data Structures (WADS)*, pages 307–318, 2003.
19. P. Erdős. On the number of complete subgraphs contained in certain graphs. *Magyar Tudomanyos Akademia Matematikai Kutató Intezetenek Közlemenyei*, 7:459–464, 1962.
20. F. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: domination - a case study. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 191–203, 2005.
21. F. Fomin, F. Grandoni, and D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the European Association for Theoretical Computer Science*, 87:47–77, 2005.
22. F. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: A simple $o(2^{0.288\,n})$ independent set algorithm. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 18–25, 2006.
23. F. V. Fomin, D. Kratsch, and I. Todinca. Exact algorithms for treewidth and minimum fill-in. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 568–580, 2004.
24. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *ACM Symposium on the Theory of Computing (STOC)*, pages 246–251, 1983.
25. F. Glover. Maximum matching in convex bipartite graph. *Naval Research Logistic Quarterly*, 14:313–316, 1967.
26. F. Grandoni and G. F. Italiano. Improved algorithms for max-restricted path consistency. In *Principles and Practice of Constraint Programming (CP)*, pages 858–862, 2003.
27. F. Grandoni and G. F. Italiano. Decremental clique problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 142–153, 2004.
28. M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of SIAM*, 10:196–210, 1962.
29. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

30. X. Huang and V. Pan. Fast rectangular matrix multiplication and applications. *Journal of Complexity*, 14(2):257–299, 1998.
31. A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.
32. K. Iwama. Worst-case upper bounds for k-SAT. *Bulletin of the European Association for Theoretical Computer Science*, 82:61–71, 2004.
33. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
34. K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In R. Dechter, editor, *Principles and Practice of Constraint Programming (CP)*, pages 306–319, 2000.
35. R. Mohr and T. C. Henderson. Arc and path consistency revised. *Artificial Intelligence*, 28:225–233, 1986.
36. D. Mölle, S. Richter, and P. Rossmanith. A faster algorithm for the Steiner tree problem. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 561–570, 2006.
37. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
38. J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *National Conference on Artificial Intelligence (AAAI)/Innovative Applications of Artificial Intelligence*, pages 359–366, 1998.
39. I. Razgon. Exact computation of maximum induced forest. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, 2006. To appear.
40. J. C. Regin. A filtering algorithm for constraints of difference in CSP. In *National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 362–367, 1994.
41. J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.
42. V. A. Saraswat. *Concurrent Logic Programming Languages.* PhD thesis, Carnegie-Mellon University, 1987.
43. U. Schöning. Algorithmics in exponential time. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 36–43, 2005.
44. K. Stergiou. *Representation and Reasoning with Non-Binary Constraints.* PhD thesis, University of Strathclyde, Glasgow, Scotland, 2001.
45. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.
46. P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.
47. R. Williams. A new algorithm for optimal constraint satisfaction and its implications. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1227–1237, 2004.
48. G. J. Woeginger. Exact algorithms for np-hard problems: A survey. In M. Jünger, G. Reinelt, and G. Rinaldi, editors, *International Workshop on Combinatorial Optimization – Eureka, You Shrink*, number 2570 in Lecture Notes in Computer Science, pages 185–207. Springer-Verlag, 2003.
49. G. J. Woeginger. Space and time complexity of exact algorithms: Some open problems. In *International Workshop on Parameterized and Exact Computation (IWPEC)*, pages 281–290, 2004.

Interval Analysis and Robotics

Jean-Pierre Merlet

Institut National de Recherche en Informatique et Automatique (INRIA)
BP 93
06902 Sophia-Antipolis Cedex, France
Jean-Pierre.Merlet@sophia.inria.fr

Robotics is a field in which numerous linear and non linear problems are occuring. The unknowns of these problems have a physical meaning and roboticians are usually interested only in solutions within restricted bounds. Furthermore dealing with uncertainties is unavoidable as any robot is a controlled mechanical system with manufacturing and control errors. Hence interval analysis is a tool of choice for solving many problems in robotics and managing uncertainties while providing certified answers (the reliability of the result is very often a critical aspect, for example in medical robotics). In this talk we will exemplify how interval anlysis may be used to efficently solve systems of equations appearing in the geometrical modeling of robots, to check the regularity of parametrized interval matrices that is required for singularity analysis and to design robots so that their performances will meet pre-defined requirements whatever are the manufacturing errors of the real robot within reasonable ranges.

Constraint Based Resilience Analysis

Helmut Simonis

CrossCore Optimization Ltd, London, UK
helmut.simonis@crosscoreop.com
http://www.crosscoreop.com

Abstract. In this paper we give an overview of applications of Constraint Programming for IP (Internet Protocol) data networks, and discuss the problem of *Resilience Analysis* in more detail. In this problem we try to predict the loading of a network in different failure scenarios, without knowing end-to-end flow values throughout the network; the inference is based only on observed link traffic values. The related problem of *Traffic Flow Analysis* aims to derive a traffic matrix from the observed link traffic data. This is a severely under-constrained problem, we can show that the obtained flow values vary widely in different, feasible solutions. Experimental results indicate that using the same data much more accurate, bounded results can be obtained for *Resilience Analysis*.

1 Introduction

In this paper we discuss the use of Constraint Programming (CP) for IP (Internet Protocol) data network applications. The bulk of constraint applications for networks [40] are in the context of data networks, covering either traditional, connection oriented networks or packet-switched, routed networks like the Internet. The survey [40] classifies them into a number of different groups:

– The first real-world constraint application in this domain was a problem of *application placement* for the Italian Inter banking network [6], a problem very closely related to the warehouse location problem [45, 32].
– In many networks, the task of *path placement* is to define the route on which a demand will be sent through the network. This is a fundamental networking problem, for which many competing CP methods have been proposed. The models fall into three main sub-classes, link-based models [33, 34, 35, 27, 11, 12, 22], path-based models [5, 18, 25, 26] and node-based models [37].
– One possible extension is the use of *multiple paths* for demands, where the secondary path is only active when the primary connection has failed [50].
– Another possible extension is to add a time dimension, where traffic demands have given start and end times, and demands compete for network bandwidth if they overlap in time. This application is called *Bandwidth on Demand* [23, 28, 43, 7, 37].
– In the previous problems, the network structure and capacity was fixed. The problem of *Network Design* deals with defining connectivity and finding the right link capacity to satisfy a projected set of demands [24, 5, 9, 10, 41].

- IP (Internet Protocol) networks usually do not use explicit routes for traffic demands. Instead, packets are routed based on a distributed shortest path algorithm. *Metric optimization* deals with choosing metric weights to influence the routing in the network and to optimize the network utilization [1, 2, 16].
- Secondary paths and routing algorithms provide some methods to maintain network communications in case of element failure. The idea of *Bandwidth Protection* offers an alternative, purely local mechanism for improving network resilience [49, 48].

The spread of wireless networks has led to a whole new class of problems, two examples of the use of CP are shown in [19, 38].

2 Flow Analysis and Resilience Analysis

Most of the problems described above assume that there is a well-defined set of demands, the *Traffic Matrix*. We know who wants to use the network for connections between specific points and how much bandwidth they require. For IP based networks this assumption is, surprisingly, not valid. In an operational, routed network there is no (simple) way of collecting data about end-to-end traffic flows, we don't know who is talking to whom and how much bandwidth the customers use. The only information we can collect is the overall traffic on each link e of the network $\texttt{traf}(e)$ and the external traffic entering $\texttt{ext}^{in}(i)$ and leaving $\texttt{ext}^{out}(j)$ at nodes i and j of the network. We can try to reconstruct a traffic matrix from these measurements, this is an active research area called *traffic flow analysis*.

2.1 Related Work

Most of the related work is concerned with the identification of the traffic matrix for all PE (provider edge router) to PE or PoP (Point of presence) to PoP flows. [21] discuss use cases for this flow analysis and compare different means of collecting this data. We will concentrate on methods which deduce the traffic matrix indirectly, without collecting flow data throughout the network. There are two main directions this work is taking, the *tomography method* and the *gravity approach*.

The *tomography approach*, pioneered by Vardi [47], is based on a model where the traffic matrix is deduced from the link traffic. As this problem is under-constrained, a series of observations are used, assuming that the measurements are independent and reflect the same traffic matrix. A stochastic algorithm is used to find the most likely traffic matrix which fits the available data. Slightly different models and assumptions are used in [46, 44, 3, 8], a survey is given in [4]. The work in [20] that uses a linear model to calculate a traffic matrix from the link data observations is the one most closely related to the model presented here. The use of lower and upper bounds for the flow analysis was suggested in [36]. An important requirement for the tomography approach is the need for a routing model, which understands how traffic is flowing through the network.

An alternative approach is the *gravity model* [31], which is based on the traffic data of the external links only. Based on the assumption that the flow between two customers is proportional to the product of their input and output traffic values, we find a traffic matrix which is consistent with all external measurements, but not necessarily with the link loads inside the network. The underlying assumption that customers talk to each other with equal likelihood can be justified for large ISPs (Internet service provider) with a consistent user base, but are harder to maintain for enterprise networks or ISPs offering mainly VPN (virtual private network) connections, where we already know that certain customers only talk to a subset of all customers. The gravity model originates in social sciences, where it is used for example to predict traffic flows in public transport systems. The data requirements for a gravity model are much more restricted than for the tomography method, it neither needs a routing model nor traffic data from the interior of the network. But for the same reason it is inherently less accurate than the tomography approach.

[54] propose a combination of the gravity based approach and the tomography approach, which they validate on parts of the ATT network. For a large tier-1 service provider the assumptions of the gravity model seem valid, if you distinguish between end-user connections and peering and up-link lines.

As described in [30], the traffic matrix can be defined at different levels of network abstractions. Typical variants are customer to customer flows, edge to edge flows, flows between core routers or aggregated flows, for example between PoPs. Different levels of abstraction are useful for different use cases, e.g. edge-to-edge flows for traffic engineering, PoP to PoP flows for capacity planning. In the context of resilience analysis, we will use edge to edge flows.

[17] deals with the problem of directly collecting flow data from the network and propose two methods which concentrate on large flows only. It shows that the current implementation of netflow has significant scaling problems in a large network. [14,15] deal with the problem of sampling of traffic flows and the resulting inaccuracies.

2.2 Flow Analysis Model

A model for the traffic flow analysis is shown below. We describe the network as a directed graph $G = (\mathbf{N}, \mathbf{E})$ with nodes $\mathbf{N}$ and directed edges $\mathbf{E}$. We use non-negative flow variables F_{ij} to denote the traffic flow from node i to node j in the network. The $[0, 1]$ constants r^e_{ij} define the routing in the network, they indicate what fraction of the flow between nodes i and j is routed over edge e.

$$\forall i, j \in \mathbf{N} : \quad \min_{\{F_{ij}\}} / \max_{\{F_{ij}\}} \quad F_{ij} \tag{1}$$

st.

$$\forall e \in \mathbf{E}: \quad \sum_{i,j \in \mathbf{N}} r_{ij}^{e} F_{ij} = \texttt{traf}(e) \tag{2}$$

$$\forall i \in \mathbf{N}: \quad \sum_{j \in \mathbf{N}} F_{ij} = \texttt{ext}^{in}(i) \tag{3}$$

$$\forall j \in \mathbf{N}: \quad \sum_{i \in \mathbf{N}} F_{ij} = \texttt{ext}^{out}(j) \tag{4}$$

$$F_{ij} \geq 0$$

For every flow, we try to find a lower and an upper bound as the result of an optimization run with the objective (1). We know that the sum of all flows routed over an edge is equal to the observed traffic on the edge (2), and that the sum of all flows starting (3) or ending (4) in a node must be equal to the observed external traffic.

2.3 Data

For an evaluation of the model, we use 6 networks from the Rocketfuel project [29] and one other network topology (dexa) of a global enterprise network. The networks range from 51 to 315 routers, and also have quite different connectivity. Table 1 compares the major parameters of the network. Lines are bi-directional connections between routers, PoPs (Points of presence) indicate places where all routers for an area are co-located. Connections inside a PoP often are LAN (local area network) type, whereas connections between PoPs typically are WAN (wide area network) type and are more expensive. All networks are nearly real-life, the

Table 1. Test networks

Network	Routers	PoPs	Lines	Lines/Router
dexa	51	24	59	1.15
as1221	108	57	153	1.41
as1239	315	44	972	3.08
as1755	87	23	161	1.85
as3257	161	49	328	2.03
as3967	79	22	147	1.86
as6461	141	22	374	2.65

topology of the Rocketfuel networks is deduced from data collected remotely off the actual ISP networks, the dexa network is an operational network. For the dexa network, we also have actual link speeds and IGP (interior gateway protocol) metric values, while the metric values for the Rocketfuel networks are derived from traceroute information and no link speed is available (we assume the same speed for all link for simplicity). While we can clearly distinguish P (core) and PE (edge) routers in the dexa network, we can only do so heuristically in the other networks.

For the dexa network, we use the actual customer VPN structure, for all other networks we generate VPNs of different sizes randomly. We then generate for all networks random traffic flows between the VPN end points, and calculate from these simulated flows s_{ij} the expected traffic load at each interface. These traffic loads are consistent with each other and are generated using the same routing model that is used in the analysis part.

2.4 Traffic Flow Analysis Results

The basic problem with the model above is that it is very under-constrained. We have $|\mathbf{N}|^2$ flow variables F_{ij}, but only $|\mathbf{E}| + 2|\mathbf{N}|$ constraints. Results from [39] shown below indicate that the values for the flows can vary in a very wide interval, with no clear preference for any particular value. It is therefore unclear how to use the results for answering further questions about the network, for example how the traffic will change in case of an element failure.

In table 2 we present the sum of all lower bounds as a percentage of the sum of the simulated flow values ($100 * \frac{\sum_{i,j} \min F_{ij}}{\sum_{i,j} s_{ij}}$), as well as the sum of all upper bounds as a percentage of the sum of the simulated flows ($100 * \frac{\sum_{i,j} \max F_{ij}}{\sum_{i,j} s_{ij}}$). A value of 100% would be the optimal result. We also show the number of objective functions and the total time (in seconds) to run the test.

Table 2. TFA results

Network	Low/Simul	High/Simul	Obj	Time
dexa	0	2310.65	1190	11
as1221	0.09	8398.64	11556	1318
as1239	n/a	n/a	n/a	n/a
as1755	0.15	6255.31	7482	699
as3257	0.04	12260.03	25760	12389
as3967	0.1	5387.10	6162	500
as6461	0.28	8688.39	19740	8676

We could not obtain a result for network AS1239, but we estimate, based on a partial result, that a complete analysis would take more than 5 days.

All results are obtained on Linux PCs running ECLiPSe 5.6, using CPLEX 6.5 as the linear solver.

The lower bound tells us how much of the traffic in the network we can associate with specific flows between routers, i.e. we know where the traffic originates and where it ends. The upper bound indicates how uncertain we are about the exact source and destination. If the number is very high, then many flows may be the cause of the traffic, and we lack the ability to differentiate between them. In this particular setting, the results are unimpressive. The lower bounds are very close to zero, and the upper bounds over-estimate the simulated flows by a factor from 23 to 122! This means that we can not pin any traffic on particular flows and most of the traffic may have been caused by lots of flows between quite different routers.

Aggregating the flows for a PoP to PoP analysis improves the results, but not significantly. Table 3 considers flows between pairs of PoPs, which is the sum of all flows between all routers in either PoP. We can note two points:

Table 3. PoP TFA results

Network	Low/Simul	High/Simul	Obj	Time
dexa	0	1068.37	557	5
as1221	0.24	2964.93	3205	424
as1239	0.63	1401.72	1931	101359
as1755	0.66	1263.28	526	103
as3257	0.30	2028.73	2378	2052
as3967	0.1	1209.37	483	90
as6461	1.47	951.41	481	768

– The results are significantly better than for the router to router flow analysis, but not nearly good enough to identify the flows. The lower bounds are still very nearly zero, and the upper bounds overestimate the flows in total between 10 and 30 times.
– The run time is much reduced, since there are far fewer objectives to calculate, but per objective the runtime did slightly increase.

2.5 Resilience Analysis Model

The idea behind *resilience analysis* is to avoid the generation of the intermediate traffic matrix, and to pose questions about the network behavior directly in the initial model. For example, we may be interested in understanding the traffic in the network under an element failure and resulting re-routing. The routing in the normal network operation is denoted with r_{ij}^e, the routing after the element failure is given by $\overline{r_{ij}^e}$. The model for resilience analysis below uses the flow variables F_{ij} only internally, without trying to deduce particular values.

$$\forall e \in \mathbf{E} : \quad \min_{\{F_{ij}\}} / \max_{\{F_{ij}\}} \quad \sum_{i,j \in \mathbf{N}} \overline{r_{ij}^e} F_{ij} \tag{5}$$

st.

$$\forall e \in \mathbf{E} : \quad \sum_{i,j \in \mathbf{N}} r_{ij}^e F_{ij} = \texttt{traf}(e) \tag{6}$$

$$\forall i \in \mathbf{N}: \quad \sum_{j \in \mathbf{N}} F_{ij} = \mathtt{ext}^{in}(i) \tag{7}$$

$$\forall j \in \mathbf{N}: \quad \sum_{i \in \mathbf{N}} F_{ij} = \mathtt{ext}^{out}(j) \tag{8}$$

$$F_{ij} \geq 0$$

The objective function (5) now tries to find a value the predicted traffic on each edge in the network under the failure scenario, and finds bounds by running minimization and maximization optimization queries. The constraints (6, 7 and 8) are the same as for the traffic flow analysis.

2.6 Resilience Analysis Results

Results on the resilience analysis are a lot more encouraging as shown in table 4. The entry Low/Simul is calculated as

$$(100 * \frac{\sum_{e \in \mathbf{E}} \min \sum_{i,j \in \mathbf{N}} \overline{r_{ij}^{e}} * F_{ij}}{\sum_{e \in \mathbf{E}} \sum_{i,j \in \mathbf{N}} \overline{r_{ij}^{e}} * s_{ij}})$$

the value High/Simul is

$$(100 * \frac{\sum_{kl \in \mathbf{E}} \max \sum_{i,j \in \mathbf{N}} \overline{r_{ij}^{kl}} * F_{ij}}{\sum_{kl \in \mathbf{E}} \sum_{i,j \in \mathbf{N}} \overline{r_{ij}^{kl}} * s_{ij}})$$

We also report the number of objective functions (Obj), the total time (Time) and the number of failure cases (Cases) considered. We identify between 68 and 96% of the simulated traffic volume in the lower bound, and the sum of the upper bounds over-estimates the simulated traffic by a maximum of 9%.

Table 4. Resilience Analysis

Network	Low/Simul	High/Simul	Obj	Time	Cases
dexa	68.91	108.25	3503	57	59
as1221	85.75	102.60	14191	2869	153
as1239	92.53	102.64	4499	44205	10
as1755	92.82	105.39	8409	1815	161
as3257	93.69	103.15	31093	39934	328
as3967	91.60	108.79	9090	1635	141
as6461	96.51	103.44	24808	20840	374

Note that we did not run all failure cases on the largest network due to time limitations.

To check if the results are typical, we repeated the experiments with one hundred randomly generated data sets for the four smallest networks. In table 5 we show the average value and its standard deviation for both lower bounds and upper bounds. Results are quite consistent and confirm our initial values.

Table 5. Average results (100 runs) for resilience analysis

Network	Lower bound/Simul		Upper bound/Simul	
	Average	Stdev	Average	Stdev
dexa	91.50	0.14	108.28	0.16
as1755	88.65	0.11	106.08	0.056
as3967	94.08	0.073	106.88	0.091
as1221	87.34	0.10	102.05	0.025

2.7 Adding Information

One basic limit of the flow analysis is that we have too few constraints for too many variables. We now add a new data source to the problem, which will provide us with many additional constraints. In MPLS networks [13], we can not only use interface traffic counters, but also use counters on each LSP (label switched path) [42]. For each output interface, we get a counter for each LSP routed through the interface leading to a destination router. This counter will give us the sum of all flows to the destination which have been forwarded through the router, but it does not contain the flow that starts in the router. The MIB (management information base) also defines LSP counters on all input interfaces, but these counters are not meaningful on Cisco routers in current IOS versions.

We can define the LSP counters formally with the following definition.

Definition 1. *The constant v_e^j is the (consistent) LSP counter volume on the directed link e from node k to node l for all flows with destination node j which are forwarded on the link. The counter does not include the flow that originates in node k.*

This naturally leads to the next constraint, which links a sum of flow variables to the counter value.

Constraint 1. *The constraint states that the sum of all flows through a link towards a destination is equal to the LSP counter for that destination on the link.*

$$\forall \quad j \in \boldsymbol{N}, e = (kl) \in \boldsymbol{E}: \quad \sum_{i \in \boldsymbol{N}, i \neq k} r_{ij}^e * F_{ij} = v_e^j$$

The exact number of non-trivial constraints of this form depends on the topology, but usually is $O(n^2)$. This means that the problem is much more tightly constrained, and the results of the flow analysis should improve dramatically.

Table 6 shows the results of the experiments for traffic flow analysis. The lower bounds now range from 10 to 30 % of the simulated flows, the upper bounds range between 2.5 and 10 times the simulated flows. This is much better than before (see table 2), but still quite disappointing. The run times decreased a lot as well, adding more constraints helped the problem solving.

We also repeated the experiments for the PoP flow analysis. Table 7 shows the result. For some networks (dexa and AS1755) the results are nearly usable, but in general they are still not good enough to identify the flow values.

Table 6. TFA results with LSP counters

Network	Low/Simul	High/Simul	Obj	Time
dexa	30.35	249.71	1190	7
as1221	9.94	685.37	11556	885
as1239	10.74	1151.03	98910	72461
as1755	25.29	269.30	7482	397
as3257	23.77	425.67	25760	5121
as3967	24.47	300.17	6162	275
as6461	19.43	477.44	19740	2683

Table 7. PoP TFA results with LSP counters

Network	Low/Simul	High/Simul	Obj	Time
dexa	60.62	145.85	557	3
as1221	28.49	499.16	3205	271
as1239	33.36	211.84	1931	2569
as1755	50.33	169.37	526	46
as3257	36.82	249.16	2378	640
as3967	40.72	182.97	483	36
as6461	34.05	210.93	481	136

If we add LSP counter constraints to the model, then the results for resilience analysis are even more impressive. The lower bounds in table 8 reach 97 to 99.44 % of the simulated values, and the sum of the upper bounds is 101.33 % of the simulated values in the worst case. Also note that again the execution times decrease when we add the LSP counter constraints to the model, by more than a factor of 10 for the largest network.

Table 8. Resilience Analysis with LSP counters

Network	Low/Simul	High/Simul	Obj	Time	Cases
dexa	97.76	101.33	3503	36	59
as1221	98.15	100.69	14191	1840	153
as1239	99.37	100.38	4499	3974	10
as1755	99.28	100.66	8409	964	161
as3257	99.41	100.44	31093	13381	328
as3967	98.88	101.00	9090	819	147
as6461	99.44	100.52	24808	8006	374

Again we check the results for the 4 smallest networks by generating one hundred data sets and recording average percentage and standard deviation for the sums of the lower bounds and the sums of the upper bounds compared to the simulated link traffic values in all single-node failure cases as shown in table 9.

Table 9. Average results (100 runs) for resilience analysis with LSP counters

Network	Lower bound/Simul		Upper bound/Simul	
	Average	Stdev	Average	Stdev
dexa	99.60	0.029	100.33	0.025
as1755	99.31	0.016	100.63	0.015
as3967	99.41	0.014	100.61	0.014
as1221	98.10	0.025	100.57	0.010

The results indicate that the resilience analysis with LSP counters is able to predict the link load in the network very accurately. In a similar way, other information can be added, for example partial netflow results for selected routers, bounds obtained from specific applications or from the VPN structure. In each case, the added information adds constraints to the problem, tightening the bounds obtained from the model. Since we obtain both lower and upper bounds, we can also easily decide how much additional data is required. Once the bounds are close enough, we can stop adding more information, thereby reducing the data collection overhead.

2.8 Discussion

In the presentation above, we have oversimplified the use of the actual traffic measurements. The models as shown only work if a consistent snapshot of all values can be collected. In practice, this poses significant problems. If the data are not collected for exactly the same time periods, then inconsistencies may occur. There are further problems caused by queues in the routers and bugs in implementing data collection facilities in devices of multiple vendors. The data collection process itself uses unreliable communications (UDP) so that some measurements may be lost due to dropped packets. One approach to overcoming these issues is the use of a separate error correction model, which tries to correct values before feeding them into the models above. Another, shown in [52,51,53] deals with the problem by integrating incomplete and inconsistent data into the constraint solving process.

References

1. F. Ajili, R. Rodosek, and A. Eremin. A branch-price-and-propagate approach for optimising IGP weight setting subject to unique shortest paths. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (ACM SAC '05)*, Santa Fe, New Mexico, March 2005.
2. F. Ajili, R. Rodosek, and A. Eremin. A scalable tabu search algorithm for optimising IGP routing. In *2nd International Network Optimization Conference (INOC '05)*, pages 348–354, March 2005.
3. J. Cao, D. Davis, S. Vander Weil, and B. Yu. Time-varying network tomography. *Journal of the American Statistical Association*, 2000.

4. R. Castro, M. Coates, G. Liang, R. Nowak, and B. Yu. Network tomography: Recent developments, 2003.
5. A. Chabrier. Heuristic branch-and-price-and-cut to solve a network design problem. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CP-AI-OR 03*, Montreal, Canada, May 2003.
6. C. Chiopris and M. Fabris. Optimal management of a large computer network with CHIP. In *2nd Conf Practical Applications of Prolog*, London, UK, April 1994.
7. Y. Chu and Q. Xia. Bandwidth-on-demand problem and temporal decomposition. In *2nd International Network Optimization Conference (INOC '05)*, pages 542–550, Lisbon, Portugal, March 2005.
8. M. Coates, A. Hero, R. Nowak, and B. Yu. Internet tomography. *IEEE Signal Processing Magazine*, May 2002.
9. W. Cronholm and F. Ajili. Strong cost-based filtering for Lagrange decomposition applied to network design. In M. Wallace, editor, *10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, pages 726–730, Toronto, Canada, 2004. Springer-Verlag.
10. W. Cronholm and F. Ajili. Hybrid branch-and-price for multicast network design. In *2nd International Network Optimization Conference (INOC '05)*, pages 796–802, Lisbon, Portugal, March 2005.
11. W. Cronholm, W. Ouaja, and F. Ajili. Strengthening optimality reasoning for a network routing application. In *4th International Workshop on Cooperative Solvers in Constraint Programming (CoSolv '04)*, Toronto, Canada, September 2004.
12. W. Cronholm, W. Ouaja, and F. Ajili. Strong reduced cost fixing in network routing. In *2nd International Network Optimization Conference (INOC '05)*, pages 688–694, Lisbon, Portugal, March 2005.
13. B. Davie and Y. Rekhter. *MPLS: Technology and Applications*. Morgan Kauffmann Publishers, 2000.
14. N. Duffield, C. Lund, and M. Thorup. Charging from sampled network usage. In *SIGCOMM Internet Measurement workshop*, November 2001.
15. N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking*, pages 226–237, June 2001.
16. A. Eremin, F. Ajili, and R. Rodosek. A set-based approach to the optimal IGP weight setting problem. In *2nd International Network Optimization Conference (INOC '05)*, pages 386–392, Lisbon, Portugal, March 2005.
17. C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM2002*, September 2002.
18. C. Frei and B. Faltings. Resource allocation in networks using abstraction and constraint satisfaction techniques. In *Principles and Practice of Constraint Programing - CP 1999*, Alexandria, Virginia, October 1999.
19. T. Fruehwirth and P. Brisset. Optimal placement of base stations in wireless indoor telecommunication. In *Principles and Practice of Constraint Programing - CP 1998*, Pisa, Italy, October 1998.
20. O. Goldschmidt. ISP backbone traffic inference methods to support traffic engineering. In *Internet Statistics and Metrics Analysis (ISMA) Workshop*, San Diego, CA, December 2000.
21. M. Grossglauser and J. Rexford. Passive traffic measurement for IP operations. Technical report, ATT, 2001.
22. O. Kamarainen and H. El Sakkout. Local probing applied to network routing. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CP-AI-OR 04*, Nice, France, April 2004.

23. M. Lauvergne, P. David, and P. Bauzimault. Connections reservation with rerouting for ATM networks: A hybrid approach with constraints. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programing - CP 2002*, Cornell University, Ithaca, N.Y., September 2002.
24. C. Le Pape, L. Perron, J. Regin, and P. Shaw. Robust and parallel solving of a network design problem. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programing - CP 2002*, Cornell University, Ithaca, N.Y., September 2002.
25. J. Lever. A local search/constraint propagation hybrid for a network routing problem. In *The 17th International FLAIRS Conference (FLAIRS-2004)*, Miami Beach, Florida, May 2004.
26. J. Lever. A local search/constraint propagation hybrid for a network routing problem. *International Journal on Artificial Intelligence Tools*, 14(1-2):43–60, 2005.
27. V. Liatsos, S. Novello, and H. El Sakkout. A probe backtrack search algorithm for network routing. In *Proceedings of the Third International Workshop on Cooperative Solvers in Constraint Programming, CoSolv'03*, Kinsale, Ireland, September 2003.
28. S. Loudni, P. David, and P. Boizumault. On-line resource allocation for ATM networks with rerouting. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CP-AI-OR 03*, Montreal, Canada, May 2003.
29. R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *IMW2002*, 2002.
30. A. Medina, C. Fraleigh, N. Taft, S. Bhattacharyya, and C. Diot. A taxonomy of IP traffic matrices. In *Workshop on Scalability and Traffic Control in IP Networks at the SPIE ITCOM+OPTICOMM 2002 Conference*, Boston, MA, June 2002.
31. A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrices estimation: Existing techniques and new directions. In *ACM SIGCOMM2002*, Pittsburgh, PA, August 2002.
32. L. Michel and P. Van Hentenryck. A simple tabu search for warehouse location. *European Journal on Operations Research*, pages 576–591, 2004.
33. W. Ouaja and B. Richards. A hybrid solver for optimal routing of bandwidth-guaranteed traffic. In *INOC2003*, pages 441–447, 2003.
34. W. Ouaja and B. Richards. A hybrid multicommodity routing algorithm for traffic engineering. *Networks*, 43(3):125–140, 2004.
35. W. Ouaja and E. B. Richards. Hybrid Lagrangian relaxation for bandwidth-constrained routing: Knapsack decomposition. In *20th Annual ACM Symposium on Applied Computing (ACM SAC '05)*, pages 383–387, Santa Fe, New Mexico, March 2005.
36. R. Rodosek and B. Richards. RiskWise constraint model. Internal Note, 2000.
37. L. Ros, T. Creemers, E. Tourouta, and J. Riera. A global constraint model for integrated routeing and scheduling on a transmission network. In *7th International Conference on Information Networks, Systems and Technologies*, Minsk, October 2001.
38. Y. Shang, M. Fromherz, Y. Zhang, and L. S. Crawford. Constraint-based routing for ad-hoc networks. In *IEEE Int. Conf. on Information Technology: Research and Education (ITRE 2003)*, pages 306–310, Newark, NJ, USA, August 2003.
39. H. Simonis. Resilience analysis in MPLS networks. Technical report, Parc Technologies Ltd, 2003.
40. H. Simonis. Constraint applications in networks. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 25. Elsevier, 2006.

41. B. M. Smith. Symmetry and search in a network design problem. In Roman Barták and Michela Milano, editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2005.
42. C. Srinivasan, A. Viswanathan, and T. D. Nadeau. Multiprotocol label switching (MPLS) label switching router (LSR) management information base. Technical report, IETF, October 2003. draft-ietf-mpls-lsr-mib-13.txt.
43. J. Symes. Bandwidth-on-demand services using MPLS-TE. In *MPLS World Congress 2004*, Paris, France, February 2004.
44. Y. Tsang, M. Coates, and R. Nowak. Passive network tomography using EM algorithms. In *IEEE Conf Acoust. Speech and Signal Proc*, May 2001.
45. P. Van Hentenryck and J.P. Carillon. Generality versus specificity: An experience with AI and OR techniques. In *AAAI*, pages 660–664, 1988.
46. R. Vanderbei and J. Iannone. An EM approach to OD matrix estimation. Technical Report SOR 94-04, Princeton University, 1994.
47. Y. Vardi. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association*, pages 365–377, 1996.
48. Q. Xia. Traffic diversion problem: Reformulation and new solutions. In *2nd International Network Optimization Conference (INOC '05)*, pages 235–241, Lisbon, Portugal, March 2005.
49. Q. Xia, A. Eremin, and M. Wallace. Problem decomposition for traffic diversions. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CP-AI-OR 2004*, pages 348–363, Nice, France, April 2004.
50. Q. Xia and H. Simonis. Primary/secondary path generation problem: Reformulation, solutions and comparisons. In *4th International Conference on Networking*, Reunion Island, France, 2005. Springer Verlag.
51. N. Yorke-Smith. *Reliable Constraint Reasoning with Uncertain Data*. PhD thesis, IC-Parc, Imperial College London, University of London, June 2004.
52. N. Yorke-Smith and C. Gervet. On constraint problems with incomplete or erroneuos data. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programing - CP 2002*, Cornell University, Ithaca, N.Y., September 2002.
53. N. Yorke-Smith and C. Gervet. Tight and tractable reformulations for uncertain CSPs. In *CP '04 Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, Toronto, Canada, September 2004.
54. Y. Zhang, M. Roughan, N.G. Duffield, and A. Greenberg. Fast accurate computation of large-scale IP traffic matrices from link loads. In *ACM Sigmetrics*, 2003.

Infinite Qualitative Simulations by Means of Constraint Programming

Krzysztof R. Apt[1,2] and Sebastian Brand[3]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] University of Amsterdam, The Netherlands
[3] NICTA, Victoria Research Lab, Melbourne, Australia

Abstract. We introduce a constraint-based framework for studying infinite qualitative simulations concerned with contingencies such as time, space, shape, size, abstracted into a finite set of qualitative relations. To define the simulations we combine constraints that formalize the background knowledge concerned with qualitative reasoning with appropriate inter-state constraints that are formulated using linear temporal logic.

We implemented this approach in a constraint programming system (ECL^{i}PS^e) by drawing on the ideas from bounded model checking. The implementation became realistic only after several rounds of optimizations and experimentation with various heuristics.

The resulting system allows us to test and modify the problem specifications in a straightforward way and to combine various knowledge aspects. To demonstrate the expressiveness and simplicity of this approach we discuss in detail two examples: a navigation problem and a simulation of juggling.

1 Introduction

1.1 Background

Qualitative reasoning was introduced in AI to abstract from numeric quantities, such as the precise time of an event, or the location or trajectory of an object in space, and to reason instead on the level of appropriate abstractions. Two different forms of qualitative reasoning were studied. The first one is concerned with reasoning about continuous change in physical systems, monitoring streams of observations and simulating behaviours, to name a few applications. The main techniques used are qualitative differential equations, constraint propagation and discrete state graphs. For a thorough introduction see [15].

The second form of qualitative reasoning focuses on the study of contingencies such as time, space, shape, size, directions, through an abstraction of the quantitative information into a finite set of qualitative relations. One then relies on complete knowledge about the interrelationship between these qualitative relations. This approach is exemplified by temporal reasoning due to [1], spatial reasoning introduced in [10] and [19], reasoning about cardinal directions (such as North, Northwest); see, e.g., [17], etc.

In this paper we study the second form of qualitative reasoning. Our aim is to show how infinite qualitative simulations can be naturally formalized by means of temporal logic and constraint satisfaction problems. Our approach allows us to use generic constraint programming systems rather than specialized qualitative reasoning systems. By a ***qualitative simulation*** we mean a reasoning about possible evolutions in time of models capturing qualitative information. One assumes that time is discrete and that only changes adhering to some desired format occur at each stage. Qualitative simulation in the first framework is discussed in [16], while qualitative spatial simulation is considered in [9].

1.2 Approach

In the traditional constraint-based approach to qualitative reasoning the qualitative relations (for example overlap) are represented as constraints over variables with infinite domains (for example closed subsets of $\mathcal{R}^2$) and path-consistency is used as the constraint propagation; see, e.g., [11].

In our approach we represent qualitative relations *as variables*. This allows us to trade path-consistency for hyper-arc consistency which is directly available in most constraint programming systems, and to combine in a simple way various theories constituting the *background knowledge*. In turn, the *domain specific knowledge* about simulations is formulated using the linear temporal logic. These temporal formulas are subsequently translated into constraints.

Standard techniques of constraint programming combined with techniques from bounded model checking can then be used to generate simulations. To support this claim, we implemented this approach in the constraint programming system ECL^{i}PS^e. However, this approach became realistic only after fine-tuning of the translation of temporal formulas to constraints and a judicious choice of branching strategy and constraint propagation. To show its usefulness we discuss in detail two case studies. In each of them the solutions were successfully found by our implementation, though for different problems different heuristic had to be used.

The program is easy to use and to interact with. In fact, in some of the case studies we found by analyzing the generated solutions that the specifications were incomplete. In each case, thanks to the fact that the domain specific knowledge is formulated using temporal logic formulas, we could add the missing specifications in a straightforward way.

1.3 Structure of the Paper

In Section 2 we discuss examples of qualitative reasoning and in Section 3 explain our formalization of the qualitative reasoning by means of constraints. Next, in Section 4 we deal with qualitative simulations by introducing inter-state constraints which connect different stages of simulation and determine which scenarios are allowed. These constraints are defined using linear temporal logic. Their semantics is defined employing the concept of a cyclic path borrowed from the bounded model checking approach (see [5]) for testing validity of temporal formulas.

In Section 5 we explain how the inter-state constraints are translated to constraints of the underlying background knowledge. Next, in Section 6 we discuss technical issues pertaining to our implementation that generates infinite qualitative simulations. In the subsequent two sections we report on our case studies. Finally, in Section 9 we discuss the related work.

2 Qualitative Reasoning: Setup and Examples

As already said, in qualitative reasoning, one abstracts from the numeric quantities and reasons instead on the level of their abstractions. These abstractions are provided in the form of a finite ***set of qualitative relations***, which should be contrasted with the infinite set of possibilities available at the numeric level. After determining the 'background knowledge' about these qualitative relations we can derive conclusions on an abstract level that would be difficult to achieve on the numeric level. The following three examples illustrate the matters.

Example 1 (Region Connection Calculus). The qualitative spatial reasoning with topology introduced in [19] and [10] is concerned with the following set of qualitative relations:

$$\mathsf{RCC8} := \{\mathsf{disjoint}, \mathsf{meet}, \mathsf{overlap}, \mathsf{equal}, \mathsf{covers}, \mathsf{contains}, \mathsf{covered\text{-}by}, \mathsf{inside}\}.$$

The objects under consideration are here spatial regions, and each region pair is in precisely one RCC8 relation; see Fig. 1.

The background knowledge in this case is the set of possible relation triples pertaining to triples of regions. For example, the relation triple $\langle \mathsf{meet}, \mathsf{meet}, \mathsf{meet} \rangle$ is possible since there exist three regions pairwise touching each other. In contrast, the triple $\langle \mathsf{inside}, \mathsf{inside}, \mathsf{disjoint} \rangle$ is impossible since for any three regions A, B, C, if A is inside B and B is inside C, then A cannot be disjoint with C. The set of possible triples is called the ***composition table***; it is presented in the above two papers. In total, the table lists 193 relation triples.

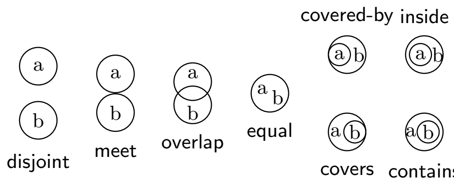


Fig. 1. The eight RCC8 relations

Example 2 (Cardinal Directions). Qualitative reasoning dealing with relative directional information about point objects can be formalized using the set of cardinal directions

$$\mathsf{Dir} := \{\mathsf{N}, \mathsf{NE}, \mathsf{E}, \mathsf{SE}, \mathsf{S}, \mathsf{SW}, \mathsf{W}, \mathsf{NW}, \mathsf{EQ}\},$$

that consists of the wind rose directions together with the identity relation denoted by EQ; see [12]. The composition table for this form of qualitative reasoning is provided in [17].

Example 3 (Relative Size). Qualitative reasoning about relative size of objects is captured by the relations in the set

$$\mathsf{Size} := \{<, =, >\}.$$

The corresponding composition table is given in [13].

Other examples of qualitative reasoning deal with shape, directional information about regions or cyclic ordering of orientations. In some of them the qualitative relations are non-binary and the background knowledge is more complex than the composition table. To simplify the exposition we assume in the following binary qualitative relations.

3 Formalization of Qualitative Reasoning

In what follows we follow the standard terminology of constraint programming. So by a ***constraint*** on a sequence $x_1, \ldots, x_m$ of variables with respective domains $dom(x_1), \ldots, dom(x_m)$ we mean a subset of $dom(x_1) \times \cdots \times dom(x_m)$. A ***constraint satisfaction problem (CSP)*** consists of a finite sequence of variables X with respective domains and a finite set of constraints, each on a subsequence of X. A ***solution*** to a CSP is an assignment of values to its variables from their domains that satisfies all constraints.

We study here CSPs with finite domains and solve them using a top-down search interleaved with constraint propagation. In our implementation we use a heuristics-controlled ***domain partitioning*** as the branching strategy and ***hyper-arc consistency*** of [18] as the constraint propagation.

We formalize the qualitative reasoning within the CSP framework as follows. We assume a finite set of objects $\mathcal{O}$, a finite set of binary qualitative relations $\mathcal{Q}$ and a ternary relation CT representing the composition table. Each qualitative relation between objects is modelled as a constraint variable the domain of which is a subset of $\mathcal{Q}$. We stipulate such a ***relation variable*** for each ordered pair of objects and organize these variables in an array Rel which we call a ***qualitative array***.

For each triple a, b, c of elements of $\mathcal{O}$ we have then a ternary constraint comp on the corresponding variables:

$$\mathsf{comp}(Rel[a,b],\ Rel[b,c],\ Rel[a,c]) := \\ CT \cap (dom(Rel[a,b]) \times dom(Rel[b,c]) \times dom(Rel[a,c])).$$

To assume internal integrity of this approach we also adopt for each ordered pair a, b of elements of $\mathcal{O}$, the binary constraint $\mathsf{conv}(Rel[a,b],\ Rel[b,a])$ that represents the converse relation table, and postulate that $Rel[a,a] = \mathsf{equal}$ for all $a \in \mathcal{O}$.

We call these constraints ***integrity constraints***.

4 Specifying Simulations Using Temporal Logic

In our framework we assume a ***conceptual neighbourhood*** between the qualitative relations. This is a binary relation $\mathsf{neighbour}$ between the elements of the relation set $\mathcal{Q}$ describing which *atomic* changes in the qualitative relations are admissible. So only 'smooth' transitions are allowed. For example, in the case of the Region Connection Calculus from Example 1, the relation between two regions can change from $\mathsf{disjoint}$ to $\mathsf{overlap}$ only indirectly via meet. The neighbourhood relation for $\mathsf{RCC8}$ has 22 elements such as $\langle\mathsf{disjoint}, \mathsf{meet}\rangle$, $\langle\mathsf{meet}, \mathsf{meet}\rangle$, $\langle\mathsf{meet}, \mathsf{overlap}\rangle$ and their converses and is shown in Fig. 2.

We assume here that objects can change size during the simulation. If we wish to disallow this possibility, then the pairs $\langle\mathsf{equal}, \mathsf{covered\text{-}by}\rangle$, $\langle\mathsf{equal}, \mathsf{covers}\rangle$, $\langle\mathsf{equal}, \mathsf{inside}\rangle$, $\langle\mathsf{equal}, \mathsf{contains}\rangle$ and their converses should be excluded from the conceptual neighbourhood relation.

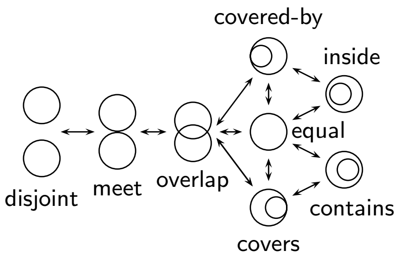


Fig. 2. The $\mathsf{RCC8}$ neighbourhood relation

In what follows we represent each stage t of a simulation by a CSP $\mathcal{P}_t$ uniquely determined by a qualitative array Q_t and its integrity constraints. Here t is a variable ranging over the set of natural numbers that represents discrete time. Instead of $Q_t[a, b]$ we also write $Q[a, b, t]$, as in fact we deal with a *ternary* array.

The stages are linked by ***inter-state constraints*** that determine which scenarios are allowed. The inter-state constraints always include constraints stipulating that the atomic changes respect the conceptual neighbourhood relation. Other inter-state constraints are problem dependent.

A qualitative simulation corresponds then to a CSP consisting of *stages* all of which satisfy the integrity constraints and the problem dependent constraints, and such that the inter-state constraints are satisfied. To describe the inter-state constraints we use ***atomic formulas*** of the form

$$Q[a, b] \in \mathcal{R},\ Q[a, b] \notin \mathcal{R},\ Q[a, b] = q,\ Q[a, b] \neq q,$$

where $\mathcal{R} \subseteq \mathcal{Q}$ and $q \in \mathcal{Q}$. As the latter three forms reduce to the first one, we deal with the first form only.

We employ a propositional linear temporal logic with four ***temporal operators***, $\Diamond$ (eventually), $\bigcirc$ (next time), $\Box$ (from now on) and U (until), and with the usual connectives. We use bounded quantification as abbreviations, e.g., $\phi(o_1) \vee \ldots \vee \phi(o_k)$ abbreviates to $\exists A \in \{o_1, \ldots, o_k\}. \phi(A)$.

Given a finite set of temporal formulas formalizing the inter-state constraints we wish then to exhibit a simulation in the form of an infinite sequence of 'atomic' transitions which satisfies these formulas and respects the integrity constraints. In the Section 5 we explain how each temporal formula is translated into a sequence of constraints.

$$\begin{tikzcd} Q_1 \arrow[r] & Q_2 \arrow[r] & \cdots \arrow[r] & Q_\ell \arrow[r] & \cdots \arrow[r] & Q_k \arrow[ll, bend right] \end{tikzcd}$$

Fig. 3. A $(k-\ell)$-loop

Paths and loops. We now proceed by explaining the meaning of a temporal formula ϕ with respect to an arbitrary infinite sequence of qualitative arrays,

$$\pi := Q_1, Q_2, \ldots,$$

that we call a ***path***. Our goal is to implement this semantics, so we proceed in two stages:

– First we provide a definition with respect to an arbitrary path.
– Then we limit our attention to specific types of paths, which are unfoldings of a loop.

In effect, we use here the approach employed in *bounded model checking*; see [5]. Additionally, to implement this approach in a simple way, we use a recursive definition of meaning of the temporal operators instead of the inductive one.

We write $\models_\pi \phi$ to express that ϕ holds along the path π. We say then that $\boldsymbol{\pi}$ ***satisfies*** $\boldsymbol{\phi}$. Given $\pi := Q_1, Q_2, \ldots$ we denote by π_i the subpath $Q_i, Q_{i+1}, \ldots$. Hence $\pi_1 = \pi$. The semantics is defined in the standard way, with the exception that the atomic formulas refer to qualitative arrays. The semantics of connectives is defined independently of the temporal aspect of the formula. For other formulas we proceed by recursion as follows:

$$\begin{array}{lll}
\models_{\pi_i} Q[a,b] \in \mathcal{R} & \text{if} & Q[a,b,i] \in \mathcal{R}; \\
\models_{\pi_i} \bigcirc\phi & \text{if} & \models_{\pi_{i+1}} \phi; \\
\models_{\pi} \Box\phi & \text{if} & \models_\pi \phi \text{ and } \models_\pi \bigcirc\Box\phi; \\
\models_{\pi} \Diamond\phi & \text{if} & \models_\pi \phi \text{ or } \models_\pi \bigcirc\Diamond\phi; \\
\models_{\pi} \chi \,\mathsf{U}\, \phi & \text{if} & \models_\pi \phi \text{ or } \models_\pi \chi \wedge \bigcirc(\chi \,\mathsf{U}\, \phi).
\end{array}$$

Next, we limit our attention to paths that are loops. Following [5] we call a path $\pi := Q_1, Q_2, \ldots$ a $\boldsymbol{(k-\ell)}$***-loop*** if

$$\pi = u \cdot v^* \quad \text{with} \quad u := Q_1, \ldots, Q_{\ell-1} \quad \text{and} \quad v := Q_\ell, \ldots, Q_k;$$

see Fig. 3. By a general result, see [5], for every temporal formula ϕ if a path exists that satisfies it, then a loop path exists that satisfies ϕ. This is exploited by our algorithm. Given a finite set of temporal formulas Φ it tries to find a path $\pi := Q_1, Q_2, \ldots$ consisting of qualitative arrays that satisfies all formulas in Φ, by repeatedly trying to construct an infinite $(k-\ell)$-loop. Each such $(k-\ell)$-loop can be finitely represented using k qualitative arrays. The algorithm is discussed in Section 6.

5 Temporal Formulas as Constraints

A temporal formula restricts the sequence of qualitative arrays at consecutive stages (time instances). We now show how to translate these formulas to constraints in a generic target constraint language. The translation is based on unravelling the temporal operators into primitive Boolean constraints and primitive constraints accessing the qualitative arrays. Furthermore, we discuss a variation of this translation that retains more structure of the formula, using non-Boolean array constraints.

We assume that the target constraint language has primitive Boolean constraints and reified versions of simple comparison and arithmetic constraints. (Recall that a reified constraint generalizes its base constraint by associating with it a Boolean variable reflecting its truth.)

Paths with and without loops. Both finite and infinite paths can be accommodated within one constraint model. To this end, we view a finite sequence of qualitative arrays together with their integrity constraints as a single CSP. The sequence $Q_1, \ldots, Q_k$ can represent both

an infinite path $\pi = (Q_1, \ldots, Q_{\ell-1}) \cdot (Q_\ell, \ldots, Q_k)^*$, for some $\ell \geqslant 1$ and $k \geqslant \ell$,
or a finite path $\pi = Q_1, \ldots, Q_k$.

To distinguish between these cases, we interpret ℓ as a constraint variable. We define $\ell = k{+}1$ to mean that there is no loop, so we have $dom(\ell) = \{1, \ldots, k{+}1\}$. A new placeholder array Q_{k+1} is appended to the sequence of qualitative arrays, *without* integrity constraints except the neighbourhood constraints connecting it to Q_k. Finally, possible looping is realized by conditional equality constraints

$$(\ell = j) \ \rightarrow \ (Q_j = Q_{k+1})$$

for all $j \in \{1, \ldots, k\}$. Here $Q_p = Q_q$ is an equality between qualitative arrays, i.e., the conjunction of equalities between the corresponding array elements.

Translation into constraints. We denote by $cons(\phi, i) \equiv b$ the sequence of constraints representing the fact that formula ϕ has the truth value b on the path π_i. The translation of a formula ϕ on $Q_1, \ldots, Q_k$ is initiated with $cons(\phi, 1) \equiv 1$.

We define the constraint translation inductively as follows.

Atomic formulas:

$cons(\mathsf{true}, i) \equiv b$	translates to	$b = 1$;
$cons(Q[a_1, a_2] \in R, i) \equiv b$	translates to	$Q[a_1, a_2, i] = q, (q \in R) \equiv b$.

Connectives:

$cons(\neg\phi, i) \equiv b$	translates to	$(\neg b') \equiv b, cons(\phi, i) \equiv b'$;

other connectives are translated analogously.

Formula $\bigcirc\phi$: The next-time operator takes potential loops into account.

```
cons(○φ, i) ≡ b    translates to
                      if $i < k$ then
                        $cons(\phi, i+1) \equiv b$;
                      if $i = k$ then
                        $\ell = k+1 \to b = 0$,
                        $\ell \leqslant k \quad \to b = \bigwedge_{j\in\{1,\dots,k\}} (\ell = j \to cons(\phi, j))$.
```

Formula $\Diamond\phi$: We translate $\Diamond\phi$ by unravelling its recursive definition $\phi \vee \bigcirc\Diamond\phi$. It suffices to do so a finite number n_{unravel} of steps beyond the current state, namely the number of steps to reach the loop, $\max(0, \ell - i)$, plus the length of the loop, $k - \ell$. A subsequent unravelling step is unneeded as it would reach an already visited state. We find

$$n_{\text{unravel}} = k - \min(\ell, i).$$

This equation is a simplification in that ℓ is assumed constant. For a variable ℓ, we 'pessimistically' replace ℓ here by the least value in its domain, $min(\ell)$.

Formulas $\Box\phi$ and $\phi \,\mathsf{U}\, \psi$: These formulas are processed analogously to $\Diamond\phi$.

The result of translating a formula is a set of primitive reified Boolean constraints and accesses to the qualitative arrays at certain times.

Translation using array constraints. Unravelling the temporal operators leads to a creation of several identical copies of subformulas. In the case of the $\Diamond$ temporal operator where the subformulas in essence are connected disjunctively, we can do better by translating differently. The idea is to push disjunctive information inside the variable domains. We use ***array constraints***, which treat array lookups such as $x = A[y_1, \dots, y_n]$ as a constraint on the variables $x, y_1, \dots, y_n$ and the (possibly variable) elements of the array A. Array constraints generalize the classic element constraint.

Since we introduce new constraint variables when translating $\Diamond\phi$ using array constraints, one needs to be careful when $\Diamond\phi$ occurs in the scope of a negation. Constraint variables are implicitly existentially quantified, therefore negation cannot be implemented by a simple inversion of truth values. We address this difficulty by first transforming a formula into a negation normal form, using the standard equivalences of propositional and temporal logic.

The constraint translations using array constraints (where different from above) follow. The crucial difference to the unravelling translation is that here i is a constraint variable.

Formula $\Diamond\phi$: A fresh variable j ranging over state indices is introduced, marking the state at which ϕ is examined. The first possible state is the current position or the loop start, whichever is earlier. Both ℓ and i are constraint variables, therefore their least possible values $min(\ell)$, $min(i)$, respectively, are considered.

$$\begin{array}{lll} cons(\Diamond\phi, i) \equiv b & \text{translates to} & \text{new } j \text{ with } dom(j) = \{1, \ldots, k\}, \\ & & \min(min(\ell), min(i)) \leqslant j, \\ & & cons(\phi, j) \equiv b. \end{array}$$

Formula $\bigcirc\phi$: This case is equivalent to the previous translation of $\bigcirc\phi$, but we now need to treat i as a variable. So both "if ... then" and $\rightarrow$ are now implemented by Boolean constraints.

6 Implementation

Given a qualitative simulation problem formalized by means of integrity constraints and inter-state constraints formulated as temporal formulas, our program generates a solution if one exists or reports a failure. During its execution a sequence of CSPs is repeatedly constructed, starting with a single CSP that is repeatedly step-wise extended. The number of steps that need to be considered to conclude failure depends on the temporal formulas and is finite [5]. The sequence of CSPs can be viewed as a single finite CSP consisting of finite domain variables and constraints of a standard type and thus is each time solvable by generic constraint programming systems. The top-down search is implemented by means of a regular backtrack search algorithm based on a variable domain splitting and combined with constraint propagation.

The variable domain splitting is controlled by domain-specific heuristics if available. We make use of the specialized reasoning techniques due to [20] for RCC8 and due to [17] for the cardinal directions. In these studies maximal tractable subclasses of the respective calculi are identified and corresponding polynomial decision procedures for non-temporal qualitative problems are discussed. In our terminology, if the domain of each relation variable in a qualitative array belongs to a certain class, then a certain sequence of domain splittings intertwined with constraint propagation finds a solving instantiation of the variables without backtracking if one exists. However, here we deal with a more complex set-up: sequences of qualitative arrays together with arbitrary temporal constraints connecting them. These techniques can then still serve as heuristics. We use them in our implementation to split the variable domains in such a way that one of the subdomains belongs to a maximal tractable subclass of the respective calculus.

We implemented the algorithm and both translations of temporal formulas to constraints in the ECL^{i}PS^e constraint programming system [22]. The resulting program is about 2000 lines of code. We used as constraint propagation hyper-arc consistency algorithms directly available in ECL^{i}PS^e in its fd and propia libraries and for array constraints through the implementation discussed in [6]. In the translations of the temporal formulas, following the insight from bounded model checking, redundancy in the resulting generation of constraints is reduced by sharing subformulas.

7 Case Study 1: Navigation

Consider a ship and three buoys forming a triangle. The problem is to generate a cyclic route of the ship around the buoys. We reason qualitatively with the cardinal directions of Example 2.

– First, we postulate that all objects occupy different positions:

$$\Box\,\forall a, b \in \mathcal{O}.\ a \neq b \ \rightarrow\ Q[a, b] \neq \mathsf{EQ}.$$

– Without loss of generality we assume that the buoy positions are given by

$$\Box\, Q[buoy_a, buoy_c] = \mathsf{NW},\ \Box\, Q[buoy_a, buoy_b] = \mathsf{SW},\ \Box\, Q[buoy_b, buoy_c] = \mathsf{NW}$$

and assume that the initial position of the ship is south of buoy c:

$$Q[ship, buoy_c] = \mathsf{S}.$$

– To ensure that the ship follows the required path around the buoys we stipulate:

$$\Box\,\big(Q[ship, buoy_c] = \mathsf{S} \quad\rightarrow\quad \Diamond(Q[ship, buoy_a] = \mathsf{W}\ \wedge \\ \Diamond\,(Q[ship, buoy_b] = \mathsf{N}\ \wedge \\ \Diamond\,(Q[ship, buoy_c] = \mathsf{E}\ \wedge \\ \Diamond\,(Q[ship, buoy_c] = \mathsf{S}\ )))))\big).$$

In this way we enforce an infinite circling of the ship around the buoys.

When fed with the above constraints our program generated the infinite path formed by the cycle through thirteen positions depicted in Fig. 4. The positions required to be visited are marked by bold circles. Each of them can be reached from the previous one through an atomic change in one or more qualitative relations between the ship and the buoys. One hour running time was not enough to succeed with the generic first-fail heuristic, but it took only 20 s to find the cycle using the Dir-specific heuristic. The array constraint translation reduced this slightly to 15 s.

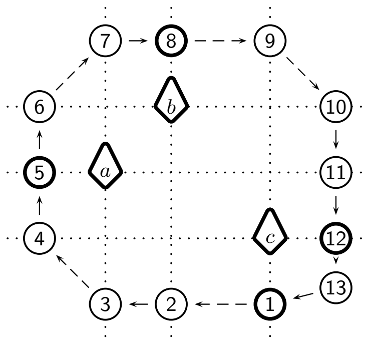


Fig. 4. Navigation path

The cycle found is a shortest cycle satisfying the specifications. Note that other, longer cycles exist as well. For example, when starting in position 1 the ship can first move to an 'intermediate' position between positions 1 and 2, characterized by:

$$Q[ship, buoy_c] = \mathsf{SW},\ Q[ship, buoy_a] = \mathsf{SE},\ Q[ship, buoy_b] = \mathsf{SE}.$$

We also examined a variant of this problem in which two ships are required to circle around the buoys while remaining in the N or NW relation w.r.t. each other. In this case the shortest cycle consisted of fifteen positions.

8 Case Study 2: Simulating of Juggling

Next, we consider a qualitative formalization of juggling. We view it as a process having an initialization phase followed by a proper juggling phase which is repeated. As such it fits well our qualitative simulation framework.

We consider two kinds of objects: the hands and the balls. For the sake of simplicity, we only distinguish the qualitative relations 'together', between a ball and a hand that holds it or between two touching balls, and 'apart'. This allows us to view the juggling domain as an instance of an existing topological framework: we identify 'together' and 'apart' with the relations meet and $\mathsf{disjoint}$ of the $\mathsf{RCC8}$ calculus.

In our concrete study, we assume a single juggler (with two hands) and three balls. We aim to recreate the three-ball-cascade, see [14, p. 8]. So we have five objects:

$$\begin{aligned}\mathcal{O} &:= \mathit{Hands} \cup \mathit{Balls},\\ \mathit{Hands} &:= \{\mathit{left\text{-}hand}, \mathit{right\text{-}hand}\},\\ \mathit{Balls} &:= \{\ \mathit{ball}_i \mid i \in \{1,2,3\}\ \}.\end{aligned}$$

The constraints are as follows.

– We only represent the relations of being 'together' or 'apart':

$$\Box\, \forall x, y \in \mathcal{O}.\ (x \neq y \ \rightarrow\ Q[x,y] \in \{\mathsf{meet}, \mathsf{disjoint}\}).$$

– The hands are always apart:

$$\Box\, Q[\mathit{left\text{-}hand}, \mathit{right\text{-}hand}] = \mathsf{disjoint}.$$

– A ball is never in both hands at the same time:

$$\Box\, \forall b \in \mathit{Balls}.\ \neg\,(Q[\mathit{left\text{-}hand}, b] = \mathsf{meet}\ \wedge\ Q[\mathit{right\text{-}hand}, b] = \mathsf{meet})\,.$$

– From some state onwards, at any time instance at most one ball is in any hand:

$$\begin{aligned}&\Diamond\Box\,(\forall b \in \mathit{Balls}.\ \forall h \in \mathit{Hands}.\ Q[b,h] = \mathsf{meet}\ \rightarrow\\ &\qquad \forall b_2 \in \mathit{Balls}.\ b \neq b_2\ \rightarrow\ \forall h_2 \in \mathit{Hands}.\ Q[b_2,h_2] = \mathsf{disjoint}).\end{aligned}$$

– Two balls touch if and only if they are in the same hand:

$$\begin{aligned}&\Box\,(\forall b_1, b_2 \in \mathit{Balls}.\ b_1 \neq b_2\ \rightarrow\\ &\qquad (Q[b_1,b_2] = \mathsf{meet}\ \leftrightarrow\ \exists h \in \mathit{Hands}.\ (Q[h,b_1] = \mathsf{meet}\ \wedge\ Q[h,b_2] = \mathsf{meet}))).\end{aligned}$$

– A ball thrown from one hand remains in the air until it lands in the other hand:

$$\begin{aligned}&\Box\,(\forall b \in \mathit{Balls}.\ \forall h_1, h_2 \in \mathit{Hands}.\ h_1 \neq h_2 \wedge Q[h_1,b] = \mathsf{meet} \rightarrow\\ &\qquad Q[h_1,b] = \mathsf{meet}\ \mathsf{U}\ (Q[h_1,b] = \mathsf{disjoint}\ \wedge\ Q[h_2,b] = \mathsf{disjoint}\ \wedge\\ &\qquad\qquad (Q[h_1,b] = \mathsf{disjoint}\ \mathsf{U}\ Q[h_2,b] = \mathsf{meet}))).\end{aligned}$$

– A ball in the air will land before any other ball that is currently in a hand,

$$\Box\,(\forall h_1, h_2 \in \mathit{Hands}.\ \forall b_1, b_2 \in \mathit{Balls}.\ Q[h_1, b_1] = \mathsf{disjoint} \wedge Q[h_2, b_2] = \mathsf{meet} \rightarrow \\ Q[h_2, b_2] = \mathsf{meet}\ \mathsf{U}\ ((\forall h \in \mathit{Hands}.\ Q[h, b_2] = \mathsf{disjoint}) \\ \mathsf{U}\ (\exists h \in \mathit{Hands}.\ Q[h, b_1] = \mathsf{meet}))).$$

– No two balls are thrown at the same time:

$$\Box\,(\forall b_1, b_2 \in \mathit{Balls}.\ b_1 \neq b_2 \rightarrow \forall h_1, h_2 \in \mathit{Hands}. \\ \neg(Q[b_1, h_1] = \mathsf{meet} \wedge \bigcirc\, Q[b_1, h_1] = \mathsf{disjoint}\ \wedge \\ Q[b_2, h_2] = \mathsf{meet} \wedge \bigcirc\, Q[b_2, h_2] = \mathsf{disjoint})).$$

– A hand can interact with only one ball at a time:

$$\Box\, \forall h \in \mathit{Hands}.\ \forall b_1 \in \mathit{Balls}. \\ (Q[h, b_1] = \mathsf{meet}\ \wedge\ \bigcirc\, Q[h, b_1] = \mathsf{disjoint}\ \vee \\ Q[h, b_1] = \mathsf{disjoint}\ \wedge\ \bigcirc\, Q[h, b_1] = \mathsf{meet})\ \rightarrow \\ \forall b_2 \in \mathit{Balls}.\ b_1 \neq b_2\ \rightarrow \\ (Q[h, b_2] = \mathsf{meet}\ \rightarrow\ \bigcirc\, Q[h, b_2] = \mathsf{meet})\ \wedge \\ (Q[h, b_2] = \mathsf{disjoint}\ \rightarrow\ \bigcirc\, Q[h, b_2] = \mathsf{disjoint}).$$

Initially balls 1 and 2 are in the left hand, while ball 3 is in the right hand:

$$Q[\mathit{left\text{-}hand}, \mathit{ball}_1] = \mathsf{meet}, Q[\mathit{left\text{-}hand}, \mathit{ball}_2] = \mathsf{meet}, Q[\mathit{right\text{-}hand}, \mathit{ball}_3] = \mathsf{meet}.$$

Note that the constraints enforce that the juggling continues forever. Our program finds an infinite simulation in the form of a path $[1..2][3..8]^*$; see Fig. 5. The running time was roughly 100 s using the generic first-fail heuristic; the RCC8-specific heuristic, resulting in 43 min, was not useful.

We stress the fact that the complete specification of this problem is not straightforward. In fact, the interaction with our program revealed that the initial specification was incomplete. This led us to the introduction of the last constraint.

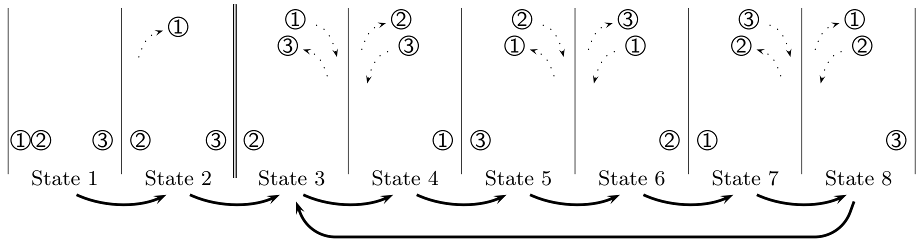


Fig. 5. Simulation of Juggling

Aspect Integration: Adding Cardinal Directions

The compositional nature of the 'relations as variables' approach makes it easy to integrate several spatial aspects (e.g., topology *and* size, direction, shape etc.) in one model. For the non-temporal case, we argued in [7] that the background knowledge on linking different aspects can be viewed as just another integrity constraint. Here we show that also qualitative simulation and aspect integration combine easily, by extending the juggling example with the cardinal directions.

As the subject of this paper is modelling and solving, not the actual inter-aspect background knowledge, we only explain the integration of the three relations $\mathsf{meet}, \mathsf{disjoint}, \mathsf{equal}$ with the cardinal directions Dir. We simply add

$$\mathsf{link}(Q[a,b],\, Q_{\mathsf{Dir}}[a,b]) \;:=\; (Q[a,b] = \mathsf{equal}) \leftrightarrow (Q_{\mathsf{Dir}}[a,b] = \mathsf{EQ})$$

as the aspect linking constraint. It refers to the two respective qualitative arrays and is stated for all spatial objects a, b. We add the following domain-specific requirements to our specification of juggling:

$$Q_{\mathsf{Dir}}[\mathit{left\text{-}hand}, \mathit{right\text{-}hand}] = \mathsf{W};$$
$$\Box\, \forall b \in \mathit{Balls}.\ \forall h \in \mathit{Hands}.\ Q[b,h] = \mathsf{meet} \rightarrow Q_{\mathsf{Dir}}[b,h] = \mathsf{N};$$
$$\Box\, \forall b \in \mathit{Balls}.\ Q[b, \mathit{left\text{-}hand}] = \mathsf{disjoint} \wedge Q[b, \mathit{right\text{-}hand}] = \mathsf{disjoint} \rightarrow$$
$$Q_{\mathsf{Dir}}[b, \mathit{left\text{-}hand}] \neq \mathsf{N} \wedge Q_{\mathsf{Dir}}[b, \mathit{right\text{-}hand}] \neq \mathsf{N}.$$

We state thus that a ball in a hand is 'above' that hand, and that a ball is not thrown straight upwards.

This simple augmentation of the juggling domain with directions yields the same first simulation as in the single-aspect case, but now with the $\mathsf{RCC8}$ and Dir components. The ball/hand relation just alternates between N and NW (or NE).

We emphasize that it was straightforward to extend our implementation to achieve the integration of two aspects. The constraint propagation for the link constraints is achieved by the same generic hyper-arc consistency algorithm used for the single-aspect integrity constraints. This is in contrast to the 'relations as constraints' approach which requires new aspect integration *algorithms*; see, e.g., the bipath-consistency algorithm of [13].

9 Conclusions

Related Work. The most common approach to qualitative simulation is the one discussed in [15, Chapter 5]. For a recent overview see [16]. It is based on a qualitative differential equation model (QDE) in which one abstracts from the usual differential equations by reasoning about a finite set of symbolic values (called *landmark values*). The resulting algorithm, called *QSIM*, constructs the tree of possible evolutions by repeatedly constructing the successor states. During this process CSPs are generated and solved. This approach is best suited to simulate the evolution of physical systems.

Our approach is inspired by the qualitative spatial simulation studied in [9], the main features of which are captured by the composition table and the neighbourhood relation discussed in Example 1. The distinction between the integrity and inter-state constraints is introduced there; however, the latter only link consecutive states in the simulation. As a result, our case studies are beyond their reach. Our experience with our program moreover suggests that the algorithm of [9] may not be a realistic basis for an efficient qualitative reasoning system.

To our knowledge the '(qualitative) relations as variables' approach to modelling qualitative reasoning was first used in [21], to deal with the qualitative temporal reasoning due to [1]. In [20] this approach is used in an argument to establish the quality of a generator of random scenarios, whilst the main part of this paper uses the customary 'relations as constraints' approach. In [2, pages 30-33] we applied the 'relations as variables' approach to model a qualitative spatial reasoning problem. In [7] we used it to deal in a simple way with aspect integration and in [3] to study qualitative planning problems.

In [8] various semantics for a programming language that combines temporal logic operators with constraint logic programming are studied. Finally, in the TLPlan system of [4] temporal logic is used to support the construction of control rules that guide plan search. The planning system is based on an incremental forward-search, so the temporal formulas are unfolded one step at a time, in contrast to the translation into constraints in our constraint-based system.

Summary. We introduced a constraint-based framework for describing infinite qualitative simulations. Simulations are formalized by means of inter-state constraints that are defined using linear temporal logic. This results in a high degree of expressiveness. These constraints are translated into a generic target constraint language. The qualitative relations are represented as domains of constraint variables. This makes the considered CSPs finite, allows one to use hyper-arc consistency as constraint propagation, and to integrate various knowledge aspects in a straightforward way by simply adding linking constraints.

We implemented this approach in a generic constraint programming system, ECL^{i}PS^e, using techniques from bounded model checking and by experimenting with various heuristics. The resulting system is conceptually simple and easy to use and allows for a straightforward modification of the problem specifications. We substantiated these claims by means of two detailed case studies.

References

1. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
2. K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
3. K. R. Apt and S. Brand. Constraint-based qualitative simulation. In *Proc. of 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 26–34. IEEE Computer Society, 2005.
4. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116, 2000.

5. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. *Advances in Computers*, volume 58, chapter Bounded Model Checking. Academic press, 2003.
6. S. Brand. Constraint propagation in presence of arrays. In K. R. Apt, R. Barták, E. Monfroy, and F. Rossi, editors, *Proc. of 6th Workshop of the ERCIM Working Group on Constraints*, 2001.
7. S. Brand. Relation variables in qualitative spatial reasoning. In S. Biundo, T. Frühwirth, and G. Palm, editors, *Proc. of 27th German Annual Conference on Artificial Intelligence (KI'04)*, volume 3238 of *LNAI*, pages 337–350. Springer, 2004.
8. Ch. Brzoska. Temporal logic programming and its relation to constraint logic programming. In V. A. Saraswat and K. Ueda, editors, *Proc. of International Symposium on Logic Programming (ISLP'91)*, pages 661–677. MIT Press, 1991.
9. Z. Cui, A. G. Cohn, and D. A. Randell. Qualitative simulation based on a logical formalism of space and time. In P. Rosenbloom and P. Szolovits, editors, *Proc. of 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 679–684. AAAI Press, 1992.
10. M. J. Egenhofer. Reasoning about binary topological relations. In O. Günther and H.-J. Schek, editors, *Proc. of 2nd International Symposium on Large Spatial Databases (SSD'91)*, volume 525 of *LNCS*, pages 143–160. Springer, 1991.
11. M. T. Escrig and F. Toledo. *Qualitative Spatial Reasoning: Theory and Practice. Application to Robot Navigation*, volume 47 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 1998.
12. A. U. Frank. Qualitative spatial reasoning about distance and directions in geographic space. *Journal of Visual Languages and Computing*, 3:343–373, 1992.
13. A. Gerevini and J. Renz. Combining topological and size constraints for spatial reasoning. *Artificial Intelligence*, 137(1-2):1–42, 2002.
14. C. Gifford. *Juggling*. Usborne Publishing, 1995.
15. B. Kuipers. *Qualitative reasoning: modeling and simulation with incomplete knowledge*. MIT Press, 1994.
16. B. Kuipers. *Encyclopedia of Physical Science and Technology*, chapter Qualitative simulation, pages 287–300. Academic Press, third edition, 2001.
17. G. Ligozat. Reasoning about cardinal directions. *Journal of Visual Languages and Computing*, 9(1):23–44, 1998.
18. R. Mohr and G. Masini. Good old discrete relaxation. In Y. Kodratoff, editor, *Proc. of European Conference on Artificial Intelligence (ECAI'88)*, pages 651–656. Pitman publishers, 1988.
19. D. A. Randell, A. G. Cohn, and Z. Cui. Computing transitivity tables: A challenge for automated theorem provers. In *Proc. of 11th Conference on Automated Deduction (CADE'92)*, volume 607 of *LNAI*, pages 786–790. Springer, 1992.
20. J. Renz and B. Nebel. Efficient methods for qualitative spatial reasoning. *Journal of Artificial Intelligence Research*, 15:289–318, 2001.
21. E. P. K. Tsang. The consistent labeling problem in temporal reasoning. In K. S. H. Forbus, editor, *Proc. of 6th National Conference on Artificial Intelligence (AAAI'87)*, pages 251–255. AAAI Press, 1987.
22. M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.

Algorithms for Stochastic CSPs

Thanasis Balafoutis and Kostas Stergiou

Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece

Abstract. The Stochastic CSP (SCSP) is a framework recently introduced by Walsh to capture combinatorial decision problems that involve uncertainty and probabilities. The SCSP extends the classical CSP by including both decision variables, that an agent can set, and stochastic variables that follow a probability distribution and can model uncertain events beyond the agent's control. So far, two approaches to solving SCSPs have been proposed; backtracking-based procedures that extend standard methods from CSPs, and scenario-based methods that solve SCSPs by reducing them to a sequence of CSPs. In this paper we further investigate the former approach. We first identify and correct a flaw in the forward checking (FC) procedure proposed by Walsh. We also extend FC to better take advantage of probabilities and thus achieve stronger pruning. Then we define arc consistency for SCSPs and introduce an arc consistency algorithm that can handle constraints of any arity.

1 Introduction

Representation and reasoning with uncertainty is an important issue in constraint programming since uncertainty is inherent in many real combinatorial problems. To model such problems, many extensions of the classical CSP have been proposed (see [9] for a detailed review). The Stochastic CSP (SCSP) is a framework that can be used to model combinatorial decision problems involving uncertainty and probabilities recently introduced by Walsh [10]. The SCSP extends the classical CSP by including both decision variables, that an agent can set, and stochastic variables that follow a probability distribution and can model uncertain events beyond the agent's control. The SCSP framework is inspired by the stochastic satisfiability problem [6] and combines some of the best features of traditional constraint satisfaction, stochastic integer programming, and stochastic satisfiability.

The expressional power of the SCSP can help us model situations where there are probabilistic estimations about various uncertain actions and events, such as stock market prices, user preferences, product demands, weather conditions, etc. For example, in industrial planning and scheduling, we need to cope with uncertainty in future product demands. As a second example, interactive configuration requires us to anticipate variability in the users' preferences. As a final example, when investing in the stock market, we must deal with uncertainty in the future price of stocks.

SCSPs have recently been introduced and only a few solution methods have been proposed. In the initial paper, Walsh described a chronological backtracking and a forward checking procedure for binary problems [10]. These are extensions of the corresponding algorithms for CSPs that explore the space of policies in a SCSP. Alternatively, scenario-based methods, which solve a SCSP by reducing it to a sequence of CSPs, were introduced in [7]. This approach carries certain advantages compared to algorithms that operate on the space of policies. Most significantly, it can exploit existing advanced CSP solvers, without requiring the implementation of (potentially complicated) specialized search and propagation techniques. As a consequence, this approach is not limited to binary problems. However, the number of scenarios in a SCSP grows exponentially with the number of stages in the problem. Therefore, the scenario-based approach may not be applicable in problems with many stochastic variables and many stages.

In this paper we develop algorithms for SCSPs following the initially proposed approach based on the exploration of the policy space. We first identify and correct a flaw in the forward checking (FC) procedure proposed by Walsh. We also extend FC to take better advantage of probabilities and thus achieve stronger pruning. Then we define arc consistency (AC) for SCSPs and introduce an AC algorithm that can handle constraints of any arity. This allows us to implement a MAC algorithm that can operate on non-binary problems. Finally, we present some preliminary experimental results.

2 Stochastic Constraint Satisfaction

In this section we review the necessary definitions on SCSPs given in [10] and [7]. A *stochastic constraint satisfaction problem* (SCSP) is a 6-tuple$< X, S, D, P, C, \Theta >$ where X is a sequence of n variables, S is the subset of X which are stochastic variables, D is a mapping from X to domains, P is a mapping from S to probability distributions for the domains of the stochastic variables, C is a set of e constraints over X, and Θ is a mapping from constraints to threshold probabilities in the interval $[0, 1]$. Each constraint is defined by a subset of the variables in X and an, extensionally or intensionally specified, relation giving the allowed tuples of values for the variables in the constraint. A hard constraint, which must be always satisfied, has an associated threshold 1, while a "chance constraint" c_i, which may only be satisfied in some of the possible worlds, is associated with a threshold $\theta_i \in [0, 1]$. This means that the constraint must be satisfied in at least a fraction θ_i of the worlds.

For the purposes of this paper, we will follow [10] and assume that the problem consists only of a single global chance constraint which is the conjunction of all constraints in the problem. This global constraint must be satisfied in at least a fraction θ of the possible worlds. We will also assume that the stochastic variables are independent (as in [10]). This assumption limits the applicability of the SCSP framework but it can be lifted, as in other frameworks for uncertainty handling, such as *fuzzy* and *possibilistic* CSPs [4].

We will sometimes denote decision variables by xd_i and stochastic variables by xs_i. Accordingly, the sets of decision and stochastic variables in the problem will be denoted by Xd and Xs respectively. The domain of a variable x_i will be denoted by $D(x_i)$, and the variables that participate in a constraint c_i will be denoted by $var(c_i)$. We assume that in each constraint c_i the variables in $var(c_i)$ are sorted according to their order in X.

The backtracking algorithms of [10] explore the space of policies in a SCSP. A *policy* is a tree with nodes labelled with value assignments to variables, starting with the values of the first variable in X labelling the children of the root, and ending with the values of the last variable in X labelling the leaves. A variable whose next variable in X is a decision one corresponds to a node with a single child, while a variable whose next variable in X is a stochastic one corresponds to a node that has one child for every possible value of the following stochastic variable. Leaf nodes take value 1 if the assignment of values to variables along the path to the root satisfies all the constraints and 0 otherwise. Each path to a leaf node in a policy represents a different possible *scenario* (set of values for the stochastic variables) and the values given to decision variables in this scenario. Each scenario has an associated probability; if xs_i is the i-th stochastic variable in a path to the root, v_i is the value given to xs_i in this scenario, and $\text{prob}(xs_i \leftarrow v_i)$ is the probability that $xs_i = v_i$, then the probability of this scenario is: $\prod_i \text{prob}(xs_i \leftarrow v_i)$.

The satisfaction of a policy is the sum of the leaf values weighted by their probabilities. A policy satisfies the constraints iff its satisfaction is at least θ. In this case we say that the policy is *satisfying*. A SCSP is satisfiable iff it has a satisfying policy. The optimal satisfaction of a SCSP is the maximum satisfaction of all policies. Given a SCSP, two basic reasoning tasks are to determine if the satisfaction is at least θ and to determine the maximum satisfaction.

The simplest possible SCSP is a one-stage SCSP in which all of the decision variables are set before the stochastic variables. This models situations in which we must act now, trying to plan our actions in such a way that the constraints are satisfied (as much as possible) for whatever outcome of the later uncertain events. Alternatively, we may demand that the stochastic variables are set before the decision variables. A one stage SCSP is satisfiable iff there exist values for the decision variables so that, given random values for the stochastic variables, the constraints are satisfied in at least the given fraction of worlds. In a two stage SCSP, there are two sets (blocks) of decision variables, Xd_1 and Xd_2, and two sets of stochastic variables, Xs_1 and Xs_2. The aim is to find values for the variables in Xd_1, so that given random values for Xs_1, we can find values for Xd_2, so that given random values for Xs_2, the constraints are satisfied in at least the given fraction of worlds. An m stage SCSP is defined in an analogous way to one and two stage SCSPs.

SCSPs are closely related to *quantified* CSPs (QCSPs). A QCSP can be viewed as a SCSP where existential and universal variables correspond to decision and stochastic variables, respectively. In such a SCSP, all values of the stochastic variables have equal probability and the satisfaction threshold is 1.

3 Forward Checking

Forward Checking for SCSPs was introduced in [10] as an extension of the corresponding algorithm for CSPs. We first review this algorithm and show that it suffers from a flaw. We then show how this flaw can be corrected and how FC can be enhanced to achieve stronger pruning.

Figure 1 depicts the `FC` procedure of [10]. `FC` instantiates the variables in the order they appear in X. On meeting a decision variable, `FC` tries each value in its domain in turn. The maximum value is returned to the previous recursive call. On meeting a stochastic variable, `FC` tries each value in turn, and returns the sum of all the answers to the subproblems weighted by the probabilities of their occurrence. On instantiating a decision or stochastic variable, `FC` checks forward and prunes values from the domains of future variables which break constraints. If the instantiation of a decision or stochastic variable breaks a constraint, the algorithm returns 0. If all variables are instantiated without breaking any constraint, `FC` returns 1.

```
Procedure FC($i, \theta_l, \theta_h$)
if $i > n$ then return 1
$\theta := 0$
for each $v_j \in D(x_i)$
    if prune($i, j$) $= 0$ then
      if check($x_i \leftarrow v_j, \theta_l$) then
        if $x_i \in Xs$ then
          $p :=$ prob($x_i \leftarrow v_j$)
          $q_i := q_i - p$
          $\theta := \theta + p \times$ FC($i+1, (\theta_l - \theta - q_i)/p, (\theta_h - \theta)/p$)
          restore($i$)
          if $\theta + q_i < \theta_l$ then return $\theta$
          if $\theta > \theta_h$ then return $\theta$
        else
          $\theta := \max(\theta, FC(i+1, \max(\theta, \theta_l), \theta_h))$
          restore($i$)
          if $\theta > \theta_h$ then return $\theta$
      else restore($i$)
return $\theta$
```

```
function check($x_i \leftarrow v_j, \theta_l$)
  for $k := i + 1$ to $n$
    dwo := true
    for each $v_l \in D(x_k)$
      if prune($k, l$) $= 0$ then
        if inconsistent($x_i \leftarrow v_j, x_k \leftarrow v_l$) then
          prune($k, l$) $:= i$
          if $x_k \in Xs$ then
            $q_k := q_k$ - prob($x_k \leftarrow v_l$)
            if $q_k < \theta_l$ then return false
        else dwo := false
    if dwo then return false
  return true
```

Fig. 1. The `FC` algorithm of [10]

In Figure 1, a 2-dimensional array prune(i, j) is used to record the depth at which the value $v_j \in D(x_i)$ is removed by forward checking. Each stochastic variable xs_i has an upper bound, q_i, on the probability that the values left in $D(xs_i)$ can contribute to a solution. This is initially set to 1. The upper and lower bounds, θ_h and θ_l are used to prune search. By setting $\theta_l = \theta_h = \theta$, we can determine if the optimal satisfaction is at least θ. By setting $\theta_l = 0$ and $\theta_h = 1$, we can determine the optimal satisfaction.

The calculation of these bounds in recursive calls is done as follows. Suppose that the current assignment to a stochastic variable returns a satisfaction of θ_0. We can ignore other values for this variable if $\theta + p\times\ \theta_0 \geq \theta_h$. That is, if $\theta_0 \geq (\theta_h$-$\theta)/p$. This gives the upper bound in the recursive call to `FC` on a stochastic variable. Alternatively, we cannot hope to satisfy the constraints adequately if $\theta + p\times\ \theta_0 + q_i \leq \theta_l$ as q_i is the maximum that the remaining values can contribute to the satisfaction. That is, if $\theta_0 \leq (\theta_l$-θ-$q_i)/p$. This gives the lower bound in the recursive call to `FC` on a stochastic variable. Finally, suppose that the current assignment to a decision variable returns a satisfaction of θ. If this is more than θ_l, then any other values must exceed θ to be part of a better policy. Hence, we can replace the lower bound in the recursive call to `FC` on a decision variable by max(θ, θ_l). Procedure `restore`, which is not shown, is called when a tried assignment is rejected and when a backtrack occurs, to restore values that have been removed from future variables and reset the value of q_i for stochastic variables.

Checking forwards fails if any variable has a domain wipeout (dwo), or (crucially) if a stochastic variable has so many values removed that we cannot hope to satisfy the constraints. When forward checking removes some value v_j from xs_i, `FC` reduces q_i by prob($xs_i \leftarrow v_j$), the probability that xs_i takes the value v_j. This reduction on q_i is undone on backtracking. If `FC` ever reduces q_i to less than θ_l, it backtracks as it is impossible to set xs_i and satisfy the constraints adequately.

3.1 A Flaw in FC

As the next example shows, this last claim can be problematic. When the current variable is a stochastic one, there are cases where, even if q_i is reduced to less than θ_l, the algorithm should continue going forward instead of backtracking because the satisfaction of the future subproblem may contribute to the total satisfaction. The example considers the case where we look for the maximum satisfaction.

Example 1. Consider a problem consisting of one decision variable xd_1 and two stochastic variables xs_2, xs_3, all with $\{0, 1\}$ domains. The probabilities of the values are shown in Figure 2a where the search tree for the problem is depicted. There is a constraint between xd_1 and xs_2 disallowing the tuple $<xd_1 \leftarrow 0, xs_2 \leftarrow 1>$. There is also a constraint between xs_2 and xs_3 disallowing the tuple $<xs_2 \leftarrow 1, xs_3 \leftarrow 0>$. Assume that we seek the maximum satisfaction of the problem. That is, initially $\theta_l = 0$ and $\theta_h = 1$.

`FC` will first instantiate xd_1 to 0 and forward check this assignment. As a result, value 1 of xs_2 will be deleted and the dashed nodes will be pruned. Then the algorithm will explore the non-pruned subtree below $xd_1 \leftarrow 0$ and eventually will backtrack to xd_1. At this point θ will be 0.5 (i.e. the satisfaction of the explored subtree). Now when `FC` moves forward to instantiate xs_2, θ_l will be set to max(θ_l,θ)=max(0,0.5)=0.5. The subtree below $xs_2 \leftarrow 0$, weighted by prob($xs_2 \leftarrow 0$), gives 0.5 satisfaction. When assigning 1 to xs_2, `check` will return false because value 0 of xs_3 will be removed and the remaining probability in the domain of xs_3 will be $0.4<\theta_l$. Therefore, FC will backtrack and terminate, incorrectly returning 0.5 as the maximum satisfaction. Clearly, the maximum satisfaction, which is achieved by the policy depicted with bold edges, is 0.7.

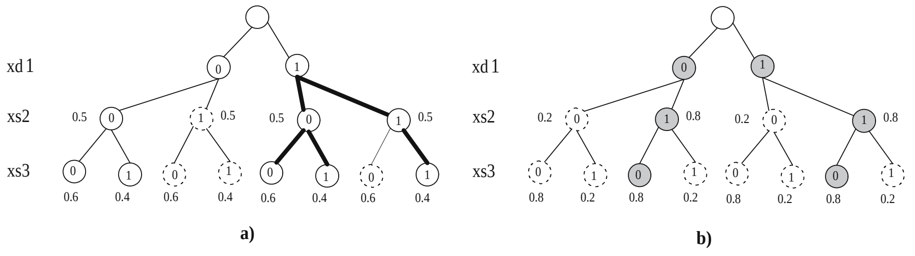


Fig. 2. Search trees of Examples 1 and 2

Function `check` correctly returns a failure when the current variable is a decision one and for some future stochastic variable xs_i forward checking reduces q_i below θ_l. In this case, there is not point in exploring the subtree below the current assignment. However, when the current variable is a stochastic one and for some future stochastic variable, q_i falls below θ_l, it is not certain that the currently explored policy cannot yield satisfaction greater than the threshold. What we need is a way to determine if the maximum satisfaction offered by the current stochastic variable is enough to lift the total satisfaction over the lower satisfaction bound or not. Therefore, we need to take into account the following quantities: 1) the already computed satisfaction of the previously assigned values of the current variable, 2) the maximum satisfaction of the subtree below the current assignment, 3) the sum of the probabilities of the following values of the current variable (i.e. the maximum satisfaction that they can contribute). If the sum of these quantities is lower than θ_l then the current assignment can be safely rejected. Otherwise, we must continue expanding it. This idea is formulated in more detail further below, after we describe a simple way to enhance the pruning power of FC.

3.2 Improving FC

We can save search effort by performing stronger pruning inside function `check`. When making forward checks and removing values from future stochastic variables, the FC algorithm of [10] exploits only a "local" view of the future problem. But as values are removed from future stochastic variables, the maximum possible satisfaction of the current assignment is reduced. FC fails to exploit this because it considers value removals from any future stochastic variable as "independent" of value removals from other future stochastic variables. However, it is possible that enough values are removed from a number of stochastic variables so that the maximum possible satisfaction of the current assignment cannot exceed

θ_l. The maximum possible satisfaction of an assignment v_j to the current variable x_i is equal to $\prod_{s=i+1}^{n} \sum_{t=1}^{|D(x_s)|} \text{prob}(x_s \leftarrow v_t)$ (weighted by the probability of $x_i \leftarrow v_j$ if x_i is stochastic), where only values that have not been pruned are considered. In words, we sum the probabilities of the remaining values for all future stochastic variables and multiply the sums. Before explaining how we can exploit this, we present an example that demonstrates the savings in search effort that can be achieved through such reasoning.

Example 2. Consider a problem consisting of one decision variable xd_1 and two stochastic variables xs_2, xs_3, all with $\{0, 1\}$ domains. The probabilities of the values are shown in Figure 2b where the search tree of the problem is depicted. There is a constraint between xd_1 and xs_2 disallowing the tuples $<xd_1 \leftarrow 0, xs_2 \leftarrow 0>$ and $<xd_1 \leftarrow 1, xs_2 \leftarrow 0>$. There is also a constraint between xd_1 and xs_3 disallowing the tuples $<xd_1 \leftarrow 0, xs_3 \leftarrow 1>$ and $<xd_1 \leftarrow 1, xs_3 \leftarrow 1>$. Assume that we are looking for the maximum satisfaction.

FC will first instantiate xd_1 to 0 and forward check this assignment. As a result, values 0 and 1 will be removed from the $D(xs_2)$ and $D(xs_3)$ respectively. Since q_2 and q_3 do not fall below θ_l, the algorithm will continue to make the instantiations $xs_2 \leftarrow 1$ and $xs_3 \leftarrow 0$. After backtracking to xd_1, the current satisfaction θ for xd_1 will be 0.64. Now FC will instantiate xd_1 to 1, forward check the assignment, remove values 0 and 1 from the domains of xs_2 and xs_3, and proceed to instantiate the stochastic variables. Similarly as before, the satisfaction of assignment $xd_1 \leftarrow 1$ will be 0.64. Therefore, FC will return the maximum satisfaction among the values of xd_1, which is 0.64. To find this, FC needs to visit six nodes in the search tree (the gray nodes in Figure 2b).

Consider again the point when after the satisfaction of assignment $xd_1 \leftarrow 0$ has been computed, the algorithm instantiates xd_1 to 1. Forward checking removes values 0 and 1 from $D(xs_2)$ and $D(xs_3)$ respectively, and as a result the maximum possible satisfaction of assignment $xd_1 \leftarrow 1$ is equal to $\text{prob}(xs_2 \leftarrow 1) \times \text{prob}(xs_3 \leftarrow 0) = 0.64$. This is not greater than the satisfaction of assignment $xd_1 \leftarrow 0$, and therefore, the algorithm need not proceed to instantiate the stochastic variables. Since there is no other value in $D(xd_1)$, we can determine that the satisfaction of the problem is 0.64. In this way, the problem is solved visiting four instead of six nodes.

Figure 3 depicts the improved `check` function of FC. The identified flaw is corrected in lines 10-13 where we differentiate between the case where the current variable is a stochastic one and the case where it is a decision one. In both cases we first compute ζ_i; the maximum satisfaction that the current assignment can yield. This is computed as the product of the sums of probabilities of the values that are left in the domains of the future stochastic variables. In this way we get a better estimation of the maximum satisfaction that the current assignment can provide and the efficiency of the algorithm, compared to the version given in [10], is improved. Note that ζ_i is computed each time FC has filtered the domain of a future variable. Alternatively, we can compute it once after FC has finished with all future variables. In this case we can save repeating some computations but may perform redundant consistency checks.

```
function check($x_i \leftarrow v_j, q_i, \theta_l, \theta$)
1:  $q_i := q_i$ - prob($x_i \leftarrow v_j$)
2:  for $k := i + 1$ to $n$
3:    dwo := true
4:    for each $v_l \in D(x_k)$
5:      if prune($k, l$) = 0 then
6:        if inconsistent($x_i \leftarrow v_j, x_k \leftarrow v_l$) then
7:          prune($k, l$) := $i$
8:          if $x_k \in Xs$ then
9:            $\zeta_i := \prod_{s=i+1}^{n} \sum_{t=1}^{|D(x_s)|}$ prob($x_s \leftarrow v_t$)
10:           if $x_i \in Xs$ then
11:             if $\zeta_i \times prob(x_i \leftarrow v_j) + \theta + q_i < \theta_l$ then return false
12:           else
13:             if $\zeta_i < \theta_l$ then return false
14:       else dwo := false
15:   if dwo then return false
16: return true
```

Fig. 3. The improved check function of FC

If the current variable is a decision one and ζ_i falls below θ_l then we return false as is is not possible to extend the current assignment in a way that the threshold is satisfied. If the current variable is a stochastic one then we multiply ζ_i with the probability of the current assignment, add the satisfaction (θ) yielded by previously tried assignments to the current variable, add the sum of probabilities (q_i) of the remaining values for the current variable, and compare the resulting quantity with θ_l. If it is lower then we return fail because we know that there is no way to extend the current assignment, so that the threshold is satisfied, even if the current assignment and the remaining assignments to the current variable yield the maximum possible satisfaction.

4 Arc Consistency

Arc consistency (AC) is an important concept in CSPs since it is the basis of constraint propagation in most CSP solvers. In this section we first define AC for SCSPs and then describe an AC algorithm for SCSPs that can handle constraints of any arity. We show that, apart from the case of domain wipeout, failure can also be determined when enough values are removed from stochastic variables. We also introduce a specialized pruning rule that can be used to remove values from certain decision variables.

Before defining AC, we give a definition of consistency for values of decision variables. To do this, we adjust the corresponding definitions for QCSPs given in [2,3]. Intuitively, a value $v \in D(xd_i)$ is inconsistent if the assignment $xd_i \leftarrow v$ cannot participate in any satisfying policy.

Definition 1. A value $v \in D(xd_i)$ is *consistent* iff there is a satisfying policy, in which one scenario at least, includes the assignment $x_i \leftarrow v$.

Given the above definition, determining the consistency of a value involves finding all solutions (satisfying policies) to a SCSP. The definition of AC and the development of relevant filtering algorithms can hopefully help us perform pruning by local reasoning. We first give some necessary notation. Given a SCSP $A =< X, S, D, P, C, \Theta >$ we denote by A_{c_j} the SCSP in which only one constraint $c_j \in C$ is considered, i.e. the SCSP $< X, S, D, P, c_j, \theta_j >$. $\tau[x_i]$ gives the value that variable x_i takes in tuple τ. A tuple of assignments τ is *valid* if none of the values in τ has been removed from the domain of the corresponding variable. A tuple τ of a constraint c_j supports a value $v \in x_i$ iff $\tau[x_i] = v$, τ is valid, and τ is allowed by c_j.

Definition 2. A value $v \in D(xd_i)$ is *arc consistent* iff, for every constraint $c_j \in C$, v is consistent in A_{c_j}. A value $v \in D(xs_i)$ is arc consistent iff, for every constraint $c_j \in C$, there is a tuple that supports it. A SCSP is arc consistent iff all values of all variables are arc consistent.

Note that we differentiate between decision and stochastic variables. The definition of AC for values of decision variables subsumes the classical AC definition (which is used for values of stochastic variables). The above definition covers the general case where they may be multiple chance constraints. But in the problems considered here, where there is only a single global chance constraint, determining if a given SCSP is AC is a task just as hard as solving it. This is analogous to achieving AC in a classical CSP where all constraints are combined in a conjunction.

In the following we describe an algorithm that is not complete (i.e. it does not compute the AC-closure of a given SCSP) but can achieve pruning of some arc inconsistent values through local reasoning, and therefore in some cases detect arc inconsistency. In addition, the algorithm can determine failure if the maximum possible satisfaction falls below θ_l because of deletions from the domains of stochastic variables. The AC algorithm we use as basis is GAC2001/3.1 [1]. This is a coarse-grained (G)AC algorithm that does not require complicated data structures, while it achieves an optimal worst-case time complexity in its binary version. In addition to these features, GAC2001/3.1 facilitates the implementation of a specialized pruning rule that can remove arc inconsistent values from certain decision variables through local reasoning. The motivation for this rule is demonstrated in the following example.

Example 3. Consider a problem consisting of two decision variables xd_1 and xd_2 and two stochastic variables xs_3, xs_4, all with $\{0, 1\}$ domains. The probabilities of the values are shown in Figure 4 where the search tree of the problem is depicted. There is a ternary constraint c_1 with $var(c_1) = \{xd_2, xs_3, xs_4\}$ which disallows tuples $<xd_2 \leftarrow 0, xs_3 \leftarrow 0, xs_4 \leftarrow 0>$ and $<xd_2 \leftarrow 0, xs_3 \leftarrow 1, xs_3 \leftarrow 1>$. Assume that we are trying to determine if the satisfaction is at least 0.6.

It is easy to see that any policy which includes assignment $xd_2 \leftarrow 0$ cannot achieve more than 0.5 satisfaction since assigning 0 to xd_2 leaves $\{xs_3 \leftarrow 0, xs_4 \leftarrow 1\}$ and $\{xs_3 \leftarrow 1, xs_4 \leftarrow 0\}$ as the only possible sets of assignments for

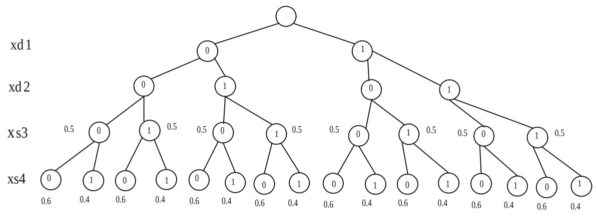


Fig. 4. Search tree of Example 3

variables xs_3 and xs_4, and these together yield 0.5 satisfaction. Therefore, we can safely prune the search tree by deleting value 0 of xd_2 prior to search.

We can generalize the idea illustrated in the example to the case where we reach a block of consecutive decision variables during search. Then if we identify certain values of these variables that, if assigned, result in policies which cannot yield satisfaction more than the current lower bound θ_l, then we know that these values are arc inconsistent and can thus prune them. We have incorporated this idea in the AC algorithm described below. Similar reasoning can be applied on decision variables further down in the variable sequence (i.e. not in the current block). However, identifying arc inconsistent values for such variables is an expensive process since it requires search.

Our algorithm for arc consistency in SCSPs is shown in Figure 5. Before explaining the algorithm we give some necessary notation and definitions.

– We assume that the tuples in each constraint are ordered according to the lexicographic ordering. In the while loop of line 26, NIL denotes that all tuples in a constraint have been searched.
– As in GAC2001/3.1, Last$((x_i, v), c_j)$ is the most recently discovered tuple in c_j that supports value $v \in D(x_i)$, where $x_i \in var(c_j)$. Initially, each Last$((x_i, v), c_j)$ is set to 0. c_var denotes the currently instantiated variable. If the algorithm is used for preprocessing, c_var is 0.
– When we say that "variable x_i belongs to the current stage in X" we mean that c_var is a decision variable and x_i belongs to the same block of variables as c_var. In case `Stochastic_AC` is used for preprocessing, we say "variable x_i belongs to the first stage in X" meaning that the first block of variables in X is composed of decision variables and x_i belongs to this block.
– $\theta_{(x_i,v),c_j}$ holds the maximum satisfaction that can be achieved by the possible assignments of the stochastic variables after decision variable x_i in $var(c_j)$ if value v is given to x_i. This is calculated by summing the probabilities of the tuples that support $x_i \leftarrow v$ in c_j. In this context, the probability of a tuple $\tau =<\ldots, x_i \leftarrow v, \ldots>$ is the product of the probabilities of the values that the stochastic variables **after** x_i take in τ.

`Stochastic_AC` uses a queue (or stack) of variable-constraint pairs. Essentially it operates in a similar way to GAC2001/3.1 with additional fail detection and

```
function Stochastic_AC($c\_var, \theta_l$)
1:  $Q \leftarrow \{x_i, c_j | c_j \in C, x_i \in var(c_j)\}$
2:  if $c\_var = 0$ then
3:    for each $(x_i, c_j) | x_i \in var(c_j), x_i \in Xd$ and $\exists x_m \in Xs, m > i$ and $x_m \in var(c_j)$
4:      for each $v \in D(x_i)$
5:        $\theta_{(x_i,v),c_j} \leftarrow 0$
6:        for $\tau$=Last($(x_i, v), c_j$) to last tuple in $c_j$
7:          if $\tau[x_i] = v$ and $\tau$ is valid and $\tau$ is allowed by $c_j$ then
8:            $\theta_\tau \leftarrow \prod_{s=x_i+1}^{|var(c_j)|}$ prob$(x_s \leftarrow \tau[x_s])$
9:            $\theta_{(x_i,v),c_j} \leftarrow \theta_{(x_i,v),c_j} + \theta_\tau$
10:       if $x_i$ belongs to the first stage in $X$ and $\theta_{(x_i,v),c_j} < \theta_l$ then
11:         remove $v$ from $D(x_i)$
12:         if $D(x_i)$ is wiped out then return false
13: while $Q$ not empty do
14:   select and remove a pair $(x_i, c_j)$ from $Q$
15:   fail $\leftarrow$ false
16:   if Revise($x_i, c_j, c\_var, \theta_l$,fail)
17:     if fail=true or a domain is wiped out then return false
18:     $Q \leftarrow Q \cap \{(x_k, c_m | c_m \in C, x_i, x_k \in var(c_m), m \neq j, i \neq k\}$
19: return true

function Revise($x_i, c_j, c\_var, \theta_l$,fail)
20: DELETION $\leftarrow$ FALSE
21: for each value $v \in D(x_i)$
22:   if Last($(x_i, v), c_j$) is not valid then
23:     if $x_i \in Xd$ and $\exists x_m \in Xs, m > i$ and $x_m \in var(c_j)$ then
24:       $\theta_{(x_i,v),c_j} \leftarrow \theta_{(x_i,v),c_j} - \theta_{Last((x_i,v),c_j)}$
25:     $\tau \leftarrow$ next tuple in the lexicographic ordering
26:     while $\tau \neq NIL$
27:       if $\tau[x_i] = v$ and $\tau$ is allowed by $c_j$ then
28:         if $\tau$ is valid then break
29:         else if $x_i \in Xd$ and $\exists x_m \in Xs, m > i$ and $x_m \in var(c_j)$ then
30:           $\theta_{(x_i,v),c_j} \leftarrow \theta_{(x_i,v),c_j} - \theta_\tau$
31:       $\tau \leftarrow$ next tuple in the lexicographic ordering
32:     if $x_i, c\_var \in Xd$ and $x_i$ belongs to the current stage in $X$ and $\theta_{(x_i,v),c_j} < \theta_l$ then
33:       remove $v$ from $D(x_i)$
34:       DELETION $\leftarrow$ TRUE
35:     else if $\tau \neq NIL$ then
36:       Last($(x_i, v), c_j$) $\leftarrow \tau$
37:     else
38:       remove $v$ from $D(x_i)$
39:       if $x_i \in Xs$
40:         $\zeta_i := \prod_{s=c\_var+1}^{n} \sum_{t=1}^{|D(x_s)|}$ prob$(x_s \leftarrow v_t)$
41:         if $\zeta_i < \theta_l$ then
42:           fail $\leftarrow$ true
43:           return true
44:       DELETION $\leftarrow$ TRUE
45: return DELETION
```

Fig. 5. An arc consistency algorithm for stochastic CSPs

pruning operations to account for the stochastic nature of the problem. Initially, all variable-constraint pairs (x_i, c_j), where $x_i \in var(c_j)$, are inserted in Q. Then, a preprocessing step, which implements the pruning rule described above, takes place (lines 2-12). For every decision variable x_i and any constraint c_j where x_i participates, such that the constraint includes stochastic variables after x_i in $vars(c_j)$ (this is tested in line 3), we iterate through the available values in $D(x_i)$. For each such value v we compute the maximum satisfaction $\theta_{(x_i,v),c_j}$ that the stochastic variables after x_i in $vars(c_j)$ can yield, under the assumption that x_i is given value v. This is computed as the sum of satisfaction for all sub-tuples that support $x_i \leftarrow v$ in c_j. The satisfaction of a sub-tuple is simply the product of probabilities for the values of the stochastic variables after $x_i \leftarrow v$ in the tuple (line 8). In case x_i belongs to the first stage in the problem and $\theta_{(x_i,v),c_j}$ is less than θ_l then we know that the assignment $x_i \leftarrow v$ cannot be part of a policy with satisfaction greater than θ_l and therefore v is removed from $D(x_i)$. If no domain wipeout is detected then the algorithm proceeds with the main propagation phase.

During this phase pairs (x_i, c_j) are removed from Q and function `Revise` is called to look for supports for the values of x_i in c_j. For each value $v \in D(x_i)$ we first check if $\text{Last}((x_i, v), c_j)$ is still valid. If it is we proceed with the next value. Otherwise we search c_j's tuples until one that supports v is found or there are no more tuples (lines 25-31). In the former case, $\text{Last}((x_i, v), c_j)$ is updated accordingly (line 36). In the latter case, v is removed from $D(x_i)$ (line 38). If x_i is a decision variable then $\theta_{(x_i,v),c_j}$ is reduced while the search for a support in c_j proceeds. This is done as follows: Whenever a tuple τ that was previously a support for $x_i \leftarrow v$ in c_j but is no longer one (because it is no longer valid) is encountered, $\theta_{(x_i,v),c_j}$ is reduced by θ_τ (lines 24,30). As in the preprocessing phase, θ_τ is computed as the product of probabilities for the values of the stochastic variables after $x_i \leftarrow v$ in τ. If $\theta_{(x_i,v),c_j}$ falls below θ_l, the current variable is a decision one and x_i belongs to the same stage as it, then v is removed from $D(x_i)$ (lines 32,33).

If a value of a stochastic variable is removed then we check if the remaining values in the domains of the future stochastic variables can contribute enough to the satisfaction of the problem so that the lower bound is met. This is done in a way similar to the improved function `check` of `FC`. That is, by comparing quantity $\prod_{s=c\_var+1}^{n} \sum_{t=1}^{|D(x_s)|} \text{prob}(x_s \leftarrow v_t)$ to θ_l. If it is lower then the algorithm returns failure as the threshold cannot be met. If this occurs during search then the currently explored policy should be abandoned and a new one should be tried.

The Pruning Rule for Binary Constraints. Pruning of decision variables that belong to the current decision stage can be made stronger when dealing with binary constraints. For each binary constraint c_j, where $var(c_j) = \{xd_i, xs_l\}$, and each value $v \in D(xd_i)$, we can calculate the maximum possible satisfaction of assignment $xd_i \leftarrow v$ on constraint c_j as $\theta_{(xd_i,v),xs_l} = \sum_{t=Last((xd_i,v),xs_l)}^{|D(xs_l)|}$, s.t. t and v are compatible. In this case $\text{Last}((xd_i, v), xs_l)$ is the most recently discovered value in $D(xs_l)$ that supports v. Therefore, the maximum satisfaction of assignment $xd_i \leftarrow v$ is the product of $\theta_{(xd_i,v),xs_l}$ for all constraints c_j, where $var(c_j) = \{xd_i, xs_l\}$ and xs_l is after xd_i in the variable sequence. By comparing this quantity with θ_l (lines 10 and 32), we can exploit the probabilities of future stochastic variables in a more "global" way, as in the enhancement of FC described in Section 3, and thus stronger pruning can be achieved.

Note that a similar, but more involved, enhancement is possible for non-binary constraints but in that case we have to be careful about future stochastic variables that appear in multiple constraints involving xd_i. When calculating the maximum possible satisfaction we have to make sure that the probabilities of the values of each such stochastic variable are taken into account only once. When dealing with binary constraints no such issue arises, assuming that each stochastic variable can participate in at most one constraint with xd_i.

We now analyze the time complexity of algorithm `Stochastic_AC`. We assume that the maximum domain size is D and the maximum constraint arity is k.

Proposition 1. The worst-case time complexity of `Stochastic_AC` is $O(enk^2D^{k+1})$.

Proof. The preprocessing phase of lines 2-12 is executed for decision variables. For every constraint c_j that includes a decision variable x_i and at least one later stochastic variable, we go through all values in $D(x_i)$. For each such value v, we iterate through the, at most, D^{k-1} tuples that include assignment $x_i \leftarrow v$. Assuming that the calculation of the product of probabilities requires $O(k)$ operations, the complexity of the preprocessing phase is $O(eDkD^{k-1}k)$=$O(ek^2D^k)$.

In the main propagation phase there are at most kD calls to `Revise` for any constraint c_j, one for every deletion of a value from the k variables in $var(c_j)$. In the body of `Revise` (called for $x_i \in var(c_j)$) there is a cost of $O(kD^{k-1})$ to search for supporting tuples for the values of x_i (see [1] for details). The complexity of the pruning rule for decision variables is constant. The failure detection process of lines 35-39 costs $O(nD)$ in the worst case. Therefore, the asymptotic cost of one call to `Revise` is $O(kD^{k-1}nD)$=$O(nkD^k)$. Since there can be kD calls to `Revise` for each of the e constraints, and the use of $\text{Last}((x_i, v), c_j)$ ensures that in all calls the search for support for $v \in D(x_i)$ on c_j will never check a tuple more than once, the complexity of `Stochastic_AC` is $O(enk^2D^{k+1})$ □

Since the preprocessing phase alone costs $O(ek^2D^k)$, we may want to be selective in the constraints on which the pruning rule is applied, based on properties such as arity and domain size of the variables involved.

The space complexity of the algorithm is determined by the data structures required to store $\text{Last}((x_i, v), c_j)$ and $\theta_{(x_i,v),c_j}$. Both need $O(ekD)$ space, so this is the space complexity of `Stochastic_AC`. However, this may rise to $O(enkD)$ when `Stochastic_AC` is maintained during search and no advanced mechanism is used to restore the $\text{Last}((x_i, v), c_j)$ and $\theta_{(x_i,v),c_j}$ structures upon failed instantiations and backtracks. This may be too expensive in large problems but it is always possible to reduce the memory requirements by dropping structures $\text{Last}((x_i, v), c_j)$ and $\theta_{(x_i,v),c_j}$ and reverting to a (G)AC-3-type of processing.

5 Experiments

We ran some preliminary experiments on randomly generated binary SCSPs. The best algorithm was the improved version of FC coupled with AC preprocessing. AC appears to be advantageous when used for preprocessing, but MAC is slower than FC on these problems. To generate random SCSPs we used a model with four parameters: the number of variables n, the uniform domain size d, the constraint density p (as a fraction of the total possible constraints), and the constraint tightness q (as a fraction of the total possible allowed tuples). The probabilities of the values for the stochastic variables were randomly distributed.

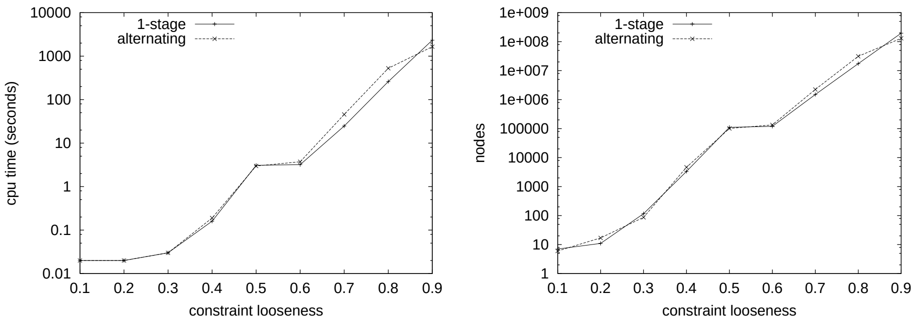


Fig. 6. AC+FC on random problems

Figure 6 demonstrates average results (over 50 instances) from SCSPs where $n = 20$, $d = 3$, $p = 0.1$, and q is varying from 0.1 to 0.9 in steps of 0.1. We show the cpu time (in seconds) and node visits required by FC with AC preprocessing to find the maximum satisfaction. The curve entitled "1-stage" corresponds to one-stage problems where 10 decision variables are followed by 10 stochastic ones, while the curve entitled "alternating" corresponds to problems where there is an alternation of decision and stochastic variables in the sequence. As we can see, both types of problems give similar results. When there are few allowed combinations of values per constraint, problems are easy as the algorithm quickly determines that most policies are not satisfying. When there are many allowed combinations of values per constraint, problems are much harder since there are many satisfying policies, and as a result, a larger part of the search tree must be searched to find the maximum satisfaction.

6 Conclusion and Future Work

We developed algorithms for SCSPs based on the exploration of the policy space. We first identified and corrected a flaw in the FC procedure proposed by Walsh. We also extended FC to better take advantage of probabilities and thus achieve stronger pruning. Then we defined AC for SCSPs and introduced an AC algorithm that can handle constraints of any arity. We ran some preliminary experiments, but further experimentation is necessary to evaluate the practical value for the proposed algorithms.

In the future we intend to further enhance the backtracking algorithms presented here, both in terms of efficiency (e.g. by adding capabilities such as backjumping), and in terms of applicability (e.g. by extending them to deal with multiple chance constraints, joint probabilities for stochastic variables and optimization problems). Also we plan to investigate alternative approaches to solving stochastic CSPs. In particular, techniques adapted from stochastic programming [8] and on-line optimization [5]. Some techniques of this kind have been already successfully developed in [7].

Acknowledgements

This work has been supported by GR Pythagoras grant number 1349 under the Operational Program for Education and Initial Training.

References

1. C. Bessière, J.C. Régin, R. Yap, and Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
2. L. Bordeaux, M. Cadoli, and T. Mancini. CSP Properties for Quantified Constraints: Definitions and Complexity. In *Proceedings of AAAI-2005*, pages 360–365, 2005.
3. Lucas Bordeaux. Boolean and interval propagation for quantified constraints. In *Proceedings of the CP-05 Workshop on Quantification in Constraint Programming*, pages 16–30, 2005.
4. D. Dubois, H. Fargier, and H. Prade. Possibility Theory in Constraint Satisfaction Problems: Handling Priority, Preference and Uncertainty. *Applied Intelligence*, 6(4):287–309, 1996.
5. A. Fiat and G.J. Woeginger. *Online Algorithms*, volume 1442. LNCS, 1997.
6. M.L. Littman, S.M. Majercik, and T. Pitassi. Stochastic Boolean Satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2000.
7. S. Manandhar, A. Tarim, and T. Walsh. Scenario-based Stochastic Constraint Programming. In *Proceedings of IJCAI-03*, pages 257–262, 2003.
8. A. Ruszczynski and A. Shapiro. *Stochastic Programming, Handbooks in OR/MS*, volume 10. Elsevier Science, 2003.
9. G. Verfaillie and N. Jussien. Constraint Solving in Uncertain and Dynamic Environments: A Survey. *Constraints*, 10(3):253–281, 2005.
10. T. Walsh. Stochastic Constraint Programming. In *Proceedings of ECAI-2002*, pages 111–115, 2002.

Graph Properties Based Filtering

Nicolas Beldiceanu[1], Mats Carlsson[2], Sophie Demassey[1], and Thierry Petit[1]

[1] École des Mines de Nantes, LINA FRE CNRS 2729, FR-44307 Nantes, France
{Nicolas.Beldiceanu, Sophie.Demassey, Thierry.Petit}@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se

Abstract. This article presents a generic filtering scheme, based on the graph description of global constraints. This description is defined by a network of binary constraints and a list of elementary graph properties: each solution of the global constraint corresponds to a subgraph of the initial network, retaining only the satisfied binary constraints, and which fulfills all the graph properties. The graph-based filtering identifies the arcs of the network that belong or not to the solution subgraphs. The objective is to build, besides a catalog of global constraints, also a list of systematic filtering rules based on a limited set of graph properties. We illustrate this principle on some common graph properties and provide computational experiments of the effective filtering on the `group` constraint.

1 Introduction

The global constraint catalog [3] provides the description of hundreds of global constraints in terms of graph properties: The solutions of a global constraint are identified to the subgraphs of one initial digraph sharing several graph properties. Existing graph properties use a small set of graph parameters such as the number of vertices, or arcs, or the number of connected components(cc).The most common graph parameters were considered in [6]. It showed how to estimate, from the initial digraph, the lower and upper values of a parameter in the possible solution subgraphs. Those bounds supply necessary conditions of feasibility for almost any global constraint.

This article goes one step further by introducing systematic filtering rules for those global constraints. The initial digraph describing a global constraint is indeed a network of constraints on pairs of variables. To each complete instantiation of the variables corresponds a final subgraph obtained by removing from the initial digraph all the arcs (i.e., the binary constraints) that are not satisfied. Since solution(s) of the global constraint correspond to final subgraphs fulfilling a given set of graph properties, filtering consists in identifying and dropping elements of the initial digraph that do not belong to such subgraphs, or to force those elements that belong to any solution subgraphs.

A first way to achieve this identification might be to use shaving [11]. That is, fix the status of an arc or a vertex, and check if it leads to a contradiction. Since this is very costly in practice, we present in this article another way to proceed. The filtering rules proposed thereafter apply whenever a graph parameter is set to one of its bounds.

Last, the global constraints can be partitioned wrt. the class that their associated final digraphs belongs to. Taking into account a given graph class leads to a better estimation of the graph parameter bounds and then a more effective filtering.

The article is organized as follows: Section 2 recalls the graph-based description of global constraints and introduces a corresponding reformulation. Section 3 sets up a list of notations in order to formalize the systematic graph-based filtering. Section 4 presents the filtering rules related to the bounds of some graph parameters. Section 5 shows how the graph-based filtering relates to existing ad-hoc filtering for some global constraints. Section 6 illustrates how refining the filtering according to a given graph class and provides computational results on the `group` constraint, which belongs to the `path_with_loops` graph class.

2 Graph-Based Description of Global Constraints

2.1 Graph-Based Description

Let $C(V_1, \ldots, V_p, x_1, \ldots, x_n)$ be a global constraint with domain variables[1] $V_1, \ldots,$ V_p, and domain or set variables[2] $x_1, \ldots, x_n$. When it exists, a *graph-based description* of C is given by one (or several) network(s) $G_{\mathcal{R}} = (X, E_{\mathcal{R}})$ of binary constraints over $X = \{x_1, \ldots, x_n\}$ in association with a set $\mathcal{GP}_{\mathcal{R}} = \{\mathcal{P}_l \; op_l \; V_l \mid l = 1, \ldots, p\}$ of graph properties and, optionally, a graph class $c_{\mathcal{R}}$, where:

- The constraints defining the digraph $G_{\mathcal{R}} = (X, E_{\mathcal{R}})$ share the same semantic (typically it is an equality, an inequality or a disequality). Let $x_j \mathcal{R} x_k$ denote the so-called *arc constraint* between the ordered pair of variables $(x_j, x_k) \in E_{\mathcal{R}}$ (or the unary constraint if $j = k$).
- $\mathcal{P}_l \; op_l \; V_l$ expresses a graph property comparing the value of a graph parameter $\mathcal{P}_l$ to the value of variable V_l. The comparison operator op_l is either $\geq$, $\leq$, $=$, or $\neq$. Among the most usual graph parameters $\mathcal{P}_l$, let $\mathbf{NARC}$ denote the number of arcs of a graph, $\mathbf{NVERTEX}$ the number of vertices, $\mathbf{NCC}$ the number of cc, $\mathbf{MIN\_NCC}$ and $\mathbf{MAX\_NCC}$ the numbers of vertices of the smallest and the largest cc respectively.
- $c_{\mathcal{R}}$ corresponds to recurring graph classes that show up for different global constraints. For example, we consider graphs in the classes `acyclic`, `symmetric`, `bipartite`.

$G_{\mathcal{R}}$ is called the *initial digraph*. When all variables x are instantiated, the subgraph of $G_{\mathcal{R}}$, obtained by removing all arcs corresponding to unsatisfied constraints $x_j \mathcal{R} x_k$ as well as all vertices becoming isolated, is called a *final digraph* (associated to the instantiation) and is denoted by $G_f = (X_f, E_f)$.

The relation between C and its graph-based description is stated as follows:

Definition 1. *A complete assignment of variables $V_1, \ldots, V_p, x_1, \ldots, x_n$ is a solution of C iff the final digraph associated to the assignment of $x_1, \ldots, x_n$, satisfies all graph properties $\mathcal{P}_l \; op_l \; V_l$ in $\mathcal{GP}_{\mathcal{R}}$ and belongs to the graph class $c_{\mathcal{R}}$.*

[1] A *domain variable* D is a variable ranging over a finite set of integers $\mathrm{dom}(D)$. $\min(D)$ and $\max(D)$ respectively denote the *minimum* and *maximum* values in $\mathrm{dom}(D)$.

[2] A *set variable* S is a variable that will be assigned to a finite set of integer values. Its domain is specified by its *lower bound* $\underline{S}$, and its *upper bound* $\overline{S}$, and contains all sets that contain $\underline{S}$ and are contained in $\overline{S}$.

Example 1. Consider the `proper_forest(NTREE, VER)` constraint introduced in [5]. It receives a domain variable `NTREE` and a digraph G described by a collection of n vertices `VER`: each vertex is labelled by an integer between 1 and n and is represented by a set variable whose lower and upper bounds are the sets of the labels of respectively its *mandatory neighbors* and its *mandatory or potential neighbors* in G. This constraint partitions the vertices of G into a set of vertex-disjoint proper trees (i.e., trees with at least two vertices each).

Part (A) of Figure 1 illustrates such a digraph G, where solid arrows depict mandatory arcs and dashed arrows depict potential arcs. Part (B) of the figure shows a possible solution on this digraph with three proper trees. In the graph-based description of `proper_forest`, the initial digraph corresponds exactly to G and has no loop. Any final digraph G_f contains all the mandatory arcs of G and belongs to the `symmetric` graph class.[3] Moreover G_f has to fulfill the following graph properties: $\mathbf{NVERTEX} = n$ (since it is a vertex partitioning problem, G_f contains all the vertices of G), $\mathbf{NARC} = 2 \cdot (n - \texttt{NTREE})$ (2 since G_f is symmetric, and $n - \texttt{NTREE}$ since we have `NTREE` acyclic connected digraphs) and $\mathbf{NCC} = \texttt{NTREE}$.

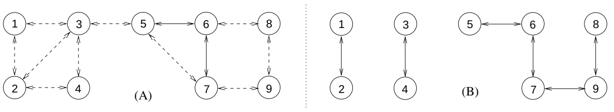


Fig. 1. (A) A digraph and (B) a solution with three proper trees for the `proper_forest` constraint

2.2 Graph-Based Reformulation

According to Definition 1, any global constraint $C(V_1, \ldots, V_p, x_1, \ldots, x_n)$ holding a graph-based description can be reformulated as follows:

Proposition 1. *Define additional variables attached to each constraint network $G_{\mathcal{R}} = (X, E_{\mathcal{R}})$: to each vertex x_j and to each arc e_{jk} of $G_{\mathcal{R}}$ correspond* 0-1 *variables respectively denoted $vertex_j$ and arc_{jk}. Vertex and Arc denote these sets of variables. Last, let G_f denote the subgraph of $G_{\mathcal{R}}$, whose vertices and arcs correspond to the variables $vertex_j$ and arc_{jk} set to 1. Then constraint C holds iff the following constraints hold:*

$$arc_{jk} = 1 \Leftrightarrow x_j \mathcal{R} x_k, \quad \forall e_{jk} \in E_{\mathcal{R}} \tag{1}$$

$$vertex_j = \min(1, \textstyle\sum_{\{k \mid e_{jk} \in E_{\mathcal{R}}\}} arc_{jk} + \sum_{\{k \mid e_{kj} \in E_{\mathcal{R}}\}} arc_{kj}), \quad \forall x_j \in X \tag{2}$$

$$ctr_{\mathcal{P}_l}(Vertex, Arc, P_l), \quad \forall (\mathcal{P}_l \; op_l \; V_l) \in \mathcal{GP}_{\mathcal{R}} \tag{3}$$

Constraint (3) *is satisfied when P_l is equal to the value of the corresponding parameter $\mathcal{P}_l$ in G_f.*

$$P_l \; op \; V_l, \quad \forall (\mathcal{P}_l \; op_l \; V_l) \in \mathcal{GP}_{\mathcal{R}} \tag{4}$$

$$ctr_{c_{\mathcal{R}}}(Vertex, Arc) \tag{5}$$

Graph-class constraint (5) *is satisfied if G_f belongs to the graph class $c_{\mathcal{R}}$.*

[3] A digraph is *symmetric* iff, if there is an arc from u to v, there is also an arc from v to u.

Example 2. Consider again the `proper_forest` constraint previously introduced. Since its final digraph is symmetric and does not contain isolated vertices, the graph-class constraint (5) is the conjunction of the following constraints: $arc_{jk} = arc_{kj}$ for each arc e_{jk} and $vertex_j = \min(1, 2 \cdot \sum_{\{k \mid e_{jk} \in E_{\mathcal{R}}\}} arc_{jk})$ for each vertex x_j in $G_{\mathcal{R}}$.

Filtering domains of variables V and x according to C can be achieved by enforcing alternatively, and for each constraint network $G_{\mathcal{R}}$, the five sets of constraints of this reformulation. Enforcing constraints (1), (2), (4) and (5) is mostly trivial since these constraints are elementary arithmetic constraints. The generic graph-based filtering mainly lies then on maintaining consistency according to constraints (3), from the arc and $vertex$ variables to the bounds of the graph parameter variables P_l, and conversely. In [6] it was presented, for some usual graph parameters, how to estimate their minimal ($\underline{P_l}$) and maximal ($\overline{P_l}$) values in the final digraphs G_f, given the current state of the arcs and vertices of $G_{\mathcal{R}}$. Section 4 shows how in turn, the status of some arcs and vertices can be determined according to a graph parameter variable when it is set to one of its extreme values (i.e. $\text{dom}(P_l) = \{\underline{P_l}\}$ or $\text{dom}(P_l) = \{\overline{P_l}\}$).

Hence, the approach relies on identifying the possible final digraphs in $G_{\mathcal{R}}$ that minimize or maximize a given graph parameter. Any final digraph contains (resp. does not contain) the arcs and vertices corresponding to arc and $vertex$ variables fixed to 1 (resp. to 0). Since it has no isolated vertices, we assume that the *normalization constraints* (2) are enforced before estimating a graph parameter. Section 6 shows how refining this estimation when the final digraphs must belong to a given graph class, by also first enforcing constraints (5).

Since the proposed reformulation allows to model a lot of global constraints for which enforcing AC is known to be impossible (e.g. `nvalue`) we cannot expect to get AC in general. Even with a complete characterization of all feasible values of a graph parameter and of all corresponding unfeasible arcs (arcs that do not belong to final digraphs satisfying a parameter value), we cannot enforce AC in general because of the inter-dependency of constraints (1) : the arc variables are not independent of each other.

3 Notations for a Systematic Filtering

As for the graph-based description of any global constraint, we aim at providing a catalog of generic filtering rules related to the bounds of graph parameters. In order to formalize this, we first need to introduce a number of notations.

Let $G_{\mathcal{R}}$ be an initial digraph associated to the graph-based description of a global constraint. The current domains of variables arc and $vertex$ of the reformulation correspond to a unique partitioning of the arc and vertex sets of $G_{\mathcal{R}}$, denoted as follows:

Notation 1. *A vertex x_j or an arc e_{jk} of $G_{\mathcal{R}}$ is either* true *(T),* false *(F), or* undetermined *(U) whether the corresponding variable $vertex_j$ or arc_{jk} is fixed to* 1*, fixed to* 0 *or yet unfixed (with domain $\{0,1\}$). This leads to the partitioning of the vertex set of $G_{\mathcal{R}}$ into $X_T \dot\cup X_F \dot\cup X_U$ and to the partitioning of the edge set of $G_{\mathcal{R}}$ into $E_T \dot\cup E_F \dot\cup E_U$. For two distinct elements Q and R in $\{T, U, F\}$, let X_{QR} denote the vertex subset $X_Q \dot\cup X_R$, and E_{QR} denote the arc subset $E_Q \dot\cup E_R$.*

Once the normalization constraints are enforced, subgraph (X_T, E_T) is well defined and is included in any final digraph. (X_{TU}, E_{TU}) is also a subgraph of $G_{\mathcal{R}}$, called the *intermediate digraph*, and any final digraph is derived from this by turning each U-arc and U-vertex into T or F.[4] We aim at identifying the final digraphs in which a graph parameter P reaches its lower value $\underline{P}$ or its upper value $\overline{P}$. An estimated bound is said to be *sharp* if for any intermediate digraph, there exists at least one final digraph where the parameter takes this value. To estimate these bounds, we deal with different digraphs derived from the intermediate digraph:

Notation 2. *For any subsets Q, R and S of $\{T, U, F\}$, X_Q and X_S are vertex subsets and E_R is an arc subset of the initial digraph, and:*

- *$X_{Q,R}$ (resp. $X_{Q,\neg R}$) denotes the set of vertices in X_Q that are extremities of at least one arc (resp. no arc) in E_R.*
- *$X_{Q,R,S}$ (resp. $X_{Q,R,\neg S}$) denotes the set of vertices in $X_{Q,R}$ that are linked to at least one vertex (resp. to no vertex) in X_S by an arc in E_R.*
- *$X_{Q,\neg R,S}$ (resp. $X_{Q,\neg R,\neg S}$) denotes the set of vertices in $X_{Q,\neg R}$ that are linked to at least one vertex (resp. to no vertex) in X_S by an arc in E_{TU}.*
- *$E_{R,Q}$ is the set of arcs in E_R that are incident on at least one vertex in X_Q.*
- *$E_{R,Q,S}$ is the set of arcs in E_R that are incident on one vertex in X_Q and on one vertex in X_S.*

Notation 3. *Given a digraph G and subsets $\mathcal{X}$ of vertices and $\mathcal{E}$ of arcs:*

- *$\overrightarrow{G}(\mathcal{X}, \mathcal{E})$ denotes the induced subgraph of G containing all vertices in $\mathcal{X}$ and all arcs of $\mathcal{E}$ having their two extremities in $\mathcal{X}$.*
- *$\overleftrightarrow{G}(\mathcal{X}, \mathcal{E})$ denotes the corresponding undirected graph: to one edge (u, v) corresponds at least one arc (or loop) (u, v) or (v, u) in $\overrightarrow{G}(\mathcal{X}, \mathcal{E})$.*
- *$cc(G)$ denotes the set of cc of G and $cc_{[cond]}(G)$ denotes the subset of cc that satisfy a given condition cond.*

Notation 4. *$\overleftrightarrow{G}_{rem}$ denotes the (undirected) induced subgraph of $\overleftrightarrow{G}(X_{TU}, E_U)$ obtained by removing all vertices present in $cc_{[|E_T|\geq 1]}(\overrightarrow{G}(X_T, E_T))$ and then by removing all vertices becoming isolated in the remaining undirected graph.*

Last, we recall some graph theoretic terms:

Definition 2. – *A* matching *of an undirected graph G is a subset of edges, excluding loops, such that no two edges have a vertex in common. A* maximum matching *is a matching of maximum cardinality. $\mu(G)$ denotes the* cardinality of a maximum matching *of G. If loops are allowed in the matching then it is called a l-*matching *and the maximum cardinality of an l-matching in G is denoted by $\mu_l(G)$.*
- *Given a bipartite graph $G((Y, Z), E)$, a* hitting set *of $G((Y, Z), E)$ is a subset Z' of Z such that for any vertex $y \in Y$ there exists an edge in E connecting y to a vertex in Z'. $h(G)$ denotes the* cardinality of a minimum hitting set *of G.*

[4] In the context of CP(Graph) [9], these two digraphs respectively correspond to the lower and upper bounds of a graph variable. Note that, as a consequence, our approach can easily be adapted to providing a generic filtering for CP(Graph).

4 Filtering from Bounds of Graph Parameters

This section illustrates on the examples of $\mathbf{NVERTEX}$ and $\mathbf{NCC}$, how to filter according to a graph-parameter constraint $ctr_{\mathcal{P}_l}(Vertex, Arc, P_l)$ (Constraint (3) of Proposition 1). Table 1 first recalls the generic formula to estimate the bounds of these two parameters according to the current instantiation of $Vertex$ and Arc. These results were previously given in [6]. All these bounds are sharp. Then we present a reverse filtering when $\text{dom}(P) = \{\underline{P}\}$ or $\text{dom}(P) = \{\overline{P}\}$. The next rules allow to determine the status of U-vertices and U-arcs of the intermediate digraph whenever any final digraph must contain exactly the minimal or the maximal number of vertices or of cc.

Table 1. Bounds of the different graph parameters

Graph parameters	Bound	Graph parameters	Bound
$\underline{\mathbf{NVERTEX}}$	$\lvert X_T\rvert + h(\overleftrightarrow{G}((X_{T,\neg T,\neg T}, X_{U,\neg T,T}), E_{U,T}))$	$\underline{\mathbf{NCC}}$	$\lvert cc_{[\lvert X_T\rvert \geq 1]}(\overrightarrow{G}(X_{TU}, E_{TU}))\rvert$
$\overline{\mathbf{NVERTEX}}$	$\lvert X_{TU}\rvert$	$\overline{\mathbf{NCC}}$	$\lvert cc_{[\lvert E_T\rvert \geq 1]}(\overrightarrow{G}(X_T, E_T))\rvert + \mu_l(\overleftrightarrow{G}_{rem})$

4.1 Filtering from $\underline{\mathbf{NVERTEX}}$

$\underline{\mathbf{NVERTEX}}$ is equal to the current number of T-vertices, $|X_T|$, plus the minimum number of U-vertices that should be turned into T-vertices to avoid isolated T-vertices. This estimation is based on the computation of the cardinality of a minimum hitting set.

Theorem 1. *If* $\text{dom}(\mathbf{NVERTEX}) = \{\underline{\mathbf{NVERTEX}}\}$ *then*

1. *Any U-vertex in $X_{U,\neg T,\neg T}$ is turned into an F-vertex.*
2. *Any U-vertex in $X_{U,\neg T,T}$ that does not belong to any minimum hitting set of $\overleftrightarrow{G}((X_{T,\neg T,\neg T}, X_{U,\neg T,T}), E_{U,T})$ is turned into an F-vertex (notably if it is not linked to any vertex in $X_{T,\neg T,\neg T}$).*
3. *Any U-vertex in $X_{U,\neg T,T}$ that belongs to all minimum hitting sets of $\overleftrightarrow{G}((X_{T,\neg T,\neg T}, X_{U,\neg T,T}), E_{U,T})$ is turned into a T-vertex.*
4. *For all edges $e = (u, v)$ such that $u \in X_{T,\neg T,\neg T}$ and $v \in X_{U,\neg T,T}$, if all minimum hitting sets are such that v is the only vertex that can be associated with u, and if a unique arc corresponds to e in $\overrightarrow{G}((X_{T,\neg T,\neg T}, X_{U,\neg T,T}), E_{U,T})$, then this arc can be turned into a T-arc.*

Example 3. Part (A) of Figure 2 illustrates Theorem 1 according to the hypothesis that the final digraph should not contain more than 7 vertices.[5] The U-vertices 1 and 10 are turned into F-vertices according to Item 1. Since they do not belong to any minimum hitting set, the U-vertices 4, 5 and 12 are turned into F-vertices according to Item 2. Since the U-vertex 9 belongs to all minimum hitting sets, it is turned into a T-vertex according to Item 3. Last, the U-arc $(8, 9)$ is turned into a T-arc according to Item 4. Part (B) depicts the corresponding graph $\overleftrightarrow{G}((X_{T,\neg T,\neg T}, X_{U,\neg T,T}), E_{U,T})$ used for computing the cardinality of a minimum hitting set (represented by thick lines).

[5] As in Figure 1, solid arrows/circles depict T-arcs/vertices and dashed arrows/circles depict U-arcs/vertices in the intermediate digraph.

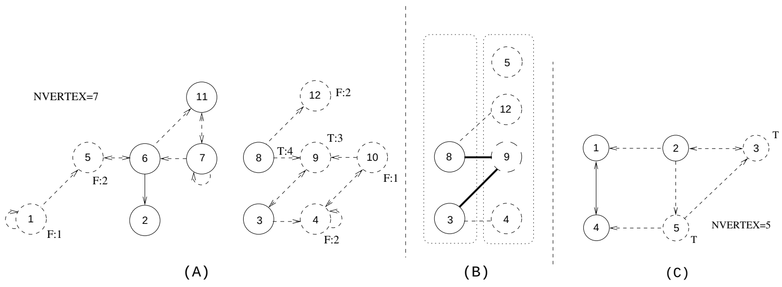


Fig. 2. Filtering according to $\mathbf{NVERTEX}$: Illustration of Theorems 1 (A, B) and 2 (C)

Since Theorem 1 involves computing the cardinality of a minimum hitting set, which is exponential, we provide a weaker form of Theorem 1.

Corollary 1. *If* $\mathrm{dom}(\mathbf{NVERTEX}) = \{|X_T|\}$ *then any U-vertex is turned into a F-vertex.*

4.2 Filtering from $\overline{\mathbf{NVERTEX}}$

$\overline{\mathbf{NVERTEX}}$ corresponds to final digraphs derived $\overleftrightarrow{G}(X_{TU}, E_{TU})$ by turning all U-vertices into T.

Theorem 2. *If* $\mathrm{dom}(\mathbf{NVERTEX}) = \{\overline{\mathbf{NVERTEX}}\}$ *then any U-vertex of $\overleftrightarrow{G}(X_{TU}, E_{TU})$ is turned into a T-vertex.*

Example 4. Part (C) of Figure 2 illustrates Theorem 2 according to the hypothesis that the final digraph should contain at least 5 vertices. Theorem 2 turns all U-vertices into T-vertices.

4.3 Filtering from $\underline{\mathbf{NCC}}$

The minimal number of cc in any final digraph is equal to the number of cc in the intermediate digraph that contain at least one T-vertex.

Theorem 3. *If* $\mathrm{dom}(\mathbf{NCC}) = \{\underline{\mathbf{NCC}}\}$ *then*

1. *Any U-arc or U-vertex of any cc in $|cc_{[|X_T|=0]}(\overrightarrow{G}(X_{TU}, E_{TU}))|$ is turned into an F-arc or an F-vertex.*
2. *Any U-vertex that is an articulation point of $\overleftrightarrow{G}(X_{TU}, E_{TU})$ such that its removal disconnects two T-vertices[6] is turned into a T-vertex.*
3. *For any edge e of $\overleftrightarrow{G}(X_{TU}, E_{TU})$ that is a bridge such that its removal disconnects two T-vertices, if a unique U-arc in $\overrightarrow{G}(X_{TU}, E_{TU})$ corresponds to e then this U-arc is turned into a T-arc.*

[6] The two T-vertices do not belong any more to the same cc.

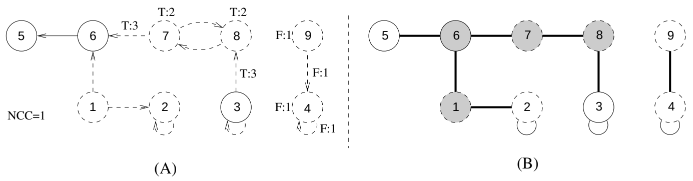


Fig. 3. Filtering according to $\underline{\mathbf{NCC}}$: Illustration of Theorem 3

Example 5. Part (A) of Figure 3 illustrates Theorem 3 according to the hypothesis that the final digraph should contain no more than one cc. Part (B) represents the undirected graph $\overleftrightarrow{G}(X_{TU}, E_{TU})$, where grey vertices correspond to articulation points and thick lines correspond to bridges. Since $\overrightarrow{G}(X_{TU}, E_{TU})$ contains one single cc involving T-vertices, the precondition of Theorem 3 holds and we get the following filtering: Since the cc of $\overrightarrow{G}(X_{TU}, E_{TU})$ with vertices $\{4, 9\}$ belongs to $cc_{[|X_T|=0]}(\overrightarrow{G}(X_{TU}, E_{TU}))$, then, from Item 1, U-vertices 4 and 9 as well as U-arcs $(4, 4)$ and $(9, 4)$ are respectively turned into F-vertices and F-arcs. From Item 2, the two U-vertices 7 and 8, which are articulation points of $\overleftrightarrow{G}(X_{TU}, E_{TU})$ belonging to an elementary chain between two T-vertices (3 and 6), are turned into T-vertices. From Item 3, among the 3 bridges of $\overleftrightarrow{G}(X_{TU}, E_{TU})$ belonging to an elementary chain between two T-vertices (3 and 6), $(3, 8)$ and $(7, 6)$ are turned into T-arcs since their respective counterparts $(8, 3)$ and $(6, 7)$ do not belong to $\overrightarrow{G}(X_{TU}, E_{TU})$.

4.4 Filtering from $\overline{\mathbf{NCC}}$

$\overline{\mathbf{NCC}}$ is equal to the current number of cc of $\overrightarrow{G}(X_T, E_T)$ containing at least one T-arc (that is, $|cc_{[|E_T|\geq 1]}(\overrightarrow{G}(X_T, E_T))|$) plus the cardinality of a maximum matching $\mu_l(\overleftrightarrow{G}_{rem})$, which is the maximum possible number of new cc that could be present in a final digraph stemming from $\overrightarrow{G}(X_{TU}, E_{TU})$, in addition to $|cc_{[|E_T|\geq 1]}(\overrightarrow{G}(X_T, E_T))|$. Then all U-arcs (and U-vertices) that may reduce the number of cc if they would belong to the final digraph have to be turned into F-arcs (and F-vertices).

Theorem 4. *If* $\mathrm{dom}(\mathbf{NCC}) = \{\overline{\mathbf{NCC}}\}$ *then*

1. *Any U-arc of $\overrightarrow{G}(X_T, E_{TU})$ joining two T-vertices belonging to two distinct cc in $cc_{[|E_T|\geq 1]}(\overrightarrow{G}(X_T, E_T))$ is turned into an F-arc.*
2. *For any edge in $\overleftrightarrow{G}_{rem}$ that does not belong to any maximum l-matching of $\overleftrightarrow{G}_{rem}$, the corresponding U-arc(s) are turned into F-arcs.*
3. *Any U-arc $e = (u, v)$ such that u belongs to a cc in $cc_{[|E_T|\geq 1]}(\overrightarrow{G}(X_T, E_T))$ and v is saturated in every maximum l-matchings of $\overleftrightarrow{G}_{rem}$ is turned into an F-arc.*
4. *Any U-vertex of $\overleftrightarrow{G}_{rem}$ belonging to all maximum l-matchings of $\overleftrightarrow{G}_{rem}$ is turned into a T-vertex.*
5. *For all edges e belonging to all maximum l-matchings of $\overleftrightarrow{G}_{rem}$, if a unique U-arc in $\overrightarrow{G}(X_{TU}, E_U)$ corresponds to e then this arc is turned into a T-arc.*

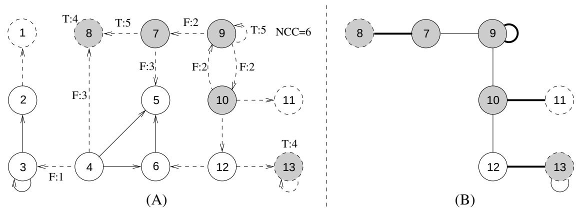


Fig. 4. Filtering according to $\overline{\mathbf{NCC}}$: Illustration of Theorem 4

6. *Any U-vertex of $\overleftrightarrow{G}_{rem}$ that does not belong to any maximum l-matching of $\overleftrightarrow{G}_{rem}$ is turned into an F-vertex.*

Example 6. Part (A) of Figure 4 illustrates Theorem 4 according to the hypothesis that the final digraph should contain at least 6 cc. $cc_{[|E_T|\geq 1]}(\overrightarrow{G}(X_T, E_T))$ consists of the following two cc, respectively corresponding to the sets of vertices $\{2, 3\}$ and $\{4, 5, 6\}$. Part (B) illustrates the corresponding undirected graph $\overleftrightarrow{G}_{rem}$, where thick lines correspond to a maximum l-matching of cardinality 4, and grey vertices are vertices that are saturated in all maximum l-matchings. Since the precondition $\mathbf{NCC} = |cc_{[|E_T|\geq 1]}(\overrightarrow{G}(X_T, E_T))| + \mu_l(\overleftrightarrow{G}_{rem}) = 6$ of Theorem 4 holds, Items 1, 2 and 3 respectively turn the U-arcs of $\{(4, 3)\}$, of $\{(9, 7), (9, 10), (10, 9)\}$ and of $\{(4, 8), (7, 5)\}$ into F-arcs. Item 4 turns the U-vertices $\{8, 13\}$ into T-vertices. Finally, Item 5 turns the U-arcs $\{(7, 8), (9, 9)\}$ into T-arcs.

4.5 Complexity Results

Table 2 provides complexity results for the triggering conditions as well as for each item of the theorems that were previously introduced. Most of these items correspond

Table 2. Complexity of each theorem. m and n respectively denote the number of arcs and the number of vertices in the intermediate digraph.

Theorem Parts	Complexity	Graph Related Problems
Theorem 1		
• Triggering	NP-hard [10]	*cardinality of a minimum hitting set*
• Item 1	$O(m)$	*iterating through the arcs*
• Items 2,4	NP-hard ?	*identifying vertices that do not belong to any minimum hitting set*
• Item 3	NP-hard ?	*identifying vertices that belong to every minimum hitting set*
Corollary 1		
• Triggering	$O(n)$	*iterating through the vertices*
• Item 1	$O(n)$	*iterating through the vertices*
Theorem 2		
• Triggering	$O(n)$	*iterating through the vertices*
• Item 1	$O(n)$	*iterating through the vertices*
Theorem 3		
• Triggering	$O(n)$	*iterating through the vertices*
• Item 1	$O(m)$	*iterating through the arcs*
• Items 2,3	$O(m)$	*depth first search*
Theorem 4		
• Triggering	$O(m\sqrt{n})$	*maximum cardinality matching* [12]
• Item 1	$O(m)$	*computing the cc*
• Items 2,5,6	$O(m \cdot n)$	*identifying edges that do not belong to any maximum cardinality matching* [14]
• Items 3,4	$O(m)$	*identifying vertices that are saturated in every maximum cardinality matching* [5]

directly to an existing graph problem that we mention in the third column of the table. The complexity stated for each item of a theorem assumes that the corresponding triggering condition was already computed: for instance, assuming that a maximum cardinality matching was already computed, identifying vertices that are saturated in every maximum cardinality matching is linear in the number of edges of the graph [5].

5 Relating to Ad-Hoc Filtering

At this point one may wonder whether our generic graph-based filtering is not too theoretical in order to have any practical interest. We show that we can obtain rational reconstructions of several ad-hoc algorithms that were constructed for specific global constraints. For this purpose, we consider the `proper_forest` constraint introduced in Example 1. A specialized filtering algorithm was recently proposed in [5].[7] It is made up from the following steps:

1. Renormalize $\overrightarrow{G}(X_{TU}, E_{TU})$ according to the fact that the final digraph has to be symmetric.
2. Check the feasibility of the `proper_forest` constraint:
   (a) The intermediate digraph has no isolated vertex.
   (b) There is no cycle made up from T-arcs.
   (c) NTREE has at least one value in [MINTREE, MAXTREE] where MINTREE is the number of cc of the intermediate digraph and MAXTREE is the number of cc of $\overrightarrow{G}(X_T, E_T)$ plus the cardinality of a maximum cardinality matching in the subgraph induced by the vertices that are not linked to any T-vertices.
3. Every U-arc that would create a cycle of T-vertices, is turned into an F-arc.
4. The minimum and maximum values of NTREE are respectively adjusted to MINTREE and MAXTREE.
5. When NTREE is fixed to MINTREE all U-arcs corresponding to bridges of $\overrightarrow{G}(X_{TU}, E_{TU})$ are turned into T-arcs.
6. When NTREE is fixed to MAXTREE each U-arc (u, v) satisfying one of the following conditions is turned into an F-arc:
   (a) Both u and v belong to two distinct cc of $\overrightarrow{G}(X_T, E_T)$ involving more than one vertex.
   (b) (u, v) does not belong to any maximum matching in the subgraph induced by the vertices that are not extremities of any T-arc.
   (c) u is the extremity of a T-arc and v is saturated in every maximum matching in the subgraph induced by the vertices that are not linked to any T-vertices.
7. Every U-arc involving a source or a sink is turned into a T-arc.

By considering the generic graph-based reformulation of Proposition 1 on the graph property NTREE $= \mathbf{NCC}$ we retrieve almost all the steps of the previous algorithm (except steps 2(b) and 3, which come from the invariant $\mathbf{NARC} = 2 \cdot (n - \mathbf{NCC})$ linking the two graph parameters $\mathbf{NARC}$ and $\mathbf{NCC}$):

– Item 1 corresponds to posting the graph-class constraints (5) (in the context of `proper_forest`, the final digraph has to be symmetric).
– Item 2(a) corresponds to posting the normalization constraints (2).

[7] In [1] we retrieve the filtering algorithm of the among constraint proposed by Bessière et al. in [7].

- Item 2(c) corresponds to checking the graph property constraint $\mathbf{NCC} = \texttt{NTREE}$ (4). MINTREE and MAXTREE respectively correspond to the general lower and upper bounds of $\mathbf{NCC}$ given in Table 1 and deduced from constraint (3).
- Item 4 is the propagation induced by the graph property constraint (4).
- Item 5 and Item 6 correspond to the propagation of constraint (3) on the graph parameter $\mathbf{NCC}$: Item 5 is the specialization of Theorem 3, namely its third item (since the first two items of Theorem 3 are irrelevant because $\overrightarrow{G}(X_{TU}, E_{TU})$ does not contain any U-vertex). Item 6 corresponds to Theorem 4.
- Item 7 corresponds to the propagation of graph-class constraint (5), which avoids creating isolated vertices in the symmetric graph (see Example 2).

6 Specializing the Filtering According to Graph Classes

Quite often the final digraph of a global constraint has a regular structure inherited from the initial digraph or stemming from some property of the arc constraint. This translates as extra elementary constraints, the *graph-class constraints* (5), in the graph-based reformulation of the global constraint. Enforcing these constraints before evaluating the graph parameter bounds in the intermediate digraph allows to refine the bound formula of Table 1 and the bound-based filtering (Section 4), both in terms of sharpness and of algorithmic complexity. This section illustrates this principle on the path_with_loops graph class for the four graph parameters $\mathbf{NVERTEX}$, $\mathbf{NCC}$, $\mathbf{MIN\_NCC}$, and $\mathbf{MAX\_NCC}$. The filtering is then experimentally evaluated on the example of the group constraint, which belongs to the path_with_loops graph class and which involves these four parameters in its graph-based description.

6.1 The path_with_loops Graph Class

The path_with_loops graph class groups together global constraints with the following graph-based description:

- The initial digraph uses the $PATH$ and the $LOOP$ arc generators. It consists of a sequence of vertices $X = \{x_1, \ldots, x_n\}$ with one arc $(x_j, x_{j+1}) \in E_{\mathcal{R}}$, $j \in \{1, \ldots, n-1\}$, for each pair of consecutive vertices, and one loop $(x_j, x_j) \in E_{\mathcal{R}}$, $j \in \{1, \ldots, n\}$, on each vertex (see Part (A) of Figure 5).
- In any final digraph, each remaining vertex has its loop and two consecutive vertices remain linked by an arc (see Part (B) of Figure 5). These conditions correspond to the following graph-class constraints in the reformulation of Proposition 1:

$$vertex_j = arc_{j,j} \qquad (6) \qquad\qquad \min(vertex_j, vertex_{j+1}) = arc_{j,j+1} \qquad (7)$$

Among the global constraints belonging to the path_with_loops graph class, the catalog mentions for example group [8] and stretch [13]. Such constraints enforce sequences of variables to satisfy given patterns.

Example 7. Consider the group (NGROUP, MIN_SIZE, MAX_SIZE, MIN_DIST, MAX_DIST, NVAL, VARIABLES, VALUES) constraint, where the first six parameters are domain variables, while VARIABLES is a sequence of n domain variables and VALUES a finite set of integers.

Let $x_i, x_{i+1}, \ldots, x_j$ $(1 \leq i \leq j \leq n)$ be consecutive variables of the sequence VARIABLES such that all the following conditions simultaneously apply: (1) all variables $x_i, \ldots, x_j$ take their value in the set of values VALUES, (2) either $i = 1$, or x_{i-1} does not take a value in VALUES, (3) either $j = n$, or x_{j+1} does not take a value in VALUES. We call such a set of variables a *group*. The constraint group is fulfilled if all the following conditions hold: (i) there are exactly NGROUP groups of variables, (ii) MIN_SIZE and MAX_SIZE are the number of variables of the smallest and largest groups, (iii) MIN_DIST and MAX_DIST are the minimum and maximum number of variables between two consecutive groups or between one border and one group, (iv) NVAL is the number of variables taking their value in the set VALUES.

For instance, group$(2, 2, 4, 1, 2, 6, \langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle, \{1, 2, 3\})$ holds since the sequence $\langle 0, 0, 1, 3, 0, 2, 2, 2, 3\rangle$ contains 2 groups $\langle 1, 3\rangle$ and $\langle 2, 2, 2, 3\rangle$ of nonzero values of size 2 and 4, 2 groups $\langle 0, 0\rangle$ and $\langle 0\rangle$ of zeros, and 6 nonzero values. The graph-based description of the group constraint uses two graph constraints which respectively mention the graph properties $\mathbf{NCC}$ = NGROUP, $\mathbf{MIN\_NCC}$ = MIN_SIZE, $\mathbf{MAX\_NCC}$ = MAX_SIZE, $\mathbf{NVERTEX}$ = NVAL and $\mathbf{MIN\_NCC}$ = MIN_DIST, $\mathbf{MAX\_NCC}$ = MAX_DIST. Figure 5 depicts the initial digraph as well as the two final digraphs associated to the two graph constraints of the example given for the group constraint.

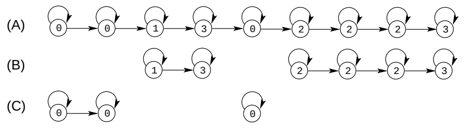


Fig. 5. Initial (A) and final digraphs (B,C) of group

6.2 Bounds and Filtering for the path_with_loops Graph Class

The path_with_loops properties highlight well the interest of specializing the parameter bound formula and the filtering rules. Firstly, in this context, the path structure of the considered digraphs naturally makes the different algorithms polynomial. The status of vertices and arcs can be determined and fixed during filtering in linear time by just following the path from vertex x_1 to vertex x_n. Secondly, some general bounds are not sharp anymore in this context because of the additional graph-class constraints. Refining those bounds then leads to a more accurate filtering. Consider for example bound $\overline{\mathbf{NVERTEX}} = |X_{TU}|$. In the general case, there exists a final digraph with a number of vertices equal to $|X_{TU}|$ because all U-vertices in the intermediate digraph can be turned into T-vertices. Since constraints (7) forbid two U-vertices linked by an F-arc to both be turned into T-vertices, bound $\overline{\mathbf{NVERTEX}}$ can be refined in the path_with_loops context to $|X_{TU}| - \sum_{i \in cc(\overrightarrow{G}(X_U, E_F))} \lfloor \frac{|vertex(i)|}{2} \rfloor$. This means that in any subpaths made of U-vertices and F-arcs, only one vertex out of

Table 3. Bounds of the graph parameters in the context of `path_with_loops`

Graph parameters	Bound
$\underline{\mathbf{NVERTEX}}$	$\lvert X_T \rvert$
$\overline{\mathbf{NVERTEX}}$	$\lvert X_{TU} \rvert - \sum_{i \in cc(\overrightarrow{G}(X_U, E_F))} \lfloor \frac{\lvert vertex(i) \rvert}{2} \rfloor$
$\underline{\mathbf{NCC}}$	$\lvert cc_{[\lvert X_T \rvert \geq 1]}(\overrightarrow{G}(X_{TU}, E_{TU})) \rvert$
$\overline{\mathbf{NCC}}$	$\lvert cc(\overrightarrow{G}(X_T, E_T)) \rvert +$ $\sum_{i \in cc_{[\lvert X_{U,U,T} \rvert = 0]}(\overrightarrow{G}(X_U, E_{UF}))} \lceil \frac{\lvert vertex(i) \rvert}{2} \rceil +$ $\sum_{i \in cc_{[\lvert X_{U,U,T} \rvert = 1]}(\overrightarrow{G}(X_U, E_{UF}))} \lfloor \frac{\lvert vertex(i) \rvert}{2} \rfloor +$ $\sum_{i \in cc_{[\lvert X_{U,U,T} \rvert = 2]}(\overrightarrow{G}(X_U, E_{UF}))} (\lceil \frac{\lvert vertex(i) \rvert}{2} \rceil - 1)$

$\underline{\mathbf{MIN\_NCC}}$
 if $\lvert X_T \rvert = 0$
 if $\lvert X_T \rvert \geq 1 \land \lvert X_{U,U,\neg T} \rvert \geq 1$
 if $\lvert X_T \rvert \geq 1 \land \lvert X_{U,U,\neg T} \rvert = 0$
	
	 0
	 1
	 $\min_{i \in cc(\overrightarrow{G}(X_T, E_T))} \lvert vertex(i) \rvert$

$\overline{\mathbf{MIN\_NCC}}$
 if $\lvert X_T \rvert \geq 1$
 if $\lvert X_T \rvert = 0$
	
	 $\min_{i \in cc_{[\lvert X_T \rvert \geq 1]}(\overrightarrow{G}(X_{TU}, E_{TU}))} \lvert vertex(i) \rvert - \epsilon$
	 $\max_{i \in cc(\overrightarrow{G}(X_{TU}, E_{TU}))} \lvert vertex(i) \rvert$

$\underline{\mathbf{MAX\_NCC}}$	$\max_{i \in cc(\overrightarrow{G}(X_T, E_T))} \lvert vertex(i) \rvert$
$\overline{\mathbf{MAX\_NCC}}$	$\max_{i \in cc(\overrightarrow{G}(X_{TU}, E_{TU}))} \lvert vertex(i) \rvert$

two may belong to a final digraph maximizing $\mathbf{NVERTEX}$. Lastly, the graph-class constraints allow to simplify some bounds that are sharp both in the general and the `path_with_loops` contexts. For example, general bound $\underline{\mathbf{NVERTEX}} = \lvert X_T \rvert + h(\overleftrightarrow{G}((X_{T,\neg T,\neg T}, X_{U,\neg T,T}), E_{U,T}))$ remains sharp for the `path_with_loops` graph class. Yet here, $X_{T,\neg T,\neg T}$ is empty since constraints (6) enforce each T-vertex to be an extremity of at least one T-arc (its loop). Hence, the formula can be simplified to $\underline{\mathbf{NVERTEX}} = \lvert X_T \rvert$ by removing the term involving the computation of the cardinal of a minimum hitting set. The same arguments hold for the three other graph parameters involved in the description of the `group` constraint.

Table 3[8] summarizes the bounds of these graph parameters in the `path_with_loops` graph class. Note that all these bounds can be evaluated in $O(n)$ time by iterating once through the n vertices of the initial digraph. Furthermore all the bounds are sharp when considering the graph-class constraints. As in the general case, they are derived by considering those final digraphs that minimize or maximize the corresponding graph parameter. Again identifying U-arcs and U-vertices belonging or not to these digraphs yields filtering rules that apply to the intermediate digraph when a given bound has to be reached.

For example, any final digraph G_f having exactly $\overline{\mathbf{NVERTEX}}$ vertices can be defined from the intermediate digraph as follows: (i) any U-vertex in $X_{U,\neg F}$

[8] By convention in these formulas, a maximum value over the empty set is zero. In the formula for $\overline{\mathbf{MIN\_NCC}}$, $\epsilon = 1$ if there exist two adjacent (linked by an F-arc) cc of minimum size in $cc_{[\lvert X_T \rvert \geq 1]}(\overrightarrow{G}(X_{TU}, E_{TU}))$, $\epsilon = 0$ otherwise.

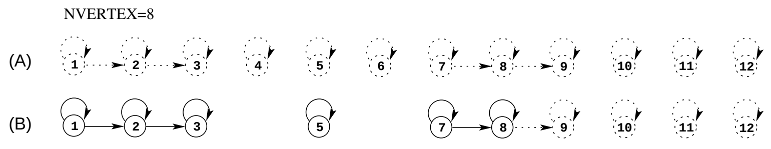


Fig. 6. Initial (A) and final digraphs (B) when filtering from $\mathrm{dom}(\mathbf{NVERTEX}) = 8$

belongs to G_f, and (ii) if a cc $\mathcal{C}$ of $\overrightarrow{G}(X_U, E_F)$ has an odd number of vertices $x_{p+1}, x_{p+2}, \ldots, x_{p+|vertex(\mathcal{C})|}$, then only vertices $x_{p+2\cdot i-1}$ $(i \in [1, \lceil \frac{|vertex(\mathcal{C})|}{2} \rceil])$ in $\mathcal{C}$ belong to G_f. Hence, filtering when condition $\mathrm{dom}(\mathbf{NVERTEX}) = \{\overline{\mathbf{NVERTEX}}\}$ holds consists in turning the pre-cited U-vertices of the intermediate digraph to T-vertices and all other vertices in a cc $\mathcal{C}$ to F-vertices.

Example 8. Figure 6 illustrates filtering according to the hypothesis that the final digraph should contain 8 vertices. The two cc of $\overrightarrow{G}(X_U, E_F)$ respectively correspond to the vertex sets $\{3, 4, 5, 6, 7\}$ and $\{9, 10, 11, 12\}$. The bound $\overline{\mathbf{NVERTEX}}$ is then equal to $12 - \lfloor \frac{5}{2} \rfloor - \lfloor \frac{4}{2} \rfloor = 8$. On one hand, since the first cc contains an odd number of vertices, then vertices $3, 5, 7$ are turned into T-vertices and vertices $4, 6$ are turned into F-vertices. On the other hand, since the second cc contains an even number of vertices nothing can be deduced about their status. Finally all vertices of $X_{U,\neg F} = \{1, 2, 8\}$ are turned into T-vertices.

We derived similar filtering rules corresponding to the bounds of Table 3. Details can be found in [1]. Note that they can be implemented in $O(m)$ by iterating once through the m variables of the initial digraph.

6.3 Performance

In order to evaluate the effectiveness of graph-based filtering, we performed two experiments, generating random instances of the `group` constraint. `VARIABLES` was chosen as a sequence of n domain variables ranging over $[0, 1]$, and `VALUES` as the singleton set $\{1\}$. A constraint instance was generated by setting the initial domain of each domain variable to a randomly chosen interval. Furthermore, with 10% probability, each variable in `VARIABLES` was randomly fixed. The experiments compare the effect of graph-based filtering with the approach of constructing an automaton for each graph characteristic and by reformulating that automaton as a conjunction of constraints as described in [2]. We call this approach automata-based filtering. For both methods, graph invariants, providing auxiliary constraints [4], were also posted.

In the first experiment, we computed the number of labeling choices made during search for all solutions for $n = 10$. In the second experiment, we computed the number of labeling choices made during search for the first solution for $n = 20$. We chose to count labeling choices as opposed to measuring runtime, as a fair runtime comparison would require a more polished implementation of graph-based filtering than we currently have. Note however that all filtering rules run in $O(n)$ time. The results are presented in two scatter plots in Figure 7. Each point represents a random instance, its

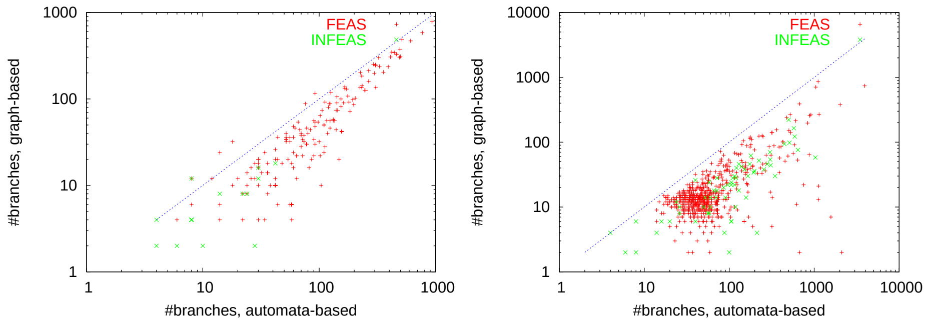


Fig. 7. Scatter plots of random instances. Left: comparing labeling choices for finding all solutions. Right: comparing labeling choices for finding the first solution.

X (resp. Y) coordinate corresponding to automata-based (resp. graph-based) filtering. Feasible and infeasible instances are distinguished in the plots.

From these experiments, we observe that most of the time, but not always, the graph-based method dominates the automata-based one. One would expect domination, as the graph method reasons about arc variables in addition to vertex variables. The graph method is currently limited by our approach to only apply the filtering when a graph parameter reaches one of its bounds. We observe no significant difference between the patterns for feasible vs. infeasible instances.

7 Conclusion

This article provided a first generic filtering scheme stemming from lower and upper bounds for common graph parameters used in the graph-based reformulation of global constraints. Moreover, it shows how we could retrieve most parts of an existing specialized filtering algorithm solely from the graph-based description. Our experiments on the example of the `path_with_loops` graph class point to an enhancement of the approach: filtering before a graph parameter reaches one of its bounds.

References

1. N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Graph properties based filtering. Technical report, Swedish Institute of Computer Science, 2006. SICS T2006-10.
2. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving Filtering Algorithms from Constraint Checkers. In M. Wallace, editor, *Principles and Practice of Constraint Programming (CP'2004)*, volume 3258 of *LNCS*, pages 107–122. Springer-Verlag, 2004.
3. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-06, Swedish Institute of Computer Science, 2005.
4. N. Beldiceanu, M. Carlsson, J.-X. Rampon, and C. Truchet. Graph Invariants as Necessary Conditions for Global Constraints. In P. van Beek, editor, *Principles and Practice of Constraint Programming (CP'2005)*, volume 3709 of *LNCS*, pages 92–106. Springer, 2005.

5. N. Beldiceanu, I. Katriel, and X. Lorca. Undirected Forest Constraints. In *CP-AI-OR'06*, volume 3990 of *LNCS*, pages 29–43, 2006.
6. N. Beldiceanu, T. Petit, and G. Rochart. Bounds of Graph Characteristics. In P. van Beek, editor, *CP'2005*, volume 3709 of *LNCS*, pages 742–746, 2005.
7. C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. *Among*, *common* and *disjoint* Constraints. In *CSCLP 2005*, pages 223–235, 2005.
8. COSYTEC. *CHIP Reference Manual*, release 5.1 edition, 1997.
9. G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint. In P. van Beek, editor, *Principles and Practice of Constraint Programming (CP'2005)*, volume 3709 of *LNCS*, pages 211–225. Springer-Verlag, 2005.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H.Freeman and co., San Francisco, 1979.
11. P. Martin and D. B. Shmoys. A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem. In *IPCO'96*, volume 1084 of *LNCS*, pages 389–403, 1996.
12. S. Micali and V. V. Vazirani. An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *FOCS 1980*, pages 17–27, New York, 1980. IEEE.
13. G. Pesant. A Filtering Algorithm for the *stretch* Constraint. In *CP'2001*, volume 2239 of *LNCS*, pages 183–195, 2001.
14. J.-C. Régin. The Symmetric *alldiff* Constraint. In *16th Int. Joint Conf. on Artificial Intelligence (IJCAI-99)*, pages 420–425, 1999.

The ROOTS Constraint

Christian Bessiere[1], Emmanuel Hebrard[2], Brahim Hnich[3], Zeynep Kiziltan[4], and Toby Walsh[5]

[1] LIRMM, CNRS/University of Montpellier, France
bessiere@lirmm.fr
[2] 4C and UCC, Cork, Ireland
e.hebrard@4c.ucc.ie
[3] Izmir University of Economics, Izmir, Turkey
brahim.hnich@ieu.edu.tr
[4] University of Bologna, Italy
zkiziltan@deis.unibo.it
[5] NICTA and UNSW, Sydney, Australia
tw@cse.unsw.edu.au

Abstract. A wide range of counting and occurrence constraints can be specified with just two global primitives: the RANGE constraint, which computes the range of values used by a sequence of variables, and the ROOTS constraint, which computes the variables mapping onto a set of values. We focus here on the ROOTS constraint. We show that propagating the ROOTS constraint completely is intractable. We therefore propose a decomposition which can be used to propagate the constraint in linear time. Interestingly, for all uses of the ROOTS constraint we have met, this decomposition does not destroy the global nature of the constraint as we still prune all possible values. In addition, even when the ROOTS constraint is intractable to propagate completely, we can enforce bound consistency in linear time simply by enforcing bound consistency on the decomposition. Finally, we show that specifying counting and occurrence constraints using ROOTS is effective and efficient in practice on two benchmark problems from CSPLib.

1 Introduction

Global constraints on the occurrence of particular values (*occurrence* constraints) or on the number of values or variables satisfying some condition (*counting* constraints) occur in many real world problems. They are especially useful in problems involving resources. For instance, if values represent resources, we may wish to count the number of occurrences of the different values used. Many global constraints proposed in the past are counting and occurrence constraints (see, for example, [13,3,14,1,4]). Bessiere et al. showed [5] that many such constraints can be specified with two new global constraints, ROOTS and RANGE, together with some simple elementary constraints like subset and set cardinality.

As we show here, specifying a global constraint using ROOTS and RANGE is also in many cases a way to provide an *efficient* propagator. There are three possible situations. In the first, we do not lose the "global" nature of our counting or occurrence constraint by specifying it with Roots and Range. The global nature of the Roots and Range constraint is enough to capture the global nature of the given counting or occurrence constraint, and propagation is not hindered. In the second situation, completely propagating the counting or occurrence constraint is NP-hard. We must accept some loss of globality if we are to make propagation tractable. Using Roots and Range is then one means to propagate the counting or occurrence constraint partially. In the third situation, the global constraint can be propagated completely in polynomial time but using Roots and Range hinders propagation. In this case, we need to develop a specialized propagation algorithm.

In [7], we focused on the Range constraint. This paper therefore concentrates on the Roots constraint. We prove that it is intractable to propagate the Roots constraint completely. We therefore propose a decomposition of the Roots constraint that can propagate it partially in linear time. This decomposition does not destroy the global nature of the Roots constraint as in many situations met in practice, it prunes all possible values. This decomposition can also easily be incorporated into a new constraint toolkit. We show experimentally the efficiency of using the Roots constraint on two real world problems from CSPLib. The rest of the paper is organised as follows. Section 2 gives the formal background. Section 3 gives many examples of counting and occurrence constraints that can be specified using the Roots constraint. In Section 4, we give a complete theoretical analysis of the Roots constraint and our decomposition of it. In Section 5, we discuss implementation details. Experimental results are presented in Section 6. Finally, we end with conclusions.

2 Formal Background

A constraint satisfaction problem consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for subsets of variables. We use capitals for variables (e.g. X, Y and S), and lower case for values (e.g. v and w). We write $D(X)$ for the domain of a variable X. For totally ordered domains, we write $min(X)$ and $max(X)$ for the minimum and maximum values. A solution is an assignment of values to the variables satisfying the constraints. A variable is *ground* when it is assigned a value. We consider both *integer* and *set* variables. A set variable S is represented by its lower bound $lb(S)$ which contains the definite elements and an upper bound $ub(S)$ which also contains the potential elements.

Constraint solvers typically explore partial assignments enforcing a local consistency property using either specialized or general purpose propagation algorithms. Given a constraint C, a *bound support* on C is a tuple that assigns to each integer variable a value between its minimum and maximum, and to each set variable a set between its lower and upper bounds which satisfies C. A bound support in which each integer variable is assigned a value in its domain is called a *hybrid support.* If C involves only integer variables, a hybrid support is a *support.* A constraint C is *bound consistent* (BC) iff for each integer variable X_i, its minimum and maximum values belong to a bound support, and for each set variable S_j, the values in $ub(S_j)$ belong to S_j in at least one bound support and the values in $lb(S_j)$ belong to S_j in all bound supports. A constraint C is *hybrid consistent* (*HC*) iff for each integer variable X_i, every value in $D(X_i)$ belongs to a hybrid support, and for each set variable S_j, the values in $ub(S_j)$ belong to S_j in at least one hybrid support, and the values in $lb(S_j)$ belong to S_j in all hybrid supports. A constraint C involving only integer variables is *generalized arc consistent* (*GAC*) iff for each variable X_i, every value in $D(X_i)$ belongs to a support. If all variables in C are integer variables, hybrid consistency reduces to generalized arc-consistency, and if all variables in C are set variables, hybrid consistency reduces to bound consistency.

To illustrate these concepts, consider the constraint $C(X_1, X_2, S)$ that holds iff the set variable S is assigned exactly the values used by the integer variables X_1 and X_2. Let $D(X_1) = \{1, 3\}$, $D(X_2) = \{2, 4\}$, $lb(S) = \{2\}$ and $ub(S) = \{1, 2, 3, 4\}$. BC does not remove any value since all domains are already bound consistent. On the other hand, HC removes 4 from $D(X_2)$ and from $ub(S)$ as there does not exist any tuple satisfying C in which X_2 does not take value 2. Note that as BC deals with bounds, value 2 was considered as possible for X_1.

3 Counting and Occurrence Constraints

Counting constraints limit the number of values or variables satisfying some condition (e.g. the global cardinality constraint [14] counts the number of variables using particular values). Occurrence constraints limit the occurrence of particular values (e.g. the all different constraint [13] ensures no value occurs twice). We previously showed [5] that many counting and occurrence constraints can be decomposed into two new global constraints, Range and Roots, together with simple non-global constraints over integer variables (like $X \leq m$) and simple non-global constraints over set variables (like $S_1 \subseteq S_2$ or $|S| = k$). We focus here on the Roots constraint. Given a sequence of variables X_1 to X_n, the Roots constraint holds iff a set variable S is the set of indices of variables which map to a value belonging to a second set variable, T.

$$\textsc{Roots}([X_1, \ldots, X_n], S, T) \text{ iff } S = \{i \mid X_i \in T\}$$

Note that elements in T may not be used by any integer variable X_i. For example, $\textsc{Roots}([1, 3, 1, 2, 3], S, T)$ is satisfied by $S = \{1, 3\}$ and $T = \{1\}$, $S = \{4\}$ and $T = \{2, 7\}$, or $S = \{2, 4, 5\}$ and $T = \{2, 3, 8\}$. We now list some of the uses of the Roots constraint for specifying other more complex global constraints.

3.1 Among Constraint

The Among constraint was introduced in CHIP to model resource allocation problems like car sequencing [3]. It counts the number of variables using values from a given set. $\textsc{Among}([X_1, \ldots, X_n], [d_1, \ldots, d_m], N)$ holds iff $N = |\{i \mid X_i \in \{d_1, \ldots, d_m\}\}|$. It can be decomposed using a Roots constraint:

$$\textsc{Among}([X_1, \ldots, X_n], [d_1, \ldots, d_m], N) \text{ iff}$$
$$\textsc{Roots}([X_1, \ldots, X_n], S, \{d_1, \ldots, d_m\}) \ \wedge \ |S| = N$$

GAC on Among is equivalent to HC on this decomposition [5]. As we show later, since the third argument of Roots is ground, we can achieve HC on the Roots constraint in linear time. We note that Roots is more than a set version of Among. With Among, we just count the number of variables using particular values. However, with Roots, we collect the set of variables using particular values. As we see later, having this set and not just its cardinality permits us to specify global constraints like Common which go beyond what can be expressed with Among.

3.2 Count Constraint

The Count constraint [2] is closely related to the Among constraint. The Count constraint permits us to constrain the number of variables using a particular value. More precisely, $\textsc{Count}([X_1, \ldots, X_n], d, \mathit{op}, N)$ where $\mathit{op} \in \{\leq, \geq, <, >, =, \neq\}$ holds iff $|\{i \mid X_i = d\}|\ \mathit{op}\ N$. The AtMost and AtLeast constraints are instances of Count where $\mathit{op} \in \{\leq, \geq\}$. The Count constraint can be decomposed into a Roots constraint:

$$\textsc{Count}([X_1, \ldots, X_n], d, \mathit{op}, N) \text{ iff}$$
$$\textsc{Roots}([X_1, \ldots, X_n], S, \{d\}) \ \&\ |S|\ \mathit{op}\ N$$

This decomposition does not hinder propagation and, as we will show later, it takes linear time to enforce HC on such an instance of the Roots constraint.

3.3 Domain Constraint

We may wish to channel between a variable and a sequence of 0/1 variables representing the possible values taken by the variable. The $\textsc{Domain}(X, [X_1, \ldots, X_m])$ constraint introduced in [12] ensures $X = i$ iff $X_i = 1$. This can be decomposed into a Roots constraint:

$$\textsc{Domain}(X, [X_1, \ldots, X_m]) \text{ iff}$$
$$\textsc{Roots}([X_1, \ldots, X_m], S, \{1\}) \ \&\ |S| = 1 \ \&\ X \in S$$

Enforcing HC on this specification again takes linear time and it is equivalent to enforcing GAC on the original global Domain constraint.

3.4 LinkSet2Booleans Constraint

We may also wish to channel between a set variable and a sequence of 0/1 variables representing the characteristic function of this set. The global constraint $\textsc{LinkSet2Booleans}(S, [X_1, \ldots, X_m])$ introduced in [2] ensures $i \in S$ iff $X_i = 1$. This can also be decomposed into a Roots constraint:

$$\textsc{LinkSet2Booleans}(S, [X_1, \ldots, X_m]) \text{ iff}$$
$$\textsc{Roots}([X_1, \ldots, X_m], S, \{1\})$$

Enforcing HC (or BC) on this specification again takes linear time and it is equivalent to enforcing HC (or BC) on the original global LinkSet2Booleans constraint.

3.5 Gcc Constraint

The global cardinality constraint [14] constrains the number of times values are used. We consider a generalization in which the number of occurrences of a value is an integer variable. That is, $\textsc{Gcc}([X_1, \ldots, X_n], [d_1, \ldots, d_m], [O_1, \ldots, O_m])$ holds iff $|\{i \mid X_i = d_j\}| = O_j$ for all j. Such a Gcc constraint can be decomposed into a set of Roots constraints:

$$\begin{aligned}&\textsc{Gcc}([X_1, \ldots, X_n], [d_1, \ldots, d_m], [O_1, \ldots, O_m]) \text{ iff}\\ &\forall j \,.\, \textsc{Roots}([X_1, \ldots, X_n], S_j, \{d_j\}) \;\&\; |S_j| = O_j\end{aligned}$$

Enforcing GAC on such a generalized Gcc constraint is NP-hard, but we can enforce GAC on the X_i and BC on the O_j in polynomial time using a specialized algorithm [11]. This is more than is achieved in general by enforcing HC on the specification using Roots [5].

3.6 Common Constraint

A generalization of the Among and AllDifferent constraints introduced in [2] is the Common constraint. $\textsc{Common}(N, M, [X_1, \ldots, X_n], [Y_1, \ldots, Y_m])$ ensures $N = |\{i \mid X_i = Y_j\}|$ and $M = |\{j \mid X_i = Y_j\}|$. That is, N variables in $X_1, \ldots, X_n$ take values in common with $Y_1, \ldots, Y_m$ and M variables in $Y_1, \ldots, Y_m$ take values in common with $X_1, \ldots, X_n$. We cannot expect to enforce GAC on such a constraint in general as it is NP-hard to do so [5]. One way to propagate a Common constraint is to decompose it into Range and Roots constraints:

$$\begin{aligned}&\textsc{Common}(N, M, [X_1, \ldots, X_n], [Y_1, \ldots, Y_m]) \text{ iff}\\ &\textsc{Range}([Y_1, \ldots, Y_m], \{1, \ldots, m\}, T) \;\&\\ &\textsc{Roots}([X_1, \ldots, X_n], S, T) \;\&\; |S| = N \;\&\\ &\textsc{Range}([X_1, \ldots, X_n], \{1, \ldots, n\}, V) \;\&\\ &\textsc{Roots}([Y_1, \ldots, Y_m], U, V) \;\&\; |U| = M\end{aligned}$$

where the Range constraint holds iff a set variable T equals the set of values used by those variables, X_1 to X_n whose index is in the set S.

$$\textsc{Range}([X_1, \ldots, X_n], S, T) \text{ iff } T = \{X_i \mid i \in S\}$$

Enforcing HC on this specification of the Common constraint again takes linear time. As no specialized propagation algorithm has yet been proposed for the Common constraint, Roots and Range provide a simple and promising means to propagate the constraint.

4 The Roots Constraint

We now give a thorough theoretical analysis of the Roots constraint. In Section 4.1, we provide a proof for the first time of the claim made in [5] that enforcing HC on Roots is NP-hard in general. Section 4.2 presents a decomposition of the Roots constraint that permits us to propagate the Roots constraint partially in linear time. Section 4.3 shows that in many cases this decomposition does not destroy the global nature of the Roots constraint as enforcing HC on the decomposition achieves HC on the Roots constraint. Finally, Section 4.4 shows that we can obtain BC on the Roots constraint by enforcing BC on its decomposition.

4.1 Complete Propagation

Unfortunately, propagating the Roots constraint completely is intractable in general. Whilst we made this claim in [5], a proof has not yet been published. For this reason, we give one here.

Theorem 1. *Enforcing HC on the* Roots *constraint is NP-hard.*

Proof. We transform 3Sat into the problem of the existence of a solution for Roots. Finding a hybrid support is thus NP-hard. Hence enforcing HC on Roots is NP-hard. Let $\varphi = \{c_1, \dots, c_m\}$ be a 3CNF on the Boolean variables $x_1, \dots, x_n$. We build the constraint Roots($[X_1, \dots, X_{n+m}], S, T$) as follows. Each Boolean variable x_i is represented by the variable X_i with domain $D(X_i) = \{i, -i\}$. Each clause $c_p = x_i \vee \neg x_j \vee x_k$ is represented by the variable X_{n+p} with domain $D(X_{n+p}) = \{i, -j, k\}$. We build S and T in such a way that it is impossible for both the index i of a Boolean variable x_i and its complement $-i$ to belong to T. We set $lb(T) = \emptyset$ and $ub(T) = \bigcup_{i=1}^{n} \{i, -i\}$, and $lb(S) = ub(S) = \{n+1, \dots, n+m\}$. An interpretation M on the Boolean variables $x_1, \dots, x_n$ is a model of φ iff the tuple τ in which $\tau[X_i] = i$ iff $M[x_i] = 0$ can be extended to a solution of Roots. (This extension puts in T value i iff $M[x_i] = 1$ and assigns X_{n+p} with the value corresponding to the literal satisfying c_p in M.) □

We thus have to look for a lesser level of consistency for Roots or for particular cases on which HC is polynomial. We will show that bound consistency is tractable and that, under conditions often met in practice (e.g. one of the last two arguments of Roots is ground), enforcing HC is also.

4.2 A Decomposition of Roots

To show that Roots can be propagated tractably, we will give a straightforward decomposition into ternary constraints that can be propagated in linear time. This decomposition does not destroy the global nature of the Roots constraint since enforcing HC on the decomposition will, in many cases, achieve HC on the original Roots constraint, and since in all cases, enforcing BC on the decomposition achieves BC on the original ROOTS constraint. Given ROOTS$([X_1, \ldots, X_n], S, T)$, we decompose it into the implications:

$$i \in S \rightarrow X_i \in T$$
$$X_i \in T \rightarrow i \in S$$

where $i \in [1..n]$. We have to be careful how we implement such a decomposition in a constraint solver. First, some solvers will not achieve HC on such constraints (see Sec 5 for more details). Second, we need an efficient algorithm to be able to propagate the decomposition in linear time. As we explain in more detail in Sec 5, a constraint solver could easily take quadratic time if it is not incremental.

We first show that this decomposition prevents us from propagating the ROOTS constraint completely. However, this is to be expected as propagating ROOTS completely is NP-hard and this decomposition is linear to propagate. In addition, as we later show, in many circumstances met in practice, the decomposition does not in fact hinder propagation.

Theorem 2. *HC on* ROOTS$([X_1, \ldots, X_n], S, T)$ *is strictly stronger than HC on* $i \in S \rightarrow X_i \in T$*, and* $X_i \in T \rightarrow i \in S$ *for all* $i \in [1..n]$.

Proof. Consider $X_1 \in \{1,2\}$, $X_2 \in \{3,4\}$, $X_3 \in \{1,3\}$, $X_4 \in \{2,3\}$, $lb(S) = ub(S) = \{3,4\}$, $lb(T) = \emptyset$, and $ub(T) = \{1,2,3,4\}$. The decomposition is HC. However, enforcing HC on ROOTS will prune 3 from $D(X_2)$. □

In fact, enforcing HC on the decompostion achieves a level of consistency between BC and HC on the original ROOTS constraint. In the next section, we identify exactly when it achieves HC on ROOTS.

4.3 Some Special Cases

Many of the counting and occurrence constraints do not use the ROOTS constraint in its more general form, but have some restrictions on the variables S, T or X_i's. For example, it is often the case that T or S are ground. We select four important cases that cover many of these uses of ROOTS and show that enforcing HC on ROOTS is then tractable.

C1. $\forall i \in lb(S), D(X_i) \subseteq lb(T)$
C2. $\forall i \notin ub(S), D(X_i) \cap ub(T) = \emptyset$
C3. $X_1 \ldots X_n$ are ground
C4. T is ground

We will show that in any of these cases, we can achieve HC on ROOTS simply by propagating the decomposition.

Theorem 3. *If one of the conditions* $C1$ *to* $C4$ *holds, then enforcing HC on* $i \in S \rightarrow X_i \in T$*, and* $X_i \in T \rightarrow i \in S$ *for all* $i \in [1..n]$ *achieves HC on* ROOTS$([X_1, \ldots, X_n], S, T)$.

Proof. Soundness. Immediate.

Completeness. We observe that if the ROOTS constraint is unsatisfiable then enforing HC on the decomposition will also fail. We assume therefore that the ROOTS constraint is satisfiable. We have to prove that, for any X_i, all the values in $D(X_i)$ belong to a solution of ROOTS, and that the bounds on S and T are as tight as possible. Our proof will exploit the following properties that are guaranteed to hold when we have enforced HC on the decomposition.

P1 if $D(X_i) \subseteq lb(T)$ then $i \in lb(S)$
P2 if $D(X_i) \cap ub(T) = \emptyset$ then $i \notin ub(S)$
P3 if $i \in lb(S)$ then $D(X_i) \subseteq ub(T)$
P4 if $i \notin ub(S)$ then $D(X_i) \cap lb(T) = \emptyset$
P5 if $D(X_i) = \{v\}$ and $i \in lb(S)$ then $v \in lb(T)$
P6 if $D(X_i) = \{v\}$ and $i \notin ub(S)$ then $v \notin ub(T)$
P7 if i is added to $lb(S)$ by the constraint $X_i \in T \rightarrow i \in S$ then $D(X_i) \subseteq lb(T)$
P8 if i is deleted from $ub(S)$ by the constraint $i \in S \rightarrow X_i \in T$ then $D(X_i) \cap ub(T) = \emptyset$

Let us prove that $lb(T)$ is tight. Suppose the tuple τ is a solution of the ROOTS constraint. Let $v \notin lb(T)$ and $v \in \tau[T]$. We show that there exists a solution with $v \notin \tau[T]$. (Remark that this case is irrelevant to condition C4.) We remove v from $\tau[T]$. For each $i \notin lb(S)$ such that $\tau[X_i] = v$ we remove i from $\tau[S]$. With C1 we are sure that none of the i in $lb(S)$ have $\tau[X_i] = v$, thanks to property P7 and the fact that $v \notin lb(T)$. With C3 we are sure that none of the i in $lb(S)$ have $\tau[X_i] = v$, thanks to property P5 and the fact that $v \notin lb(T)$. There remains to check C2. For each $i \in lb(S)$, we know that $\exists v' \neq v, v' \in D(X_i) \cap ub(T)$, thanks to properties P3 and P5. We set X_i to v' in τ, we add v' to $\tau[T]$ and add all k with $\tau[X_k] = v'$ to $\tau[S]$. We are sure that $k \in ub(S)$ because $v' \in ub(T)$ plus condition C2 and property P8.

Completeness on $ub(T)$, $lb(S)$, $ub(S)$ and X_i's are shown with similar proofs. Let $v \in ub(T) \backslash \tau[T]$. (Again C4 is irrelevant.) We show that there exists a solution with $v \in \tau[T]$. Add v to $\tau[T]$ and for each $i \in ub(S)$, if $\tau[X_i] = v$, put i in $\tau[S]$. C2 is solved thanks to property P8 and the fact that $v \in ub(T)$. C3 is solved thanks to property P6 and the fact that $v \in ub(T)$. There remains to check C1. For each $i \notin ub(S)$ and $\tau[X_i] = v$, we know that $\exists v' \neq v, v' \in D(X_i) \setminus lb(T)$ (thanks to properties P4 and P6). We set X_i to v' in τ and remove v' from $\tau[T]$. Each k with $\tau[X_k] = v'$ is removed from $\tau[S]$, and this is possible because we are in condition C1, $v' \notin lb(T)$, and thanks to property P7.

Let $v \in D(X_i)$ and $\tau[X_i] = v', v' \neq v$. (C3 is irrelevant.) Assign v to X_i in τ. If both v and v' or none of them are in $\tau[T]$, we are done. There remain two cases. First, if $v \in \tau[T]$ and $v' \notin \tau[T]$, the two alternatives to satisfy ROOTS are to add i in $\tau[S]$ or to remove v from $\tau[T]$. If $i \in ub(S)$, we add i to $\tau[S]$ and we are done. If $i \notin ub(S)$, we know that $v \notin lb(T)$ thanks to property P4. So, v is removed from $\tau[T]$ and we are sure that the X_j's can be updated consistently for the same reason as in the proof of $lb(T)$. Second, if $v \notin \tau[T]$ and $v' \in \tau[T]$, the two alternatives to satisfy ROOTS are to remove i from $\tau[S]$ or to add v to $\tau[T]$. If $i \notin lb(S)$, we remove i from $\tau[S]$ and we are done. If $i \in lb(S)$, we know that $v \in ub(T)$ thanks to property P3. So, v is added to $\tau[T]$ and we are sure that the X_j's can be updated consistently for the same reason as in the proof of $ub(T) \setminus \tau[T]$.

Let $i \notin lb(S)$ and $i \in \tau[S]$. We show that there exists a solution with $i \notin \tau[S]$. We remove i from $\tau[S]$. Thanks to property P1, we know that $D(X_i) \not\subseteq lb(T)$. So, we set X_i to a value $v' \in D(X_i) \setminus lb(T)$. With C4 we are done because we are sure $v' \notin \tau[T]$. With conditions C1, C2, and C3, if $v' \in \tau[T]$, we remove it from $\tau[T]$ and we are sure that the X_j's can be updated consistently for the same reason as in the proof of $lb(T)$.

Let $i \in ub(S) \setminus \tau[S]$. We show that there exists a solution with $i \in \tau[S]$. We add i to $\tau[S]$. Thanks to property P2, we know that $D(X_i) \cap ub(T) \neq \emptyset$. So, we set X_i to a value $v' \in D(X_i) \cap ub(T)$. With condition C4 we are done because we are sure $v' \in \tau[T]$. With conditions C1, C2, and C3, if $v' \notin \tau[T]$, we add it to $\tau[T]$ and we are sure that the X_j's can be updated consistently for the same reason as in the proof of $ub(T) \setminus \tau[T]$. □

4.4 Bound Consistency

In addition to being able to enforce HC on ROOTS in some special cases, enforcing HC on the decomposition always enforces a level of consistency at least as strong as BC. In fact, in any situation (even those where enforcing HC in intractable), enforcing BC on the decomposition enforces BC on the ROOTS constraint.

Theorem 4. *Enforcing BC on $i \in S \rightarrow X_i \in T$, and $X_i \in T \rightarrow i \in S$ for all $i \in [1..n]$ achieves BC on* ROOTS$([X_1, \ldots, X_n], S, T)$.

Proof. Soundness. Immediate.
Completeness. The proof follows the same structure as that in Theorem 3. We relax the properties P1–P4 into properties P1'–P4'.

P1' if $[min(X_i), max(X_i)] \subseteq lb(T)$ then $i \in lb(S)$
P2' if $[min(X_i), max(X_i)] \cap ub(T) = \emptyset$ then $i \notin ub(S)$
P3' if $i \in lb(S)$ then the bounds of X_i are included in $ub(T)$
P4' if $i \notin ub(S)$ then the bounds of X_i are outside $lb(T)$

Let us prove that $lb(T)$ and $ub(T)$ are tight. Let o be the total ordering on $D = \bigcup_i D(X_i) \cup ub(T)$. Build the tuples σ and τ as follows: For each $v \in lb(T)$: put v in $\sigma[T]$ and $\tau[T]$. For each $v \in ub(T) \setminus lb(T)$, following o, do: put v in $\sigma[T]$ or $\tau[T]$ alternately. For each $i \in lb(S)$, P3' guarantees that both $min(X_i)$ and $max(X_i)$ are in $ub(T)$. By construction of $\sigma[T]$ (and $\tau[T]$) with alternation of values, if $min(X_i) \neq max(X_i)$, we are sure that there exists a value in $\sigma[T]$ (in $\tau[T]$) between $min(X_i)$ and $max(X_i)$. In the case $|D(X_i)| = 1$, P5 guarantees that the only value is in $\sigma[T]$ (in $\tau[T]$). Thus, we assign X_i in σ (in τ) with such a value in $\sigma[T]$ (in $\tau[T]$). For each $i \notin ub(S)$, we assign X_i in σ with a value in $[min(X_i), max(X_i)] \setminus \sigma[T]$ (the same for τ). We know that such a value exists with the same reasoning as for $i \in lb(S)$ on alternation of values, and thanks to P4' and P6. We complete σ and τ by building $\sigma[S]$ and $\tau[S]$ consistently with the assignments of X_i and T. The resulting tuples satisfy Roots. From this we deduce that $lb(T)$ and $ub(T)$ are BC as all values in $ub(T) \setminus lb(T)$ are either in σ or in τ, but not both.

We show that the X_i are BC. Take any X_i and its lower bound $min(X_i)$. If $i \in lb(S)$ we know that $min(X_i)$ is in T either in σ or in τ thanks to P3' and by construction of σ and τ. We assign $min(X_i)$ to X_i in the relevant tuple. This remains a solution of Roots. If $i \notin ub(S)$, we know that $min(X_i)$ is outside T either in σ or in τ thanks to P4' and by construction of σ and τ. We assign $min(X_i)$ to X_i in the relevant tuple. This remains a solution of Roots. If $i \in ub(S) \setminus lb(S)$, assign X_i to $min(X_i)$ in σ. If $min(X_i) \notin \sigma[T]$, remove i from $\sigma[S]$ else add i to $\sigma[S]$. The tuple obtained is a solution of Roots using the lower bound of X_i. By the same reasoning, we show that the upper bound of X_i is BC also, and therefore, all X_i's are BC.

We prove that $lb(S)$ and $ub(S)$ are BC with similar proofs. Let us show that $ub(S)$ is BC. Take any X_i with $i \in ub(S)$ and $i \notin \sigma[S]$. Since X_i was assigned any value from $[min(X_i), max(X_i)]$ when σ was built, and since we know that $[min(X_i), max(X_i)] \cap ub(T) \neq \emptyset$ thanks to P2', we can modify σ by assigning X_i a value in $ub(T)$, putting the value in T if not already there, and adding i into S. The tuple obtained satisfies Roots. So $ub(S)$ is BC.

There remains to show that $lb(S)$ is BC. Thanks to P1', we know that values $i \in ub(S) \setminus lb(S)$ are such that $[min(X_i), max(X_i)] \setminus lb(T) \neq \emptyset$. Take $v \in [min(X_i), max(X_i)] \setminus lb(T)$. Thus, either σ or τ is such that $v \notin T$. Take the corresponding tuple, assign X_i to v and remove i from S. The modified tuple is still a solution of Roots and $lb(S)$ is BC. □

5 Implementation Details

This decomposition of the Roots constraint can be implemented in many solvers using disjunctions of membership constraints: `or(member(`i, S`), notmember(`X_i, T`))` and `or(notmember(`i, S`), member(`X_i, T`))`. However, this requires a little care. Unfortunately, some existing solvers (like Ilog Solver) may not achieve HC on such disjunctions of primitives. For instance, the negated membership constraint `notmember(`X_i, T`)` is activated only if X_i is instantiated with a value of T (whereas it should be as soon as $D(X_i) \subseteq lb(T)$). We have to ensure that the solver wakes up when it should to ensure we achieve HC. As we explain in the complexity proof, we also have to be careful that the solver doesn't wake up too often or we will lose the optimal $O(nd)$ time complexity which can be achieved.

Theorem 5. *It is possible to enforce HC (or BC) on the decomposition of* Roots$([X_1, \ldots, X_n], S, T)$ *in* $O(nd)$ *time, where* $d = max(\forall i.|D(X_i)|, |ub(T)|)$.

Proof. The decomposition of Roots is composed of $2n$ constraints. To obtain an overall complexity in $O(nd)$, the total amount of work spent propagating each of these constraints must be in $O(d)$.

First, it is necessary that each of the $2n$ constraints of the decomposition is not called for propagation more than d times. Since S can be modified up to n times (n can be larger than d) it is important that not all constraints are called for propagation at each change in $lb(S)$ or $ub(S)$. By implementing 'propagating events' as described in [10,15], we can ensure that when a value i is added to $lb(S)$ or removed from $ub(S)$, constraints $j \in S \rightarrow X_j \in T$ and $X_j \in T \rightarrow j \in S$, $j \neq i$, are not called for propagation.

Second, we show that enforcing HC on contraint $i \in S \rightarrow X_i \in T$ in $O(d)$. Testing the precondition (does i belong to $lb(S)$?) is constant time. If true, removing from $D(X_i)$ all values not in $ub(T)$ is in $O(d)$ and updating $lb(T)$ (if $|D(X_i)| = 1$) is constant time. Testing that the postcondidtion is false (is $D(X_i)$ disjoint from $ub(T)$?) is in $O(d)$. If false, updating $ub(S)$ is constant time. Thus HC on $i \in S \rightarrow X_i \in T$ is in $O(d)$. Enforcing HC on $X_i \in T \rightarrow i \in S$ is in $O(d)$ as well because testing the precondition ($D(X_i) \subseteq lb(T)$?) is in $O(d)$, updating $lb(S)$ is constant time, testing that the postcondition is false ($i \notin ub(S)$?) is constant time, and removing from $D(X_i)$ all values in $lb(T)$ is in $O(d)$ and updating $ub(T)$ (if $|D(X_i)| = 1$) is constant time.

When T is modified, all constraints are potentially concerned. Since T can be modified up to d times, we can have d calls of the propagation in $O(d)$ for each of the $2n$ constraints. It is thus important that the propagation of the $2n$ constraints is *incremental* to avoid an $O(nd^2)$ overall complexity. An algorithm for $i \in S \rightarrow X_i \in T$ is incremental if the complexity of calling the propagation of the constraint $i \in S \rightarrow X_i \in T$ up to d times (once for each change in T or $D(X_i)$) is the same as propagating the constraint once. This can be achieved by an AC2001-like algorithm that stores the last value found in $D(X_i) \cap ub(T)$, which is a witness that the postcondition can be true. (Similarly, the last value found in $D(X_i) \setminus lb(T)$ is a witness that the precondition of the constraint $X_i \in T \rightarrow i \in S$ can be false.) Finally, each time $lb(T)$ (resp. $ub(T)$) is modified, $D(X_i)$ must be updated for each i outside $ub(S)$ (resp. inside $lb(S)$). If the propagation mechanism of the solver provides the values that have been added to $lb(T)$ or removed from $ub(T)$ to the propagator of the $2n$ constraints (as described in [16]), updating a given $D(X_i)$ has a total complexity in $O(d)$ for the d possible changes in T. The proof that BC can also be enforced in linear time follows a similar argument. □

6 Experimental Results

We now demonstrate that specifying global counting and occurrence constraints using Roots is effective and efficient in practice using two benchmark problems.

6.1 Balanced Academic Curriculum Problem

We implemented in Ilog Solver the constraint model of the Balanced Academic Curriculum Problem (BACP) (prob030 in CSPLib) proposed in [9] and compared it against a model using Roots. In this problem, we need to design a balanced

Variables	Encoding
curriculum:	CURMATRIX[1..#courses][1..#periods] in $\{0,1\}$
academic load:	LOAD[1..#periods] in $[a..b]$

Constraints	Encoding
exactly one period per course	$\forall i \in [1..\#courses]$ $\sum_{j=1}^{\#periods}\text{CURMATRIX}[i][j] = 1$
academic load	$\forall j \in [1..\#periods]$ $\text{LOAD}[j] = \sum_{i=1}^{\#courses}(\text{CURMATRIX}[i][j] * credit[i])$
prerequisites	$\forall (i \prec j) \in$ prerequisites, $\forall k \in [1..\#\text{periods}]$ $\sum_{r=1}^{k-1}(\text{CURMATRIX}[i][r]) \geq \text{CURMATRIX}[i][k]$
number of courses	$\forall j \in [1..\#periods]$ $c \leq \sum_{i=1}^{\#courses}\text{CURMATRIX}[i][j] \leq d$

Fig. 1. Boolean model

Variables	Encoding
curriculum:	CURRICULUM[1..#courses] in [1..#periods]
	CURMATRIX[1..#courses][1..#periods] in $\{0,1\}$
academic load:	LOAD[1..#periods] in $[a..b]$

Constraints	Encoding
channeling	$\forall i \in [1..\#\text{courses}]$ $\text{CURMATRIX}[i][\text{CURRICULUM}[i]] = 1$
academic load	$\forall j \in [1..\#\text{periods}]$ $\text{LOAD}[j] = \sum_{i=1}^{\#\text{courses}}(\text{CURMATRIX}[i][j] * credit[i])$
prerequisites	$\forall (i \prec j) \in$ prerequisites $\text{CURRICULUM}[i] < \text{CURRICULUM}[j]$
number of courses	$\text{Gcc}([c..d], ..[c..d], \{1, 2, ..\#\text{periods}\}, \text{CURRICULUM})$

Fig. 2. Primal-dual model

academic curriculum by assigning periods to courses so that the academic load of each period is balanced, i.e., as similar as possible. The goal is to assign a period to every course so that the constraints on the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships are satisfied. An optimal balanced curriculum minimises the maximum academic load for all periods.

We used two models from [9] (Figures 1 and 2) and compared them against a model using Roots (Figure 3). In the Roots model, the curriculum is encoded with integer variables mapping courses to periods, as in the primal-dual model of Figure 2. However, instead of using a Gcc constraint to restrict the number of courses per periods, we use the Roots constraint to link these variables to set variables standing for periods. We then restrict the number of courses per periods with cardinality constraints on these sets. The constants a, b, c, d correspond respectively to the minimum and maximum academic load, and the minimum and maximum number of courses per period. The array $credit[1..\#\text{courses}]$ map courses to their academic credits. We added to all models the implied constraint $\sum_{j=1}^{\#\text{periods}}(\text{LOAD}[j]) = \sum_{i=1}^{\#\text{courses}}(credit[i])$.

Variables	Encoding
curriculum:	CURRICULUM[1..#courses] in [1..#periods]
	CURSET[1..#periods] $\subseteq$ {1..#courses}
academic load:	LOAD[1..#periods] in $[a..b]$

Constraints	Encoding
channeling	$\forall j \in$ [1..#periods]
	Roots(CURRICULUM,CURSET[j], $\{j\}$)
academic load	$\forall j \in$ [1..#periods]
	LOAD[j] $= \sum_{i=1}^{\#\text{courses}}((i \in$CURSET[$j$]) $*$ $credit[i]$)
prerequisites	$\forall (i \prec j) \in$ prerequisites
	CURRICULUM[i] <CURRICULUM[j]
number of courses	$\forall j \in$ [1..#periods]
	$c \leq$ |CURSET[j]| $\leq d$

Fig. 3. Roots model

Table 1. Balanced Academic Curriculum Problem

	Boolean model			Prima-dual model			Roots model		
Size	#fails	time (s)	max load	#fails	time (s)	max load	#fails	time (s)	max load
8	413,418	18.44	17(*)	294	**0.04**	17(*)	**75**	0.09	17(*)
10	-	-	14(*)	170	**0.02**	14(*)	**121**	0.15	14(*)
12	1251	0.05	17(*)	255	**0.05**	17(*)	**194**	0.51	17(*)
16	-	-	-	429	**0.15**	17(*)	**263**	1.58	17(*)
20	-	-	-	410	**0.21**	19	**406**	3.18	19
20	-	-	-	701	**0.41**	18	**510**	13.12	18

We report in Table 1 the number of fails, cpu time for finding the best solution and the maximum academic load on 6 different instances. The 3 first instances, involving 8, 10 and 12 periods, are those solved in [9]. The 3 next instances were created by simply duplicating and renaming courses. The number of periods and courses are doubled, and for each prerequisite relation $(i \prec j)$ in the initial instance, we add $(i \prec j')$ and $(i' \prec j')$ where i' and j' are the duplicated courses for respectively i and j.

When the maximum academic load is followed by a star (*), it means that this is optimal and is proved so with a few more backtracks. The time cutoff was set to 300 seconds. An entry marked as "-" means no answer was obtained by the cut-off time. We observe that the model using Roots is the most efficient in terms of size of the search tree by a small margin. However the most efficient model in cpu time is the primal-dual model which uses the highly optimized Gcc constraint. Both clearly dominate the simple Boolean model despite the fact that this model only has linear constraints.

6.2 Mystery Shopper Problem

We used a model for the Mystery Shopper problem [8] due to Helmut Simonis that appears in CSPLib (prob004). We used the same problem instances as in [5] but perform a more thorough and extensive analysis. We partition the constraints of this problem into three groups:

Table 2. Mystery Shopper, branching on the integer variable with minimum domain

	`Alld-Gcc-Sum`			`Alld-Gcc`-Roots			`Alld`-Roots-Roots		
Size	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved
10	6	0.01	**9/10**	6	0.01	**9/10**	7	0.03	**9/10**
15	6,566	0.76	**29/52**	6,468	1.38	**29/52**	10,749	19.47	28/52
20	98,497	14.52	**21/35**	2,425	0.83	20/35	2,429	7.30	20/35
25	317	0.13	**13/20**	317	0.20	**13/20**	285	1.37	11/20
30	93,461	26.09	**7/10**	93,461	43.89	**7/10**	7,239	42.00	5/10
35	52,435	16.33	**22/56**	23,094	14.25	21/56	13	1.10	18/56

Table 3. Mystery Shopper, branching on set variables when possible

	`Alld-Gcc-Sum`			`Alld-Gcc`-Roots			`Alld`-Roots-Roots		
Size	#fails	time (s)	#solved	#fails	time (s)	#solved	#fails	time (s)	#solved
10	6	0.01	9/10	4247	0.83	3/10	318	0.38	**10/10**
15	6566	0.76	29/52	17210	4.31	16/52	102	0.25	**52/52**
20	98497	14.52	21/35	150473	49.95	7/35	930	2.95	**32/35**
25	317	0.13	13/20	265219	124.49	2/20	2334	11.17	**19/20**
30	93461	26.09	7/10	37	0.08	1/10	6766	39.63	**9/10**
35	52435	16.33	22/56	1216	0.53	4/56	4798	35.60	**49/56**

Temporal and geographical: All visits for any week are made by different shoppers. Similarly, a particular area cannot be visited more than once by the same shopper.

Shopper: Each shopper makes exactly the required number of visits.

Saleslady: A saleslady must be visited by some shoppers from at least 2 different groups (the shoppers are partitioned into groups).

Whilst the first group of constraints can be modelled by using AllDifferent constraints [13], the second can be modelled by Gcc [14] and the third by Among constraints [3]. We experimented with several models using Ilog Solver where these constraints are either implemented as their Ilog Solver primitives (respectively, `IloAllDiff`, `IloDistribute`, and a decomposition using `IloSum` on Boolean variables) or as their decompositions with Roots. Note that the Boolean decomposition of the Among constraint maintains GAC [6]. Due to space limitation, we report results for just the following models: `Alld-Gcc-Sum` (only Ilog Solver primitives), `Alld-Gcc`-Roots (Among encoded as Roots), and `Alld`-Roots-Roots (Among and Gcc encoded as Roots). Among encoded as Roots uses the decomposition presented in Section 3.1 and Gcc uses the decomposition presented in Section 3.5. All instances solved in the experiments use a time limit of 5 minutes. The cpu time reported for a method on a class of problems is averaged on the instances solved (#solved) by the method.

When branching on the integer variables (Table 2), the `Alld-Gcc-Sum` model tends to perform better than the other models (bold numbers). However, we obtain the best results by branching on the set variables introduced for modelling with Roots (see Table 3). By encoding the second and the third groups of constraints using Roots (the `Alld`-Roots-Roots model) and branching on the set variables, we are able to solve more instances. These results are primarily due to the better branching strategy. However, such a strategy would not be easily implementable without ROOTS since the extra set variables are part of it.

7 Conclusion

We have presented a comprehensive study of ROOTS, a global constraint that can specify many other global constraints, such as occurrence and counting constraints. We proved that propagating completely the ROOTS constraint is intractable in general. We therefore proposed a decomposition to propagate it partially. This decomposition achieves hybrid consistency on the global ROOTS constraint under some simple conditions often met in practice. In addition, enforcing bound consistency on the decomposition achieves bound consistency on the global ROOTS constraint whatever conditions hold. Our experiments show that this is practical method to implement many global constraints. We hope that by presenting these results, developers of the many different constraint toolkits will be encouraged to include the ROOTS constraint into their solvers. In our future work, we intend to consider other classes of global constraints (e.g. sequencing constraints) and to identify the primitives needed to specify and propagate these.

References

1. N. Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. In *Proc. CP'01*, pp. 211–224, Springer, 2001.
2. N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global constraint catalog. Technical Report T2005:08, SICS, 2005.
3. N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
4. N. Beldiceanu, I. Katriel, and S. Thiel. Filtering algorithms for the *same* and *usedby* constraints. In *MPI Technical Report MPI-I-2004-1-001*, 2004.
5. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The Range and Roots constraints: Specifying counting and occurrence problems. In *Proc. of IJCAI'05*, pp 60–65, 2005.
6. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh, 'Among, Common and Disjoint Constraints, to appear in LNAI of Springer.
7. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The Range constraint: Algorithms and implementation, to appear in *Proc. of CPAIOR'06*.
8. B.M.W. Cheng, K.M.F. Choi, J.H.M. Lee, and J.C.K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4:167–192, 1999.
9. B. Hnich, Z. Kiziltan and T. Walsh. Modelling a Balanced Academic Curriculum Problem. In *Proc. of CPAIOR'02*, pp. 121–131, 2002.
10. F. Laburthe. Choco: implementing a CP kernel. In *Proc. of CP'00 Workshop TRICS: Techniques foR Implementing Constraint programming Systems*, 2000.

11. C.G. Quimper, P. van Beek, A. Lopez-Ortiz, A. Golynski and S.B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Proc. of CP'03*, Springer, 2003.
12. P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In *Proc. of CP'00*, pp. 369–383, Springer, 2000.
13. J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI'94*, pp 362–367, 1994.
14. J.C. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. of AAAI'96*, pp. 209–215, 1996.
15. C. Schulte and P.J. Stuckey. Speeding up constraint propagation. In *Proc. of CP'04*, pp. 619–633, Springer, 2004.
16. P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

CoJava: Optimization Modeling by Nondeterministic Simulation

Alexander Brodsky[1,2] and Hadon Nash[3]

[1] George Mason Unviersity, Virginia, USA
[2] Adaptive Decisions, Inc., Maryland, USA
[3] Google, California, USA
brodsky@gmu.edu, hadonn@gmail.com

Abstract. We have proposed and implemented the language CoJava, which offers both the advantages of simulation-like process modeling in Java, and the capabilities of true decision optimization. By design, the syntax of CoJava is identical to the programming language Java, extended with special constructs to (1) make a nondeterministic choice of a numeric value, (2) assert a constraint, and (3) designate a program variable as the objective to be optimized. A sequence of specific selections in nondeterministic choice statements corresponds to an *execution path*. We define an *optimal* execution path as one that (1) satisfies the range conditions in the choice statements, (2) satisfies the assert-constraint statements, and (3) produces the optimal value in a designated program variable, among all execution paths that satisfy (1) and (2). The semantics of a CoJava program amounts to first finding an optimal execution path, and then procedurally executing it. To find an optimal execution path, the implemented CoJava compiler reduces the problem to a standard optimization formulation, and then solves it on an external solver. Then, the CoJava program is run as a Java program, where the choice statements select the found optimal values, and the assert and optimization statements are ignored. We illustrate the usage and semantics of CoJava using a simple supply-chain example, in which elastic demand, a manufacturer and a supplier are modeled as Java classes.

1 Introduction

Both numeric simulation and constraint-based optimization are successfully applied in wide variety of domains. This paper is concerned with developing a unified object-oriented (OO) language supporting both simulation modeling and decision optimization.

The Problem: Simulation vs. Optimization Models

In decision optimization problems one often needs to model a real-world process, such as a supply chain, an interaction of physical devices, a chemical process, or activities of a robot. However, describing a process using traditional operations research (OR) modeling, with decision variables, constraints, and objective functions, is quite a challenging task for non-OR professionals, even for those with general computer science and programming skills.

The reason for that challenge is that the elements of an OR model are abstract constraints, which have only an indirect connection to elements of a real-world process. For example, one equation may combine elements from several real-world devices. Also, the notions of order and timing of events are usually not explicit in OR models, which puts additional burden on the modeler. Furthermore, the execution of the optimization is typically a black box for the modeler, with no clear connection to the flow of the real world process. This makes debugging of an optimization model a challenging task. If the optimization fails there is no clear explanation for the failure. Finally, OR models typically lack the modularity of modern OO languages, so they tend to become difficult to maintain over time (like "spaghetti code").

By contrast, simulations are generally well understood by software developers. The elements of a simulation are state variables and state-transitions, which have a clear one-to-one correspondence with elements of a real-world process. Every quantity from the real-world process is represented by a single state variable, so there is little room for confusion. Real-world time and sequence of events correspond to time and sequence in the running simulation in an obvious way. Also, the "cause and effect" progression of the simulation is easy to follow. If the simulation fails, the exact time and place of the failure is reported. Finally, simulation modelers can practice modern OO software engineering. Complex building blocks can be modeled using simpler building blocks, and so on. In fact, modern OO languages have been derived from early simulation systems.

While simulation offers numerous advantages in ease of modeling and debugging, OR modeling has one major advantage. If modeled correctly using a manageable constraint domain such as LP or MILP, an optimization problem can be solved efficiently using existing solvers with sophisticated optimization algorithms. By contrast, no such solvers exist for simulation models. Typically, simulations are optimized by choosing parameters manually. An optimization layer can be added by running a simulation multiple times, with possible heuristics. However, such a search cannot compete with performance of solvers on manageable constraint domains.

Contributions

We have proposed and implemented the language CoJava, which offers both the advantages of simulation-like process modeling in Java, and the capabilities of true decision optimization.

By design, the syntax of CoJava is identical to the programming language Java, extended with special constructs to (1) make a nondeterministic choice of a numeric value that satisfies a range condition, (2) assert a constraint, and (3) designate a program variable as the objective to be optimized.

A sequence of specific selections in nondeterministic choice statements corresponds to an *execution path*. We define a *feasible* execution path as one that satisfies (1) the range conditions in the choice statements, and (2) the assert-constraint statements. An *optimal* execution path is a *feasible* path that produces the optimal value in a designated program variable, among all *feasible* execution paths. The semantics of a CoJava program amounts to first finding an *optimal* execution path, and then procedurally executing it.

To optimize a process, such as a supply chain example (Figures 1 and 2) discussed in the next section, each real-world device or facility is modeled, tested and debugged in pure Java as a class of objects with private state and public methods which change the state. A process is described as a method of a separate Java class, which invokes methods of the model objects passing nondeterministic choices for arguments, and which designates an optimization objective.

For model developers, it appears as if the program has simply followed a single execution path which coincidentally produces the optimal objective value. Since CoJava builds on software developers' existing skills, the learning curve for them is minimal. For OR professionals, CoJava enables development of decision models in the most natural way, which preserves the one-to-one correspondence with the components of a real-world process. Moreover, it provides OR modelers with the powerful OO language features, and permits complex models to be organized into self-contained business objects and processes, which reduces development time and allows easy extensibility.

To be able to find an optimal execution path for a CoJava program, we have developed a reduction to a standard constraint optimization formulation. Constraint variables represent values on program variables that can be created at any state of a nondeterministic execution. Constraints encode transitions, triggered by CoJava statements, from one program state to the next, and also capture conditions in the assert statements.

The reduction and the implementation are made under three simple restrictions, explained later in the paper. Namely, that (1) Boolean exit conditions in loops can not involve nondeterministic values, (2) recursive method calls cannot be made from a nondeterministic conditional statement, and (3) an `objective` parameter requested to be optimized cannot appear in a nondeterministic conditional statement.

The restrictions ensure that the length of any execution path (in terms of the number of program states it goes through), and thus a number of values generated, are not dependent on nondeterministic choices. The restrictions also ensure that any nondeterministic `choice` statements, and Boolean conditions on nondeterministic values in `assert` statements have a unique corresponding `objective` parameter, which needs to be optimized.

We have developed a CoJava constraint compiler, and integrated it, loosely, with the Eclipse IDE. The compiler is based on the reduction of the problem of finding an optimal execution path to a standard constraint optimization formulation. The CoJava compiler operates by first translating the CoJava program into a very similar Java program in which the primitive numeric operators and data types are replaced by symbolic constraint operators and data types. This intermediate Java program functions as a constraint generator. This program is compiled and executed to produce a standard optimization problem formulation (in AMPL in the current implementation). The standard optimization formulation is then submitted to and solved by an external optimization solver. Finally, the original CoJava program, being a compilable Java program, is run deterministically using the Java compiler, with the *choice* method re-implemented to select optimal values returned by the solver. The current

Under the restrictions on the use of nondeterministic values, the reduction to the standard optimization formulation is guaranteed to always work and be correct. However, the resulting formulation may be beyond the constraint domain handled by the available external solver. Thus, similar to OR modeling, a CoJava developer needs to be aware to only use arithmetic expressions that can be handled by the solver used. For example, if an MILP solver is used, a CoJava program can only use arithmetic expressions that are linear in nondeterministic values (but arbitrarily complex in deterministic values). Or if a non-linear solver such as MINUS or SNOPT is used, a CoJava program can use non-linear arithmetic expressions, but no nondeterministic conditional statements.

Related Work

CoJava addresses the goals of constraint modeling and object oriented simulation. Object oriented simulation has traditionally been approached through procedural object oriented languages, such as Smalltalk [1] and Java [2]. These languages start with a syntax for variable assignment and add support for modular organization of procedures. There are many specialized object oriented simulation languages such as Simula [3] and ModSim [4], and there are simulation environments layered on top of existing object oriented languages such as Silk [5] and JWarp [6]. These languages allow complex models to be constructed and maintained effectively, but lack support for systematic optimization.

Constraint modeling has traditionally been approached through specialized constraint modeling languages, such as AMPL [7] and GAMS [8]. These languages start with a syntax for equations and layer additional support for organizing equations and other constraints. They enable systematic optimization, but they require explicit definition and maintenance of equations and other constraints.

Constraint programming (CP) languages, such as OPL [9], CLP [10], Claire [11], Oz [12], ECLiPSe [13], ILOG SOLVER [14] allow developers to specify strategies for solving optimization problems. In [15], an imperative programming language is extended with CP-like search semantics. Certain CP languages provide support for object oriented modeling as well. However, the focus of CP is on solving optimization problems, while modeling such problems remains similar to AMPL-like modeling languages. CP languages do provide the notion of (non-deterministic) choice points, which are used in the specification of a search strategy. The nondeterministic *choice* method in CoJava is used for a different purpose, namely to indicate that the compiler has the flexibility to choose a particular value in order to minimize or maximize a program variable which appears later in the program.

In recent years, there has been considerable interest in languages that combine constraints with object oriented programming. These languages are motivated by the need for modular construction of optimizable models. Languages such as Cob [16] and Siri [17] add object oriented modeling constructs, such as inheritance and encapsulation, to an equational syntax. These languages provide a very clean representation for steady-state optimization problems, but they do not model state changes in the direct way that procedural languages do.

Other languages combine constraints with object oriented procedures in a "hybrid" fashion, maintaining program states and constraints side by side. Some languages and systems such as ILOG Solver [14,18], Choco [19], Gecode [20] and CCUBE [21] add explicit constraint objects to an OO language. Other languages such as ThingLab [22] and Kaleidoscope [23] impose both state changes and constraints on the same object attributes. These languages provide separate procedure execution semantics and constraint solving semantics which interact during program execution.

The languages most closely related to CoJava are those that translate procedural algorithms into declarative constraints. These languages unify procedural and constraint semantics, such that the same program statements determine both interpretations. The language Modelica [24] supports unified models, which can define both simulation and optimization problems. Modelica models are translated into equations, which may in turn be solved by an optimizer or sorted and compiled into an efficient sequential procedure. Within pure functions (functions without side effects), Modelica can also translate procedural algorithms into constraint equations. Within these functions, Modelica gains the advantage of specifying constraints using familiar procedural operations and flow of control. However, Modelica is fundamentally an equational language, and it supports procedural algorithms only within this limited context.

By contrast, CoJava has a thoroughly procedural object oriented syntax and semantics, (which is in fact identical to that of Java). CoJava presents the developer with no visible boundary between procedures and constraints. Familiar procedural operations and flow of control can be used uniformly throughout an entire model, or even throughout an entire software system. Our philosophy is to minimize the learning curve for developers, and to minimize the "impedance mismatch" between procedures and constraints, by conforming to a single well understood syntax and semantics. CoJava gives developers the flexibility to move model components freely back and forth between procedural algorithms and declarative optimization models. We believe this capability is unique to CoJava.

2 Syntax and Semantics of CoJava

Syntax: CoJava is Extended Java, Which Is Compilable Java

By design, the syntax of CoJava is identical to that of the Java programming language, extended with one special library class, and a few restrictions on how its methods can interact with the rest of the program. More specifically, CoJava adds the following special class to Java:

```
public class Nd {
  public double choice(double min, double max) {...}
  public double checkMinObjective(double objective) {...}
  public double checkMaxObjective(double objective) {...}
}
```

We describe CoJava's semantics, which is procedural execution of an optimal execution path, in this section. A CoJava program can also be run as a regular Java program, in which the methods of the class `Nd` operate as follows. The method `choice(min,max)` returns a single specific value between `min` and `max`, inclusive. The user can use her own implementation of the *choice* method, or use the default CoJava implementation (which is currently a random selection using uniform distribution).

The methods `checkMinObjective` and `checkMaxObjective`, in the regular Java execution, do nothing but output the value of the parameter `objective`. A CoJava program can also use the Java command `assert(booleanCondition)` with the standard procedural semantics, namely the program will report an error if the `booleanCondition` is not satisfied. We start with an example.

Example

Figures 1 and 2 show the source code for a simple CoJava program, composed of four classes: ExampleSupplyChain, Demand, Manufacturer, and Supplier. The class ExampleSupplyChain encodes a simulation procedure which uses the three other classes. We first explain the program as purely a Java program, and then explain its optimal execution path semantics.

Class Demand models consumer demand resulting from the chosen prices. Class Manufacturer models the manufacturing process, and computes the quantity of materials required, and the manufacturing costs, given the quantities of product consumed. Class Supplier models the cost of the required materials, and offers a discount for larger quantities. All of these results are computed in constructor methods in order to keep the example concise.

The remaining class, ExampleSupplyChain, calls procedures of the other classes in sequence to simulate a complete production process. First, prices for the two products are chosen. Then the consumer demand, manufacturing costs, and supply costs are computed. Then revenue from consumers is tallied, and production costs are subtracted, to compute total profit. Finally, total profit is designated as the "objective" for this procedure. If we ran the program as a simple Java program, on each run prices for the two products would be randomly selected, which would result in different total profit. For example, we might run the program three times, and see the following output:

```
  price_0: 124.69  price_1: 181.46  profit: 181381.83
  price_0: 179.40  price_1: 131.72  profit:  61274.39
  price_0: 121.82  price_1: 194.85  profit: -49093.00
```

```
package example;
import cojava.Nd;

public class ExampleSupplyChain
{
    public static void main(String[] args)
    {
        double[] prices = new double[2];
        prices[0] = Nd.choice(100, 200);
        prices[1] = Nd.choice(105, 205);

        Demand demand = new Demand(prices);
        Manufacturer manufacturer = new Manufacturer(demand.quantities);
        Supplier supplier = new Supplier(manufacturer.materials);

        double revenue = 0;
        for (int i = 0; i < 2; i = i + 1) {
            revenue += demand.revenues[i];
        }
        double cost = manufacturer.cost + supplier.cost;
        double profit = revenue - cost;
        Nd.checkMaxObjective(profit);
        /* ... printing statements omitted here ... */
    }
}
```

Fig. 1. Simple Supply Chain Class

Now let us see how the same program, interpreted as a CoJava program, represents an optimization problem. Above, we ran the program three times, and exhibited three different execution paths for the program. Given its two nondeterministic-choices, this program defines multiple execution paths. The semantics of a CoJava program amounts to first finding an optimal execution path, i.e., the one that would result in the maximal profit, and then procedurally executing it. In this example, an optimal execution path corresponds to the choice of prices 200.0 and 185.0, which would result in a profit of 401555.00. When we compile and run this program using the CoJava constraint compiler, and produce the correct CoJava execution path, we get the following:

```
price_0: 200.0  price_1: 185.0  profit: 401555.00
```

Restrictions on the Nondeterministic (Nd) Class Methods

Certain restrictions on how the methods `choice(...)`, `checkObjective`, and the command `assert` interact with the rest of CoJava program are imposed to make the optimization semantics well-defined and computable.

More specifically, the purpose of the restrictions is to control the size of the set of values that are computed during any execution path of the program. As

```
class Demand
{
    public double[] quantities = {0, 0};
    public double[] revenues = {0, 0};
    final double[] pLow = {100, 105};
    final double[] pMid = {150, 185};
    final double[] pHigh = {200, 205};
    final double[] qLow = {4000, 6000};
    final double[] qHigh = {1000, 1800};

    public Demand(double[] prices)
    {
        for (int i = 0; i < 2; i = i + 1)
        {
            assert(prices[i] >= pLow[i]);
            assert(prices[i] <= pHigh[i]);
            quantities[i] = qHigh[i];
            revenues[i] = prices[i] * qHigh[i];
            if (prices[i] <= pMid[i]) {
                     quantities[i] = qLow[i];
                     revenues[i] = prices[i] * qLow[i];}}
    }
}
class Manufacturer
{
    public double[] materials = {0, 0};
    public double cost = 0;
    final double[] reqs0 = {2.3, 2.1};
    final double[] reqs1 = {3.5, 0.6};
    final double[] costs = {25.3, 42.5};

    public Manufacturer(double[] products)
    {
        for (int i = 0; i < 2; i = i + 1) {
            materials[i] = reqs0[i] * products[0] + reqs1[i] * products[1];
            cost = cost + costs[i] * materials[i];}
    }
}
class Supplier
{
    public double cost = 0;
    inal double[] unitCosts = {3.0, 14.3};
    final double discount = 0.5;

    public Supplier(double[] materials)
    {
        for (int i = 0; i < 2; i = i + 1) {cost = cost + materials[i] * unitCosts[i];}
        if (cost > 2000) {cost = 2000 + (cost - 2000) * 0.5;}
    }
}
```

Fig. 2. Classes for Demand, Manufacturer, Supplier

long as the program computes a predictable number of values, we can identify a correspondence between values across all program paths.

To formulate the restrictions, we use the notion of *nondeterministic values*, or ND-values for short, which we define recursively as follows.

- The output of a `choice` method is a ND-value
- A variable is a ND-value, if it appears on the left-hand side of an assignment with a ND-value on the right-hand side.
- A variable is a ND-value, if it appears on the left-hand side of an assignment that appears in the THEN or ELSE part of a conditional statement, where the Boolean condition is a ND-value.
- The result of an arithmetic or Boolean operation on one or more ND-values is a ND-value.

We also say that a conditional statement is ND, if its Boolean condition is ND. A LOOP statement is ND, if its exit Boolean condition is ND. A method call is ND, if it is done from within a ND conditional statement. We also say that a variable, expression, conditional statement etc. are *deterministic* to mean the negation of being *nondeterministic*.

The following simple restrictions are imposed in order to make the number of values computed by the program independent of the nondeterministic choices, and associate at most one `checkObjective` method call with each `choice` call and `assert` statement.

- No ND loops
- No ND recursive method calls
- No ND calls for `checkObjective`.

The first restriction controls the number of iterations of each loop. As long as the loop's exit conditions are deterministic, the loop will continue for a deterministic number of iterations. If a loop executed a nondeterministic number of iterations, it would compute a nondeterministic number of values.

The second restriction controls the depth of recursive calls. By prohibiting recursive calls within nondeterministic conditionals, we prevent the depth of recursion from depending on nondeterministic choices. We do allow arbitrary non-recursive method calls, whether or not they are deterministic, and also recursive method calls as long as they are deterministic. Note that the conditions above are sufficient, but not necessary, to control the number of states in the program. More flexible conditions are subject for further work.

The third restriction, namely that no `checkObjective` is called from a ND conditional statement, makes sure that per given input to the program, (1) all `checkObjective` method calls have a total ordering, which is deterministic, and (2) that every execution path that goes through a specific `choice` or `assert` statement will deterministically "continue" to a unique "nearest" `checkObjective` call (if there is any). In this case, we say that such a `choice` or `assert` statement is in the *scope* of that nearest `checkObjective` call.

We are now ready to define optimization semantics in a general way.

CoJava Semantics

A sequence of specific selections in nondeterministic choice statements corresponds to an *execution path*. We define a *feasible* execution path as one that satisfies (1) the range conditions in the choice statements, and (2) the assert-constraint statements. An *optimal* execution path is a *feasible* path that produces the optimal value in a designated program variable, among all *feasible* execution paths.

Case 1: Single checkObjective Call, as Last Program Statement. We assume here that all the restrictions outlined are satisfied. Given a CoJava program P, input I, and the `checkObjective(v)` statement S as the last program statement, we denote by EP the set of all feasible execution paths e, i.e., execution paths that reach S. For a particular feasible execution path $e \in EP$, we denote by $v(e)$ the value of the program variable v at the statement S.

An *optimal* execution path is a solution to the following optimization problem OP:

$$Optimize \;\; v(e) \; s.t. \;\; e \in EP$$

where *Optimize* stands for *Minimize* in the case of `checkMinObjective` and for *Maximize* in the case of `checkMaxObjective`.

Note that a solution to this problem may not be unique, as more than one feasible execution path $e \in EP$ may have the minimal/maximal $v(e)$. An optimal execution path e defines the values for each execution of a `choice` method.

An execution of the program P according to CoJava semantics is a regular procedural execution where the values returned by each `choice` statement are those corresponding to an optimal execution path e.

Case 2: No checkObjective in the program. In this case, consider a satisfaction problem SP: Find $e \in EP$, where EP is the set of all *feasible* execution paths, i.e., those that reach a special `NOOP` method (which does nothing) at the end of the program.

An execution of the program P according to CoJava semantics is a regular procedural execution where the values returned by each `choice` statement are those corresponding to a feasible execution path e.

Case 3: One or More checkObjective Calls According to Restrictions. We do not discuss this general case in detail in this paper, nor was it implemented. Here we only provide a general idea. Because of the restrictions, (1) all `checkObjective` method calls have a total ordering, which is deterministic, and (2) every execution path that goes through a specific `choice` or `assert` statement will deterministically “continue” to a unique “nearest” `checkObjective` call (if there is any). In this case, we say that such a `choice` or `assert` statement is in the *scope* of that nearest `checkObjective` call.

The idea here, is to consider an execution as split into sections, in the order of `checkObjective` calls, each with the `choice` and `assert` statements in its scope, and apply Case 1 on all but the last section, and Case 2 on the last.

3 Reduction to Standard Constraint Optimization Formulation

The semantics of CoJava was precisely defined in the previous section. A CoJava program behaves exactly like the corresponding Java program invoked with an optimal assignment to its nondeterministic-choice inputs. While this definition is correct, it begs the question of how such an assignment can be identified.

This section describes how a CoJava program can be reduced to a set of constraints, decision variables, and an objective function (in other words, to a conventional optimization problem). The set of constraints is a declarative description of the set of legal execution paths of the Java program. Every consistent assignment of decision variables defines one legal execution path of the Java program. The assignment producing the optimal value of the objective function corresponds to the optimal nondeterministic-choice input to the program.

If the resulting optimization problem can be solved, the optimal execution path can be identified, and the CoJava program can be correctly executed. However, the decision problem may not be solvable by any existing optimization algorithm. In practice, only relatively simple CoJava programs using a restricted set of numeric operators can be optimized and correctly executed.

Intuitively, a CoJava program is reduced to constraints by encoding with constraint variables the states of program execution, and by encoding with constraints the valid transitions from one program state to the next state. To illustrate the reduction, the following subsection shows the constraint encoding for the supply chain example. The final subsection defines the reduction to constraints more precisely.

Constraints for the Supply Chain Example

Initial program state is encoded by a constraint store CS containing the empty conjunction of constraints, i.e., TRUE. Assignments with the `choice` method on the right hand side are encoded as range constraints, within which a nondeterministic choice can be made. Thus, after the first two assignments, the new program state is represented by the constraint variables $price_1$ and $price_2$, which represent the values in the program variables `prices[0]` and `prices[1]` after the assignments. The constraints:

$$\begin{gathered} 100.0 \leq price_1 \leq 200.0 \\ 105.0 \leq price_2 \leq 205.0 \end{gathered}$$

are added to the constraint store CS.

Next, the three object constructor procedures are invoked. In these procedures, several instance variables are assigned, and these create several constraints and constraint variables. These variables and constraints are omitted to keep this example concise.

Next, an initial value is assigned to the program variable `revenue`. The constraint $revenue_1 = 0$ is added to the constraint store CS.

Next, the loop executes exactly two iterations. On each iteration, a new constraint and new constraint variable are introduced. The constraints:

$$\begin{aligned} revenue_2 &= revenue_1 + demand_2 \\ revenue_3 &= revenue_2 + demand_4 \end{aligned}$$

are added to the constraint store CS. Note: the constraint variables $demand_2$ and $demand_4$ represent values defined within the constructor calls, which are omitted from this example presentation.

Similarly, the last two assignments introduce two more constraints. The constraint store CS now contains:

$$\begin{aligned} &CS = \\ &price_1 >= 100.0 \ \wedge \ price_1 <= 200.0 \wedge \\ &price_2 >= 105.0 \ \wedge \ price_2 <= 205.0 \wedge \\ &revenue_1 = 0 \wedge \\ &revenue_2 = revenue_1 + demand_2 \wedge \\ &revenue_3 = revenue_2 + demand_4 \wedge \\ &profit_1 = revenue_3 - cost \wedge \\ &revenues_2 = ... \wedge revenue_4 = ... \wedge cost = ... \end{aligned}$$

Finally, the `checkMaxObjective(profit)` statement is encoded as the decision optimization problem:

$$Maximize \ \ profit_1 \ s.t. \ \ CS$$

In fact, if we ran this problem using an LP solver, we would get the answer $price_1 = 200$, $price_2 = 185$, $profit_1 = 401555.00$.

Reduction Procedure

In this subsection we briefly describe how a decision optimization problem in terms of constraint variables, constraints and an objective function is formulated, so that it would be equivalent to the optimization problem in the *optimization* semantics of CoJava. To do that, we conceptually describe a modified Java program that generates symbolic constraints, to be used in the optimization problem.

First, we introduce symbolic expression types, for *arithmetic numeric* and *Boolean* types. This is done by implementing the `symbExpression` class. This class has methods which are their arithmetic counterparts: add, subtract, multiply etc., that construct more complex arithmetic symbolic expressions from simpler ones.

Similarly, symbolic atomic constraints correspond to inequalities or equations between two symbolic arithmetic expressions. Symbolic constraints are either atomic, or constructed using Boolean operators of simpler symbolic constraints in the standard fashion.

The Java code of the CoJava program is modified as follows. All numeric and Boolean types are replaced with their symbolic counterparts, and so are the operators. Some program statements extend CS (i.e., conjuncts it with additional constraints). Initially, constraint store CS is empty. Then:

– For an assignment statement of the form `v = AE`, where AE is an arithmetic expression, a new constraint variable v_{new} is generated for the program variable v, a symbolic arithmetic expression *symb(AE)* is created for the RHS `AE`, and the constraint $v_{new} = symb(AE)$ is added to the constraint store CS. Also, a program variable v is converted to the type `symbExpression` in the modified program.
– A conditional statement of the form

```
if C { S1 } else { S2 }
```

where `C` is a nondeterministic (ND) Boolean condition, and `S1` and `S2` are statements, is first replaced with two conditional statements

```
if C { S1 }; if not C { S2 }
```

– For a conditional statement of the form `if C { S }` , including those generated from `if ... else ...` statement, the following constraints are added to CS:

$$\begin{array}{lcl} symb(C) & \longrightarrow & Constraints(S) \\ \neg symb(C) & \longrightarrow & Equalities \end{array}$$

where *Constraints(S)* stand for the constraints that would be generated for the statement `S` if it was executed unconditionally, and *Equalities* is the set of equality constraints of the form $v_{new} = v_{old}$, where v_{new} is the newest constraint variable v_{new} generated for a program variable `v` in *Constraints(S)*, and v_{old} is the last constraint variable for `v` before the conditional statement. The implication constraint is then emulated using binary variables and linear constraints; we omit the precise description due to lack of space.
– For an assignment `v = choice(min,max)`, a constraint $min \leq v_{new} \leq max$ is generated and added to CS
– For an `assert(C)` statement a constraint *symb(C)* is generated and added to CS

For Case 1 of optimization semantics, where `checkObjective(objective)` is the last statement of the program, the optimization problem constructed is:

$$Optimize \quad objective_{current} \quad s.t. CS$$

where *Optimize* stands for *Minimize* in the case of `checkMinObjective` and *Maximize* in the case of `checkMaxObjective`, and $objective_{current}$ is the most recent constraint variable for the program variable `objective`.

4 Overview of Implementation

We have developed a *constraint compiler* for the the CoJava language. The constraint compiler translates a nondeterministic simulation procedure into an equivalent decision problem. The input is a program in the CoJava (our restricted version of Java). The resulting decision problem consists of a set of equations and inequalities in the mathematical modeling language AMPL.

The overall flow of the constraint compiler is as follows: First, a simulation procedure is made nondeterministic by initializing it with values from the nondeterministic choice library, and designating its output as an objective value. This requires no change to the procedure itself, only to its parameters and return value. Next, the procedure is transformed to create a constraint generator procedure. This involves uniformly converting all of its numeric data types to symbolic expression data types. Next, the constraint generator is compiled and executed (using a standard java compiler). The result generated by this procedure is a set of symbolic expression data structures, represent the nondeterministic output of the simulation procedure. Finally, these symbolic expressions are translated into a standard constraint programming language such as AMPL.

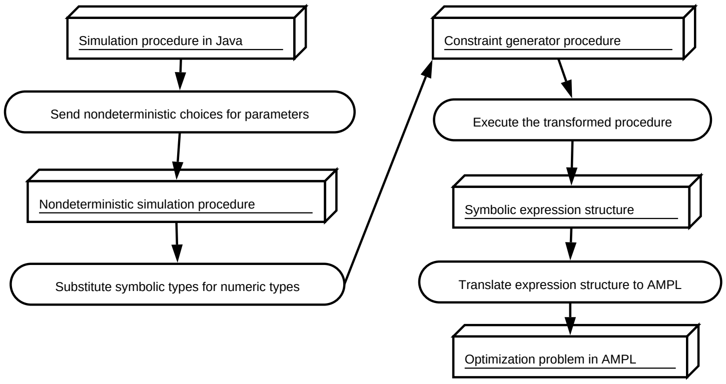

Simulation procedure in Java
Send nondeterministic choices for parameters
Nondeterministic simulation procedure
Substitute symbolic types for numeric types
Constraint generator procedure
Execute the transformed procedure
Symbolic expression structure
Translate expression structure to AMPL
Optimization problem in AMPL

Fig. 3. Diagram of the Constraint Compiler

5 Conclusions and Future Work

We presented a unified language with complementing procedural and optimization semantics. Interesting questions remain for future research. This includes: (1) how to generalize CoJava to serve as a general computational paradigm; (2) how to extend it with intelligent debugging capability (including testing whether generated constraints fall into specific classes supported by external solvers); (3) how to add CP facilities and extend underlying constraint solvers, and (4) experimenting with using CoJava on diverse optimization applications. Finally, we would like to thank anonymous reviewers for their insightful suggestions and Garrett Kaminski for his helpful comments.

References

1. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
2. Gosling, J., Joy, B., , Steele, G.: The Java language specication. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)
3. Dahl, O.J., Nygaard, K.: Simula: an algol-based simulation language. Commun. ACM **9** (1966) 671–678
4. Thomasma, T., Madsen, J.: Object oriented programming languages for developing simulation-related software. In: WSC' 90: Proceedings of the 22nd conference on Winter simulation, Piscataway, NJ, USA, IEEE Press (1990) 482–485
5. Healy, K.J., Kilgore, R.A.: Introduction to silk and java-based simulation. In: WSC '98: Proceedings of the 30th conference on Winter simulation, Los Alamitos, CA, USA, IEEE Computer Society Press (1998) 327–334
6. P. Bizarro, L.M.S., Silva, J.G.: Jwarp: A java library for parallel discrete-event simulations. In: Proceedings of the ACM Workshop on Java for High- Performance Network Computing, 1998. (1998) 999–1005
7. Fourer, R., Gay, D.M., Kernighan, B.W.: A modeling language for mathematical programming. Manage. Sci. **36** (1990) 519–554
8. Boisvert, R.F., Howe, S.E., Kahaner, D.K.: Gams: a framework for the management of scientific software. ACM Trans. Math. Softw. **11** (1985) 313–355
9. Hentenryck, P.V., Michel, L., Perron, L., R&#233;gin, J.C.: Constraint programming in opl. In: PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming, London, UK, Springer-Verlag (1999) 98–116
10. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1987) 111–119
11. Caseau, Y., Josset, F., Laburthe, F.: CLAIRE: combining sets, search and rules to better express algorithms. Theory and Practice of Logic Programming **2** (2002) 769–805
12. Smolka, G., Henz, M., Wurtz, J.: Object-oriented concurrent constraint programming in Oz. Principles and Practice of Constraint Programming (1995) 29–48
13. Wallace, M., Novello, S., Schimpf, J.: ECLiPSe: A platform for constraint logic programming. ICL Systems Journal **12** (1997) 159–200
14. Puget, J.F., Leconte, M.: Beyond the glass box: Constraints as objects. In: International Logic Programming Symposium. (1995) 513–527
15. Apt, K., Schaerf, A.: Search and imperative programming. In: Conference Record of POPL 97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, New York, NY (1997) 67–79
16. Jayaraman, B., Tambay, P.: Semantics and applications of constrained objects. Technical Report 2001-15 (2001)
17. Horn, B.L.: Siri : a constrained-object language for reactive program implementation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., USA (1991)
18. Puget, J.F.: A c++ implementation of clp. In: Proceedings of SPICIS. (1994)
19. Laburthe, F.: Choco: Implementing a cp kernel. In: CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS), Singapour. (2000) 71–85

20. Schulte, C., Tack, G.: Views and iterators for generic constraint implementations. In: Recent Advances in Constraints (2005). Volume 3978 of Lecture Notes in Computer Science., Springer-Verlag (2006) 118–132
21. Brodsky, A., Segal, V.E., Chen, J., Exarkhopoulo, P.A.: The ccube constraint object-oriented database system. Constraints **2** (1997) 245–277
22. Borning, A.: The programming language aspects of thinglab, a constraint-oriented simulation laboratory. ACM Trans. Program. Lang. Syst. **3** (1981) 353–387
23. Freeman-Benson, B.N.: Kaleidoscope: mixing objects, constraints, and imperative programming. In: OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (1990) 77–88
24. Fritzson, P., Engelson, V.: Modelica - a unified object-oriented language for system modelling and simulation. In: ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1998) 67–90

An Algebraic Characterisation of Complexity for Valued Constraints

David A. Cohen[1], Martin C. Cooper[2], and Peter G. Jeavons[3]

[1] Department of Computer Science, Royal Holloway, University of London, UK
d.cohen@rhul.ac.uk
[2] IRIT, University of Toulouse III, France
cooper@irit.fr
[3] Computing Laboratory, University of Oxford, UK
peter.jeavons@comlab.ox.ac.uk

Abstract. Classical constraint satisfaction is concerned with the feasibility of satisfying a collection of constraints. The extension of this framework to include optimisation is now also being investigated and a theory of so-called soft constraints is being developed. In this extended framework, tuples of values allowed by constraints are given desirability weightings, or costs, and the goal is to find the most desirable (or least cost) assignment.

The complexity of any optimisation problem depends critically on the type of function which has to be minimized. For soft constraint problems this function is a sum of cost functions chosen from some fixed set of available cost functions, known as a valued constraint language. We show in this paper that when the costs are rational numbers or infinite the complexity of a soft constraint problem is determined by certain algebraic properties of the valued constraint language, which we call feasibility polymorphisms and fractional polymorphisms.

As an immediate application of these results, we show that the existence of a non-trivial fractional polymorphism is a necessary condition for the tractability of a valued constraint language with rational or infinite costs over any finite domain (assuming P $\neq$ NP).

1 Introduction

Classical constraint satisfaction is concerned with the feasibility of satisfying a collection of constraints. The extension of this framework to include optimisation is now also being investigated and a theory of so-called *soft* constraints is being developed.

Several alternative mathematical frameworks for soft constraints have been proposed in the literature, including the very general frameworks of 'semi-ring based constraints' and 'valued constraints' [2]. For simplicity, we shall adopt the valued constraint framework here. In this framework, every tuple of values allowed by a constraint has an associated *cost*, and the goal is to find an assignment with minimal total cost.

The general constraint satisfaction problem (CSP) is NP-hard, and so is unlikely to have a polynomial-time algorithm. However, there has been much success in finding tractable fragments of the CSP by restricting the types of relation allowed in the constraints. A set of allowed relations has been called a *constraint language*. For some constraint languages the associated constraint satisfaction problems with constraints chosen from that language are solvable in polynomial-time, whilst for other constraint languages this class of problems is NP-hard [21]; these are referred to as *tractable languages* and *NP-hard languages*, respectively. Dichotomy theorems, which classify each possible constraint language as either tractable or NP-hard, have been established for constraint languages over 2-element domains [29], and 3-element domains [3]. Considerable progress has also been made in the search for a complete characterisation of the complexity of constraint languages over finite domains of arbitrary size [4, 14, 19, 21].

The general valued constraint problem (VCSP) is also NP-hard, but again we can try to identify tractable fragments by restricting the types of allowed *cost functions* that can be used to define the valued constraints. A set of allowed cost functions has been called a *valued constraint language*. Much less is known about the complexity of the optimisation problems associated with different valued constraint languages, although some results have been obtained for certain special cases. In particular, a complete characterisation of complexity has been obtained for valued constraint languages over a 2-element domain with real-valued or infinite costs [6]. This result generalizes a number of earlier results for particular optimisation problems such as Max-Sat [11, 12].

One class of cost functions has been extensively studied: the so-called *submodular* functions. The problem of minimizing a real-valued submodular objective function occurs in many diverse application areas, including statistical physics [1], the design of electrical networks [25], and operations research [5, 33, 27]. One of the first problems to be recognized as a case of submodular function minimisation was the Max-Flow/Min-Cut problem [13]. Another class of examples arises in pure mathematics: the rank function of a matroid is always a submodular function [15]. Recently, several polynomial-time algorithms have been proposed for submodular function minimisation [17, 18, 30], although the complexity of the best of these general algorithms is $O(n^5 \min\{\log nM, n^2 \log n\})$ where M is an upper bound on the values taken by the function to be minimized [18]. More practical cubic time algorithms have been developed for many special cases [9, 24, 26], including the Max-Flow/Min-Cut problem [13], the minimization of a symmetric submodular function [28], the minimization of a $\{0, 1\}$-valued submodular function over a 2-element domain [12] and the minimization of any sum of binary submodular functions over an arbitrary finite domain [8].

The results of [12] show that submodularity is essentially the *only* property giving rise to tractable $\{0, 1\}$-valued constraint languages over a 2-element domain (see [6]). Jonsson et al. [23] recently generalized this result to 3-element domains. However, for languages allowing more general costs, or defined over larger finite domains, very little is known about the possible tractable cases.

In the classical CSP framework it has been shown that the complexity of any constraint language over any finite domain is determined by certain algebraic properties known as *polymorphisms* [19,21]. This result has reduced the problem of the identification of tractable constraint languages to that of the identification of suitable sets of polymorphisms. In other words, it has been shown to be enough to study just those constraint languages which are characterised by having a given set of polymorphisms. This both reduces the number of different languages to be studied and allows the application of results from universal algebra to the study of the complexity of constraint languages.

To analyse the complexity of valued constraint languages in the VCSP framework we recently introduced a more general algebraic property known as a *multimorphism* [6,7,10]. Using this algebraic property we have shown that there are precisely eight maximal tractable valued constraint languages over a 2-element domain with real-valued or infinite costs, and each of these is characterised by having a particular form of multimorphism [6]. Furthermore, we have shown that many known maximal tractable valued constraint languages over larger finite domains are precisely characterised by a single multimorphism and that key NP-hard examples have (essentially) no multimorphisms [7].

In this paper we slightly generalise the notion of a multimorphism to that of a *fractional polymorphism*. We are then able to show that fractional polymorphisms, together with the polymorphisms of the underlying feasibility relations (which we call *feasibility polymorphims*), characterise the complexity of any valued constraint language, Γ, with non-negative rational or infinite costs over any finite domain. Specifically we show that:

- the class of all cost functions having the same fractional polymorphisms and feasibility polymorphisms as Γ corresponds precisely to the closure of Γ by three natural extension operators (one of which is *expressibility*);
- the extended class containing Γ together with all cost functions obtained using these three extension operators has the same complexity as Γ (up to polynomial-time reduction).

This very general result has the immediate corollary that a finite-valued rational cost function is expressible over a valued constraint language if and only if it has all the fractional polymorphisms of that language.

The applications of these results to the search for tractable valued constraint languages are very similar to the applications of polymorphisms to the search for tractable constraint languages in the classical CSP. First, we need only consider valued constraint languages defined by these algebraic properties. This will greatly simplify the search for a characterisation of all tractable valued constraint languages. Secondly, by showing that there exists an NP-hard valued constraint language with only finite rational costs we show that any tractable valued constraint language with finite rational or infinite costs must have a non-trivial fractional polymorphism.

Hence the results of this paper provide a powerful new set of tools in the search for a polynomial-time/NP-hard dichotomy for finite-domain optimisation problems defined by valued constraints.

2 Valued Constraint Problems

In the valued constraint framework each constraint has an associated function which assigns a *cost* to each possible assignment of values and these costs are chosen from some *valuation structure*, satisfying the following definition.

Definition 1. *A* ***valuation structure****, Ω, is a totally ordered set, with a minimum and a maximum element (denoted 0 and ∞), together with a commutative, associative binary* ***aggregation operator*** *(denoted $\oplus$), such that for all $\alpha, \beta, \gamma \in \Omega$, $\alpha \oplus 0 = \alpha$ and $\alpha \oplus \gamma \geq \beta \oplus \gamma$ whenever $\alpha \geq \beta$.*

Definition 2. *An instance of the* ***valued constraint satisfaction problem, VCSP****, is a 4-tuple $\mathcal{P} = \langle V, D, C, \Omega \rangle$ where:*

- *V is a finite set of* ***variables;***
- *D is a finite set of possible* ***values;***
- *Ω is a valuation structure representing possible* ***costs;***
- *C is a set of* ***constraints****. Each element of C is a pair $c = \langle \sigma, \phi \rangle$ where σ is a tuple of variables called the* ***scope*** *of c, and ϕ is a mapping from $D^{|\sigma|}$ to Ω, called the* ***cost function*** *of c.*

Definition 3. *For any VCSP instance $\mathcal{P} = \langle V, D, C, \Omega \rangle$, an* ***assignment*** *for $\mathcal{P}$ is a mapping $s : V \rightarrow D$. The* ***cost*** *of an assignment s, denoted $Cost_{\mathcal{P}}(s)$, is given by the aggregation of the costs for the restrictions of s onto each constraint scope, that is,*

$$Cost_{\mathcal{P}}(s) \overset{\text{def}}{=} \bigoplus_{\langle \langle v_1, v_2, \ldots, v_m \rangle, \phi \rangle \in C} \phi(s(v_1), s(v_2), \ldots, s(v_m)).$$

A ***solution*** *to $\mathcal{P}$ is an assignment with minimal cost, and the question is to find a solution.*

Definition 4. *A* ***valued constraint language*** *is any set Γ of cost functions from some fixed finite domain D to some fixed valuation structure Ω.*

We define VCSP(Γ) *to be the set of all VCSP instances in which all cost functions belong to Γ.*

The *complexity* of a valued constraint language Γ will be identified with the complexity of the associated VCSP(Γ).

Definition 5. *A valued constraint language Γ is called* ***tractable*** *if, for every finite subset $\Gamma_f \subseteq \Gamma$, there exists an algorithm solving any instance $\mathcal{P} \in$* VCSP(Γ_f) *in time at most $p(|\mathcal{P}|)$, for some polynomial p.*

Conversely, Γ is called ***NP-hard*** *if there is some finite subset $\Gamma_f \subseteq \Gamma$ for which* VCSP(Γ_f) *is NP-hard.*

For the remainder of this paper we will focus for concreteness on the valuation structure containing the non-negative rational numbers ($\mathbb{Q}_+$) together with infinity (∞), with the standard addition operation, $+$ (extended so that $a + \infty = \infty$, for all a). This valuation structure will be denoted $\overline{\mathbb{Q}_+}$. It is sufficiently general to encode many standard optimisation problems (for examples, see [7, 10]).

3 Extending a Valued Constraint Language

We now define three ways to extend a valued constraint language. In Section 5 we will show that applying these three extensions does not alter the complexity of a valued constraint language.

What is more, in Section 6 we will give a simple algebraic characterisation of the languages obtained by applying all three extensions.

The first extension makes use of a natural equivalence relation on cost functions.

Definition 6. *Two cost functions ϕ and ϕ' are said to be* ***cost-equivalent*** *if one is obtained from the other by adding a constant cost and scaling by some constant factor. In other words, ϕ and ϕ' are cost-equivalent if they have the same arity r and there exist positive integers, a, b, and a finite constant c, such that*

$$\forall x \in D^r, \quad a\,\phi'(x) \;=\; b\,\phi(x) + c.$$

For any valued constraint language Γ the language consisting of all cost functions which are cost-equivalent to some member of Γ will be denoted $\Gamma_{\equiv}$.

The next extension adds in those cost functions which are *expressible* over Γ.

Definition 7. *For any VCSP instance $\mathcal{P} = \langle V, D, C, \Omega\rangle$, and any list $L = \langle v_1, \ldots, v_r\rangle$ of variables of $\mathcal{P}$, the* ***projection*** *of $\mathcal{P}$ onto L, denoted $\pi_L(\mathcal{P})$, is the r-ary cost function defined as follows:*

$$\pi_L(\mathcal{P})(x_1, \ldots, x_r) \;\stackrel{\text{def}}{=}\; \min_{\{s:V\to D \;\mid\; \langle s(v_1),\ldots,s(v_r)\rangle = \langle x_1,\ldots,x_r\rangle\}} Cost_{\mathcal{P}}(s)$$

We say that a cost function ϕ is ***expressible*** *over a constraint language Γ if there exists a VCSP instance $\mathcal{P} \in \mathrm{VCSP}(\Gamma)$ and a list L of variables of $\mathcal{P}$ such that $\pi_L(\mathcal{P}) = \phi$. We call the pair $\langle \mathcal{P}, L\rangle$ a* ***gadget*** *for expressing ϕ over Γ.*

For any valued constraint language Γ the language consisting of all cost functions expressible over Γ will be denoted $\exp(\Gamma)$.

Example 1. In this example we show that Γ is always a subset of $\exp(\Gamma)$.

Let Γ be a valued constraint language over a domain D with costs in Ω. Choose any $\gamma \in \Gamma$. Let the arity of γ be r and define $V = \{x_1, \ldots, x_r\}$. The pair $\langle\langle V, D, \{\langle\langle x_1, \ldots, x_r\rangle, \gamma\rangle\}, \Omega\rangle, \langle x_1, \ldots, x_r\rangle\rangle$ is a gadget for expressing γ over Γ.

The final extension adds cost functions obtained using a *feasibility* operator.

Definition 8. *For any cost function ϕ, with arity r, we denote by* $\mathrm{Feas}(\phi)$ *the r-ary cost function defined by*

$$\mathrm{Feas}(\phi)(x_1, x_2, \ldots, x_r) \;\stackrel{\text{def}}{=}\; \begin{cases} \infty & \textit{if} \quad \phi(x_1, x_2, \ldots, x_r) = \infty \\ 0 & \textit{if} \quad \phi(x_1, x_2, \ldots, x_r) < \infty. \end{cases}$$

For any valued constraint language Γ the language $\{\mathrm{Feas}(\phi) \mid \phi \in \Gamma\}$ *will be denoted* $\mathrm{Feas}(\Gamma)$.

4 Polymorphisms and Fractional Polymorphisms

For classical constraint satisfaction problems it has been shown that the complexity of a constraint language is characterised by certain algebraic properties of the relations in that language, known as *polymorphisms* [4,19,21]. This result was obtained by showing that the expressive power of a constraint language is determined by the polymorphisms of that language.

Polymorphisms are defined for *relations* rather than cost functions, but we will now define an analogous notion which can be applied to arbitrary valued constraint languages.

Definition 9. *For any r-ary cost function ϕ, we say that $f : D^k \to D$ is a k-ary* ***feasibility polymorphism*** *of ϕ if, for all $x_1, \ldots, x_r \in D^k$,*

$$\mathrm{Feas}(\phi)(f(x_1), \ldots, f(x_r)) \;\le\; \sum_{i=1}^{k} \mathrm{Feas}(\phi)(x_1[i], \ldots, x_r[i]).$$

For any valued constraint language Γ, we will say that f is a feasibility polymorphism of Γ if f is a feasibility polymorphism of every cost function in Γ. The set of all feasibility polymorphisms of Γ will be denoted $\mathrm{Pol}(\Gamma)$*, and the finite subset containing all k-ary feasibility polymorphisms will be denoted* $\mathrm{Pol}_k(\Gamma)$*.*

Theorem 1. *For any valued constraint language Γ, and any cost function ϕ,* $\mathrm{Feas}(\phi) \in \exp(\mathrm{Feas}(\Gamma))$ *if and only if* $\mathrm{Pol}(\Gamma) \subseteq \mathrm{Pol}(\{\phi\})$

Proof. Follows immediately from the corresponding result for classical constraint languages (see, for example, Corollary 1 of [22]).

To obtain a more precise result about the expressive power of arbitrary valued constraint languages we need to generalize the definition of a polymorphism.

Definition 10. *A k-ary* ***weighted function*** *F on a set D is a set of the form $\{\langle w_1, f_1\rangle, \ldots, \langle w_n, f_n\rangle\}$ where each w_i is a positive integer, and each f_i is a distinct function from D^k to D.*

For any r-ary cost function ϕ, we say that a k-ary weighted function F is a k-ary ***fractional polymorphism*** *of ϕ if, for all $x_1, \ldots, x_r \in D^k$,*

$$k \sum_{i=1}^{n} w_i \phi(f_i(x_1), \ldots, f_i(x_r)) \;\le\; \left(\sum_{i=1}^{n} w_i\right) \cdot \left(\sum_{i=1}^{k} \phi(x_1[i], \ldots, x_r[i])\right).$$

For any valued constraint language Γ, we will say that f is a fractional polymorphism of Γ if f is a fractional polymorphism of every cost function in Γ. The set of all fractional polymorphisms of Γ will be denoted $\mathrm{fPol}(\Gamma)$*.*

In earlier papers we introduced the notion of a *multimorphism* [6,7,10]. A multimorphism is precisely a k-ary fractional polymorphism where the sum of the weights w_i is exactly k.

Example 2. Consider the unary fractional polymorphism given by $\{\langle 1, c\rangle\}$ for some constant $c \in D$ (seen as a unary function). An r-ary cost function ϕ has this unary fractional polymorphism when, for all $x_1, \ldots, x_r \in D$,

$$\phi(c, \ldots, c) \ \leq \ \phi(x_1, \ldots, x_r)$$

Hence, if all cost functions $\phi \in \Gamma$ have this fractional polymorphism then we can solve any instance of VCSP(Γ) trivially, by assigning the value c to each variable. In fact, it was shown in [7, 10] that the class of all cost functions with this simple fractional polymorphism is a maximal tractable class.

Example 3. An r-ary cost function ϕ on an ordered set D has the binary fractional polymorphism $\{\langle 1, \min\rangle, \langle 1, \max\rangle\}$ when, for all $x_1, \ldots, x_r \in D^2$,

$$\begin{aligned}\phi(\min(x_1[1], x_1[2]), \ldots, \min(x_r[1], x_r[2]))& \\ + \ \phi(\max(x_1[1], x_1[2]), \ldots, \max(x_r[1], x_r[2]))& \\ \leq \ \phi(x_1[1], \ldots, x_r[1]) \ + \ \phi(x_1[2], \ldots, x_r[2])& \ .\end{aligned}$$

Hence, the fractional polymorphism $\{\langle 1, \min\rangle, \langle 1, \max\rangle\}$ exactly captures the notion of *submodularity* [15] which we can thus define as follows.

Definition 11. *A cost function (over an ordered domain) is* ***submodular*** *if and only if it has the fractional polymorphism* $\{\langle 1, \min\rangle, \langle 1, \max\rangle\}$.

Recall from the Introduction that submodular function minimization is a central problem in discrete optimization. The notion of a fractional polymorphism allows us to capture the property of submodularity by using a particular weighted function. It also allows us to generalize to many other properties by considering different weighted functions.

5 Extensions Preserve Tractability

In this section we will show that the three extensions defined in Section 3 all preserve the tractability or NP-hardness of a valued constraint language.

Theorem 2. *For any valued constraint language Γ, we have:*

1. *$\Gamma_\equiv$ is tractable if and only if Γ is tractable, and $\Gamma_\equiv$ is NP-hard if and only if Γ is NP-hard.*
2. *$\exp(\Gamma)$ is tractable if and only if Γ is tractable, and $\exp(\Gamma)$ is NP-hard if and only if Γ is NP-hard.*
3. *$\Gamma \cup \mathrm{Feas}(\Gamma)$ is tractable if and only if Γ is tractable, and $\Gamma \cup \mathrm{Feas}(\Gamma)$ is NP-hard if and only if Γ is NP-hard.*

Proof. For part (1), by Definition 5, it is sufficient to show that for any finite subset Γ' of $\Gamma_\equiv$ there exists a polynomial-time reduction from VCSP(Γ') to VCSP(Γ''), where Γ'' is a finite subset of Γ.

Let Γ' be a finite subset of $\Gamma_{\equiv}$ and let $\mathcal{P}'$ be any instance of VCSP(Γ'). By Definition 6, any cost function $\phi' \in \Gamma_{\equiv}$ is cost-equivalent to some cost function $\phi \in \Gamma$. Hence we can replace each of the constraints $\langle \sigma, \phi' \rangle$ in $\mathcal{P}'$ with a new constraint $\langle \sigma, \phi \rangle$, where $\phi \in \Gamma$ and $a\phi' = b\phi + c$ for some positive integers a, b and some (positive or negative) constant c, to obtain an instance $\mathcal{P}$ of VCSP(Γ). The constant c is added to the cost of all assignments and so does not affect the choice of solution. The effect of the scale factors a and b can be simulated by taking b copies of the new constraint in $\mathcal{P}$ and a copies of all other constraints in $\mathcal{P}$. The values of a, b are constants determined by the finite set Γ', so this construction can be carried out in polynomial time in the size of $\mathcal{P}'$.

For part (2), by Definition 5, it is sufficient to show that for any finite subset Γ' of $\exp(\Gamma)$ there exists a polynomial-time reduction from VCSP(Γ') to VCSP(Γ''), where Γ'' is a finite subset of Γ.

Let Γ' be a finite subset of $\exp(\Gamma)$ and let $\mathcal{P}'$ be any instance of VCSP(Γ'). By Definition 7, any cost function $\phi' \in \exp(\Gamma)$ can be constructed by using some gadget $\langle \mathcal{P}_{\phi'}, L \rangle$ where $\mathcal{P}_{\phi'}$ is an instance of VCSP(Γ). Hence we can simply replace each constraint in $\mathcal{P}'$ which has a cost function ϕ' not already in Γ with the corresponding gadget to obtain an instance $\mathcal{P}$ of VCSP(Γ) which is equivalent to $\mathcal{P}'$. The maximum size of any of the gadgets used is a constant determined by the finite set Γ', so this construction can be carried out in polynomial time in the size of $\mathcal{P}'$.

For part (3), by Definition 5, it is sufficient to show that for any finite subset Γ' of $\Gamma \cup \mathrm{Feas}(\Gamma)$ there exists a polynomial-time reduction from VCSP(Γ') to VCSP(Γ''), where Γ'' is a finite subset of Γ.

Let Γ' be a finite subset of $\Gamma \cup \mathrm{Feas}(\Gamma)$ and let $\mathcal{P}'$ be any instance of VCSP(Γ'). By Definition 8, any cost function $\phi' \in \mathrm{Feas}(\Gamma)$ is obtained from some cost function $\phi \in \Gamma$ by setting all finite values to 0. Now assume that $\mathcal{P}'$ has k constraints with cost functions in Feas(Γ). If we replace each of these constraints $\langle \sigma, \phi' \rangle$ with a new constraint $\langle \sigma, \phi \rangle$, where $\phi \in \Gamma$ and $\mathrm{Feas}(\phi) = \phi'$, then we obtain an instance $\mathcal{P}$ of VCSP(Γ).

Let M be the maximum finite value taken by any cost function in the finite set Γ', and let m be the minimum difference between any two distinct finite values taken on by cost functions in Γ'. The cost of any assignment for $\mathcal{P}$ differs by at most kM from the cost of the same assignment for $\mathcal{P}'$. Hence if we also replace all the remaining constraints $\langle \sigma, \phi \rangle$ of $\mathcal{P}'$ with $\lceil \frac{Mk}{m} + 1 \rceil$ copies of $\langle \sigma, \phi \rangle$, then we obtain an instance of VCSP(Γ) with the same solutions as $\mathcal{P}'$. Since M and m are constants determined by the finite set Γ', this construction can be carried out in polynomial time in the size of $\mathcal{P}'$.

We can now combine all three extensions to obtain the following result.

Definition 12. *For any valued constraint language, Γ, we define the* ***closure*** *of Γ, denoted $\widehat{\Gamma}$, as follows:*

$$\widehat{\Gamma} \overset{\mathrm{def}}{=} (\exp(\Gamma \cup \mathrm{Feas}(\Gamma)))_{\equiv}.$$

Corollary 1. *A valued constraint language Γ is tractable if and only if $\widehat{\Gamma}$ is tractable; similarly, Γ is NP-hard if and only if $\widehat{\Gamma}$ is NP-hard.*

6 Characterising $\widehat{\Gamma}$

The main result of this paper is the following theorem, which characterises the extended language $\widehat{\Gamma}$ in terms of the feasibility polymorphisms and fractional polymorphisms of Γ.

Theorem 3. *For any valued constraint language Γ with costs in $\overline{\mathbb{Q}}_+$, and any cost function ϕ taking values in $\overline{\mathbb{Q}}_+$, $\phi \in \widehat{\Gamma}$ if and only if* $\mathrm{Pol}(\Gamma) \subseteq \mathrm{Pol}(\{\phi\})$ *and* $\mathrm{fPol}(\Gamma) \subseteq \mathrm{fPol}(\{\phi\})$.

The following result is an immediate consequence of Corollary 1 and Theorem 3.

Corollary 2. *The tractability or NP-hardness of a valued constraint language Γ with costs in $\overline{\mathbb{Q}}_+$ is determined by its feasibility polymorphisms and fractional polymorphisms.*

We also observe that when the cost functions in Γ take finite rational values only, the tractability or NP-hardness is determined by the fractional polymorphisms alone. Conversely, when $\Gamma = \mathrm{Feas}(\Gamma)$ the tractability or NP-hardness is determined by the feasibility polymorphisms alone.

We will prove Theorem 3 in two halves. First we show, in Proposition 1, that the feasibility polymorphisms and fractional polymorphisms of Γ are preserved by all members of $\widehat{\Gamma}$. Then we show, in Theorem 4, that every cost function with all the feasibility polymorphisms and fractional polymorphisms of Γ is in fact a member of $\widehat{\Gamma}$.

Proposition 1. *If Γ is any valued constraint language then* $\mathrm{Pol}(\Gamma) = \mathrm{Pol}(\widehat{\Gamma})$ *and* $\mathrm{fPol}(\Gamma) = \mathrm{fPol}(\widehat{\Gamma})$.

Proof. This follows immediately from the fact that feasibility polymorphisms and fractional polymorphisms are preserved by aggregating cost functions, projecting onto subsets of variables, adding constants, scaling by a natural number, and applying the feasibility operator.

To see this note that if ϕ_1 and ϕ_2 both satisfy the inequality in Definition 9, then so does any extension of ϕ_1 and ϕ_2 obtained by adding dummy arguments. So also does $\phi_1 \oplus \phi_2$ (by the monotonicity of the $\oplus$ operation). So also does the projection of each ϕ_i onto any list of arguments. Hence if Γ has the feasibility polymorphism f, then so does any cost function expressible over Γ. Furthermore adding a constant to ϕ preserves this inequality, and scaling by a natural number also preserves the inequality. Finally replacing ϕ_i with $\mathrm{Feas}(\phi_i)$ also preserves this inequality.

Similar remarks apply to the inequality in Definition 10.

To establish Theorem 4 below we will use the following result, which is a variant of the well-known Farkas' Lemma used in linear programming [27, 31].

Lemma 1 (Farkas 1894). *Let S and T be finite sets of indices, where T is the disjoint union of two subsets, $T_{\geq}$ and $T_{=}$. For all $i \in S$, and all $j \in T$, let $a_{i,j}$ and b_j be rational numbers. Exactly one of the following holds:*

– *Either there exists a set of non-negative rational numbers* $\{x_i \mid i \in S\}$ *and a rational number* C *such that*

$$\begin{aligned} \textit{for each } j \in T_{\geq}, \quad \sum_{i \in S} a_{i,j}\, x_i &\geq b_j + C, \qquad \textit{and,} \\ \textit{for each } j \in T_{=}, \quad \sum_{i \in S} a_{i,j}\, x_i &= b_j + C. \end{aligned}$$

– *Or else there exists a set of integers* $\{y_j \mid j \in T\}$ *such that* $\sum_{j \in T} y_j = 0$ *and:*

$$\begin{aligned} \textit{for each } j \in T_{\geq}, \quad y_j &\geq 0, \\ \textit{for each } i \in S, \quad \sum_{j \in T} y_j\, a_{i,j} &\leq 0, \qquad \textit{and} \\ \sum_{j \in T} y_j\, b_j &> 0. \end{aligned}$$

Such a set is called a ***certificate of unsolvability.***

The proof of Theorem 4 also uses a number of constructions related to the notion of an *indicator problem*, as introduced in [20, 22].

Definition 13. *A* ***k-matching*** *of a valued constraint language* Γ *is defined to be a pair* $\langle M, \gamma \rangle$ *where*

– γ *is a cost function in* Γ *with arity* r*, and*
– M *is a* $k \times r$ *matrix of elements of* D *such that* γ *has a finite value when applied to any of the* k *rows.*

Definition 14. *A* ***k-weighting****,* X*, of a valued constraint language* Γ *is defined to be a mapping from the set of all* k*-matchings of* Γ *to the non-negative integers.*

Note that a k-weighting of Γ can be seen as associating a multiplicity (possibly zero) with each k-matching.

Definition 15. *Given a* finite *valued constraint language* Γ *over a finite set* D *and a* k*-weighting,* X*, of* Γ*, we define the* ***X-weighted indicator problem*** *over* Γ*, denoted* $\mathcal{IP}(\Gamma, X)$*, as follows:*

– *The set of variables of* $\mathcal{IP}(\Gamma, X)$ *is the set* D^k *consisting of all* k*-tuples of elements from* D*.*
– *The domain of* $\mathcal{IP}(\Gamma, X)$ *is the domain* D *of* Γ*.*
– *The constraints of* $\mathcal{IP}(\Gamma, X)$ *are defined as follows. Note that any list of* r *variables,* $v_1, \ldots, v_r$*, can be seen as (the columns of) a* $k \times r$ *matrix. For every list* S *of variables, if* $\langle S, \gamma \rangle$ *is a* k*-matching of* Γ*, then* $\mathcal{IP}(\Gamma, X)$ *has the constraint* $\langle S, \mathrm{Feas}(\gamma) \rangle$ *and* $X(\langle S, \gamma \rangle)$ *copies of the constraint* $\langle S, \gamma \rangle$*.*

Theorem 4. *Let* Γ *be a* finite *valued constraint language over a finite set* D *with costs in* $\overline{\mathbb{Q}}_+$*, and let* $\phi : D^r \to \overline{\mathbb{Q}}_+$ *be any cost function such that* $\mathrm{Pol}(\Gamma) \subseteq \mathrm{Pol}(\{\phi\})$*.*

Either $\phi \in \widehat{\Gamma}$*, or else there is some fractional polymorphism of* Γ *which is not a fractional polymorphism of* ϕ*.*

Proof. The idea of the proof is as follows. We will attempt to construct a weighted indicator problem to express a cost function ϕ' which is cost-equivalent to ϕ. If this succeeds, then we have shown that $\phi \in \widehat{\Gamma}$, since every weighted indicator problem for Γ is an instance of VCSP(exp($\Gamma \cup$ Feas(Γ))).

On the other hand, if this fails then we will show that we must have an unsatisfiable collection of equations and inequations. We will then use Lemma 1, to get a certificate of insolvability. This certificate will give us the required fractional polymorphism of Γ that is not a fractional polymorphism of ϕ.

We now give the details of the proof. Let k be the number of r-tuples for which the value of ϕ is finite and fix an arbitrary order, $\langle x_1, \ldots, x_k \rangle$, for these tuples. This list of tuples can be viewed as (the rows of) a matrix with k rows and r columns, which we will call S_ϕ.

Note that (the columns of) S_ϕ can be viewed as a list $\langle s_1, \ldots, s_r \rangle$ of k-tuples, and hence as a list of variables of an indicator problem. We will now try to find some k-weighting X of Γ so that the X-weighted indicator problem $\mathcal{IP}(\Gamma, X)$ can be used to express a cost function ϕ' which is cost-equivalent to ϕ. More precisely, we will seek to find a k-weighting X such that $\langle \mathcal{IP}(\Gamma, X), S_\phi \rangle$ is a gadget for expressing such a ϕ'.

The variables of $\mathcal{IP}(\Gamma, X)$ are the possible k-tuples over D, so each assignment to these variables can be viewed as a function $f : D^k \to D$. Whatever k-weighting X we choose, the set of assignments for $\mathcal{IP}(\Gamma, X)$ which have infinite cost is the same as for the classical indicator problem for Feas(Γ) of order k, as defined in [20, 22]. Hence, by Theorem 1 of [22], every assignment for $\mathcal{IP}(\Gamma, X)$ which has a finite cost corresponds to a feasibility polymorphism of Γ. Since we are assuming that ϕ has all of the feasibility polymorphisms of Γ, it follows that $\langle \mathcal{IP}(\Gamma, X), S_\phi \rangle$ is a gadget for some r-ary cost function which is finite-valued exactly when ϕ is finite-valued.

Now consider any $f : D^k \to D \in \mathrm{Pol}_k(\Gamma)$. By Definition 9, we know that, for any k-matching $\langle S, \gamma \rangle$ of Γ, we must have $\gamma(f(S)) < \infty$, where $f(S)$ denotes the tuple of values obtained by applying f to each column of S. Since ϕ has all the feasibility polymorphisms of Γ, we also have that $\phi(f(S_\phi)) < \infty$.

We now define a finite system of inequalities and equations, with finite coefficients, which together specify the required properties for an unknown constant C and k-weighting X, to ensure that $\langle \mathcal{IP}(\Gamma, X), S_\phi \rangle$ is a gadget for $\phi + C$.

For each $f \in \mathrm{Pol}_k(\Gamma)$,

$$\sum_{\gamma \in \Gamma} \sum_{\{\text{all } k\text{-matchings } \langle S, \gamma \rangle\}} X(\langle S, \gamma \rangle)\, \gamma(f(S)) \quad \geq \quad \phi(f(S_\phi)) + C. \quad (1)$$

For each projection $e \in \mathrm{Pol}_k(\Gamma)$,

$$\sum_{\gamma \in \Gamma} \sum_{\{\text{all } k\text{-matchings } \langle S, \gamma \rangle\}} X(\langle S, \gamma \rangle)\, \gamma(e(S)) \quad = \quad \phi(e(S_\phi)) + C. \quad (2)$$

We claim that if a non-negative rational solution X to this system of inequalities and equations exists, then $\langle \mathcal{IP}(\Gamma, X), S_\phi \rangle$ is a gadget for the cost function $\phi + C$.

To see this, note that the set of inequalities imply that the value of the projection of $\mathcal{IP}(\Gamma, X)$ onto the list of variables S_ϕ must be at least as great as the value of $\phi + C$, for all possible assignments. Furthermore, from the set of equations, and the choice of S_ϕ, it follows that whenever $\phi + C$ is finite, the value of the projection of $\mathcal{IP}(\Gamma, X)$ onto the list of variables S_ϕ must be the same as $\phi + C$.

By applying a suitable scale factor to the solution obtained, we can choose an *integer*-valued X such that $\langle \mathcal{IP}(\Gamma, X), S_\phi \rangle$ is a gadget for some cost function which is cost-equivalent to ϕ.

On the other hand, if this system of equations and inequalities has no solution, then we appeal to Lemma 1, to get a certificate of insolvability. That is, in this case we know that there exists a set of integers $\{y_f \mid f \in \mathrm{Pol}_k(\Gamma)\}$, such that $\sum_{f \in \mathrm{Pol}_k(\Gamma)} y_f = 0$, $y_f \geq 0$ when f is not a projection, and:

$$\text{for each } k\text{-matching } \langle S, \gamma \rangle \text{ of } \Gamma, \quad \sum_{f \in \mathrm{Pol}_k(\Gamma)} y_f \; \gamma(f(S)) \;\leq\; 0, \quad \text{and} \tag{3}$$

$$\sum_{f \in \mathrm{Pol}_k(\Gamma)} y_f \; \phi(f(S_\phi)) \;>\; 0 \tag{4}$$

Let $m = \min\{y_f \mid f \text{ is a projection }\}$. Since $\sum_{f \in \mathrm{Pol}_k(\Gamma)} y_f = 0$, we know that $m < 0$.

Define a set of integers $\{z_f \mid f \in \mathrm{Pol}_k(\Gamma)\}$ as follows:

$$z_f = \begin{cases} y_f - m & \text{if } f \text{ is a projection} \\ y_f & \text{otherwise} \end{cases}$$

Now we have that each $z_f \geq 0$, and that $\sum_{f \in \mathrm{Pol}_k(\Gamma)} z_f = k|m|$.

Let E be the set of all k-ary projections. It follows (rewriting Equation 3) that for any k-matching $\langle S, \gamma \rangle$ of Γ

$$|m| \sum_{e \in E} \gamma(e(S)) \quad \geq \sum_{f \in \mathrm{Pol}_k(\Gamma)} z_f \; \gamma(f(S))$$

Moreover, if S is any $k \times r$ matrix for which $\langle S, \gamma \rangle$ is not a k-matching of Γ, then $\sum_{e \in E} \gamma(e(S)) = \infty$. Hence, for any set of k-tuples $x_1, \ldots, x_r$ we get that

$$\left(\sum_{f \in \mathrm{Pol}_k(\Gamma)} z_f \right) \cdot \left(\sum_{i=1}^{k} \gamma(x_1[i], \ldots, x_r[i]) \right) \geq \; k \sum_{f \in \mathrm{Pol}_k(\Gamma)} z_f \; \gamma(f(x_1), \ldots, f(x_r))$$

which precisely states that the k-ary weighted function $\{\langle z_f, f \rangle \mid f \in \mathrm{Pol}_k(\Gamma)\}$ is a fractional polymorphism of Γ.

On the other hand, rewriting Equation 4 in the same way gives:

$$\left(\sum_{f \in \mathrm{Pol}_k(\Gamma)} z_f \right) \cdot \left(\sum_{i=1}^{k} \phi(s_1[i], \ldots, s_r[i]) \right) < \; k \sum_{f \in \mathrm{Pol}_k(\Gamma)} z_f \; \phi(f(s_1), \ldots, f(s_r))$$

where $s_1, \ldots, s_r$ are the columns of S_ϕ. This provides a witness that the weighted function $\{\langle z_f, f \rangle \mid f \in \mathrm{Pol}_k(\Gamma)\}$ is *not* a fractional polymorphism of ϕ.

7 A Necessary Condition for Tractability

In this section, we exhibit a well-known (finite-valued) intractable cost function $\phi_{\neq}$ and use it to establish a necessary condition for a valued constraint language to be tractable.

Definition 16. *Define the binary cost function $\phi_{\neq} : D^2 \to \mathbb{Q}_+$ as follows:*

$$\phi_{\neq}(x,y) \;\overset{\text{def}}{=}\; \begin{cases} 1 \textit{ if } \; x = y \\ 0 \textit{ otherwise} \end{cases}$$

The cost function $\phi_{\neq}(x,y)$ penalizes the assignment of the same value to its two arguments.

Lemma 2. *For any set D with $|D| \geq 2$,* VCSP$(\{\phi_{\neq}\})$ *is NP-hard.*

Proof. For $|D| = 2$, this follows immediately from the fact that the version of Max-Sat consisting of only XOR constraints is NP-hard [11].

For $|D| \geq 3$, a polynomial-time algorithm to solve VCSP$(\{\phi_{\neq}\})$ would immediately provide a polynomial-time algorithm to determine whether a graph has a $|D|$-colouring, which is a well-known NP-complete problem [16].

Definition 17. *A k-ary* ***fractional projection*** *is a k-ary weighted function $\{\langle n, \pi_1\rangle, \ldots, \langle n, \pi_k\rangle\}$ where n is a constant and each π_i is the projection that returns its ith argument.*

Lemma 3. *For any cost function ϕ,* fPol$(\{\phi\})$ *contains all fractional projections.*

Proof. Consider the inequality in Definition 10 defining a fractional polymorphism. All fractional projections satisfy this inequality (with equality), so they are all fractional polymorphisms of any cost function ϕ.

Theorem 5. *If all fractional polymorphisms of a valued constraint language Γ are fractional projections, then Γ is NP-hard.*

Proof. Suppose that every fractional polymorphism of Γ is a fractional projection. By Lemma 3, we have that fPol$(\Gamma) \subseteq$ fPol$(\{\phi_{\neq}\})$.

Since Feas$(\phi_{\neq})$ is the cost function whose costs are all zero, it follows that $\phi_{\neq}$ has all possible feasibility polymorphisms. Hence Pol$(\Gamma) \subseteq$ Pol$(\{\phi_{\neq}\})$.

By Theorem 3, we have $\phi_{\neq} \in \widehat{\Gamma}$, so by Lemma 2, $\widehat{\Gamma}$ is NP-hard. Hence, by Corollary 1, it follows that Γ is also NP-hard.

Hence, assuming that P $\neq$ NP, we have that any tractable valued constraint language must have some fractional polymorphism which is not a fractional projection.

8 Conclusion

We have shown that the complexity of any valued constraint language with rational-valued or infinite costs is determined by certain algebraic properties of the cost functions, which we have identified as feasibility polymorphisms and fractional polymorphisms.

When the cost functions can only take on the values zero or infinity, the optimisation problem VCSP collapses to the classical constraint satisfaction problem, CSP. In previous papers we have shown that the existence of a non-trivial polymorphism is a necessary condition for tractability in this case [19,21]. This result sparked considerable activity in the search for and characterization of tractable constraint languages [3, 4, 21]. We hope that the results in this paper will provide a similar impetus for the characterization of tractable valued constraint languages using algebraic methods.

Of course there are still many open questions concerning the complexity of valued constraint satisfaction problems. In particular, it will be interesting to see how far these results can be extended to other valuation structures, and to more general frameworks, such as the semiring-based framework.

References

1. Anglès d'Auriac, J-Ch., Igloi, F., Preismann, M. & Sebö, A. "Optimal cooperation and submodularity for computing Potts' partition functions with a large number of statistics", *J. Physics A: Math. Gen.* 35, (2002), pp. 6973–6983.
2. Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T. & Verfaillie, G. "Semiring-based CSPs and valued CSPs: Frameworks, properties and comparison", *Constraints* 4, (1999), pp. 199–240.
3. Bulatov, A.A. "A dichotomy theorem for constraints on a three-element set", *Proc. 43rd IEEE Symposium on Foundations of Computer Science (FOCS'02)*, (2002), pp. 649–658.
4. Bulatov, A.A., Jeavons, P. & Krokhin, A. "Classifying the complexity of constraints using finite algebras", *SIAM Journal on Computing* 34, (2005), pp. 720–742.
5. Burkard, R., Klinz, B. & Rudolf, R. "Perspectives of Monge properties in optimization", *Discrete Applied Mathematics* 70, (1996), pp. 95–161.
6. Cohen, D., Cooper, M.C. & Jeavons, P. "A complete characterisation of complexity for Boolean constraint optimization problems", *Proc. 10th Int. Conf. on Principles and Practice of Constraint Programming (CP'04)*, *LNCS* 3258, (2004), pp. 212–226.
7. Cohen, D., Cooper, M.C., Jeavons, P. & Krokhin, A. " Soft constraints: complexity and multimorphisms" *Proceedings of CP'03*, *LNCS* 2833, (2003), pp. 244–258.
8. Cohen, D., Cooper, M.C., Jeavons, P. & Krokhin, A. "A maximal tractable class of soft constraints" *Journal of Artificial Intelligence Research* 22, (2004), pp. 1-22.
9. Cohen, D., Cooper, M.C., Jeavons, P. & Krokhin, A. "Supermodular functions and the complexity of Max CSP", *Discrete Applied Mathematics*, 149 (1-3),(2005), pp. 53–72.
10. Cohen, D., Cooper, M.C., Jeavons, P. & Krokhin, A. "The Complexity of Soft Constraint Satisfaction", *Artificial Intelligence*, to appear.

11. Creignou, N. “A dichotomy theorem for maximum generalised satisfiability problems”, *Journal of Computer and Systems Sciences* 51(3), (1995), pp. 511–522.
12. Creignou, N., Khanna, S. & Sudan, M. *Complexity classification of Boolean constraint satisfaction problems*, volume 7 of *SIAM Monographs on Discrete Mathematics and Applications*, (2001).
13. Cunningham, W.H. “Minimum cuts, modular functions, and matroid polyhedra”, *Networks* 15(2), (1985), pp. 205–215.
14. Feder, T. & Vardi, M.Y. “The computational structure of monotone monadic SNP and constraint satisfaction: a study through datalog and group theory”, *SIAM Journal on Computing* 28(1), (1998), pp. 57–104.
15. Fujishige, S. *Submodular Functions and Optimization*, 2nd edn., Annals of Discrete Mathematics, Vol. 58, Elsevier, (2005).
16. Garey, M.R. & Johnson, D.S., *Computers and Intractability: A guide to the theory of NP-completeness*, W.H. Freeman, (1979).
17. Iwata, S. “A fully combinatorial algorithm for submodular function minimization”, *Journal of Combinatorial Theory, Series B* 84(2), (2002), pp. 203–212.
18. Iwata, S., Fleischer, L. & Fujishige, S. “A combinatorial, strongly polynomial-time algorithm for minimizing submodular functions”, *Journal of the ACM* 48(4), (2001), pp. 761–777.
19. Jeavons, P.G. “On the algebraic structure of combinatorial problems”, *Theoretical Computer Science* 200 (1998), pp. 185–204.
20. Jeavons, P.G. “Constructing constraints”, *Proceedings of CP’98), Lecture Notes in Computer Science* 1520, (1998), pp. 2–17.
21. Jeavons, P.G., Cohen D.A. & Gyssens, M. “Closure properties of constraints”, *Journal of the ACM* 44, (1997), pp. 527–548.
22. Jeavons, P.G., Cohen D.A. & Gyssens, M. “How to determine the expressive power of constraints”, *Constraints*, 4, (1999), pp. 113–131.
23. Jonsson, P., Klasson, M. & Krokhin, A. “The approximability of three-valued MAX CSP”, *SIAM Journal of Computing*, 35, (2006), pp. 1329–1349.
24. Nagamochi, H. & Ibaraki, I. “A note on minimizing submodular functions”, *Information Processing Letters* 67, (1998), pp. 239–244.
25. Narayanan, H., *Submodular Functions and Electrical Networks*, Annals of Discrete Mathematics 54, North Holland (London, New York, Amsterdam), (1997).
26. Narayanan, H. “A note on the minimisation of symmetric and general submodular functions” *Discrete Applied Mathematics* 131(2), (2003), pp. 513–522.
27. Nemhauser, G.L. & Wolsey, L.A. *Integer and Combinatorial Optimisation*, Wiley, (1999).
28. Queyranne, M. “Minimising symmetric submodular functions”, *Mathematical Programming* 82(1-2), (1998), pp. 3–12.
29. Schaefer, T.J. “The complexity of satisfiability problems”, *Proc. 10th ACM Symposium on Theory of Computing (STOC’78)*, (1978), pp. 216–226.
30. Schrijver, A. “A combinatorial algorithm minimizing submodular functions in strongly polynomial time”, *J. Combinatorial Theory*, Series B, 80, (2000), pp. 346–355.
31. Schrijver, A. *Theory of Linear and Integer Programming*, Wiley, (1986).
32. Szendrei, A., *Clones in Universal Algebra*, Seminaires de Mathematiques Superieures, University of Montreal, 99, (1986).
33. Topkis, D. *Supermodularity and Complementarity*, Princeton University Press, (1998).

Typed Guarded Decompositions for Constraint Satisfaction

David A. Cohen and Martin J. Green[*]

Department of Computer Science
Royal Holloway, University of London, UK

Abstract. The constraint satisfaction problem is in general NP-hard. As such, our aim is to identify tractable classes of constraint satisfaction problem instances (CSPs). Tractable classes of CSPs are normally described by limiting either the structure or the language of the CSPs. Structural decomposition methods identify CSPs whose reduction to the acyclic class is bound by a polynomial. These structural decompositions have been a very useful way to identify large tractable classes.

However, these decomposition techniques have not yet been applied to relational tractability results. In this paper we introduce the notion of a typed guarded decomposition as a way to generalize the structural decompositions. We develop a no-promise algorithm which derives large new tractable classes of CSPs that are not describable as purely structural nor purely relational classes.

1 Introduction

The constraint satisfaction problem is in general NP-hard. As such, we aim to identify tractable classes of constraint satisfaction problem instances (CSPs). We call a class of CSPs tractable if there exists a uniform polynomial time algorithm for identifying and solving the instances of the class. These two restrictions for tractable classes allow us to consider tractability as corresponding to the existence of a good solution method.

Tractable classes of CSPs are normally described by limiting either the structure of the CSPs (the interaction of their constraints) or the language of the CSPs (the types of constraint relation that are allowed). A basic tractable structural class is those CSPs whose structure is acyclic [1]. This class has been extended by considering CSPs whose structure is "nearly acyclic" in that there is a tractable reduction to an acyclic CSP [7]. Unfortunately, these structural decomposition techniques have not yet made use of relational tractability results.

The framework of guarded decomposition is a natural way to express all known structural decompositions. Bounded width acyclic guarded decompositions are tractably solvable, but it is thought that they are intractable to identify. To overcome this limitation, Chen and Dalmau [2] consider a no-promise algorithm for the class of CSPs whose structure has a bounded width acyclic guarded

[*] This work was supported by EPSRC Grant EP/C525949/1.

decomposition. In this paper, we use the no-promise approach to solve a class of CSPs that generalize the guarded decompositions.

This paper presents new and exciting results on hybrid tractability. Unfortunately, due to the limitations of a conference paper, the presentation is necessarily terse. A full version with longer descriptions and proofs will appear in a journal presently.

Our new notion of *typed guarded decomposition* allows us to distinguish the underlying *separation structure* from the *components* themselves. We then develop a no-promise algorithm which derives large new tractable classes of CSPs that are not describable as purely structural nor purely relational classes. These preliminary but very exciting results demonstrate that there is a future in hybrid tractability research and, in particular, generalized decompositions.

Outline of the Paper. In Sect. 2 we provide the necessary background and definitions needed throughout the paper. In Sect. 3 we develop the framework of typed guarded decompositions. We consider how these might be used to solve CSPs in Sect. 4. We draw conclusions in Sect. 5.

2 Definitions

Definition 1. *A* ***Constraint Satisfaction Problem instance*** *(CSP), P, is a triple, $\langle V, D, C\rangle$ where;*

- *V is a set of* ***variables****,*
- *D is any set, called the* ***domain*** *of the instance, and*
- *C is a set of* ***constraints****.*
  Each constraint $c \in C$ is a pair $\langle \sigma, \rho\rangle$ where σ is a subset of variables from V, called the ***constraint scope****, and ρ is a set of functions from σ to D, called the* ***constraint relation****. Each function in ρ represents one allowed assignment to the variables in σ. We call $|\sigma|$ the* ***arity*** *of the constraint.*

A ***solution*** *to P is a function $\phi : V \to D$ such that, for each $\langle \sigma, \rho\rangle$ in C, the restriction of ϕ to σ, denoted $\phi_{|\sigma}$, is in ρ. We say that two CSPs are* ***solution equivalent*** *if they have the same set of solutions.*

We define the ***join*** *of a pair of constraints $\langle \sigma_1, \rho_1\rangle$ and $\langle \sigma_2, \rho_2\rangle$ to be the constraint $\big\langle \sigma_1 \cup \sigma_2, \big\{f : \sigma_1 \cup \sigma_2 \to D \mid f_{|\sigma_1} \in \rho_1 \wedge f_{|\sigma_2} \in \rho_2\big\}\big\rangle$. The* ***projection*** *of $\langle \sigma, \rho\rangle$ onto a set of variables X is the constraint $\big\langle \sigma \cap X, \big\{f_{|\sigma\cap X} \mid f \in \rho\big\}\big\rangle$.*

Informally, we may describe V as a set of questions that need to be answered. The domain D is the set of all possible answers that can be given to any of these questions. A constraint is a rationality condition that limits the answers that may be simultaneously assigned to some group of questions. A solution is then a satisfactory set of answers to all of the questions.

It is often very convenient to use an alternative representation of a constraint. For this we will assume (without loss of generality) that the variables of any CSP are indexed (as $\{v_1, \dots, v_n\}$) and use the index order as a natural order for sets of variables. We then represent a constraint $c = \langle \sigma, \rho \rangle$ as a pair $\langle s, R \rangle$ where s is a list comprising the variables of σ ordered by increasing index and R is the $|\sigma|$-ary relation $\{f(s) \mid f \in \rho\}$. We call R the *defining relation* of c.

Definition 2. *For any CSP $\langle V, D, C \rangle$ and set of constraints $Y \subseteq C$ we define the **component** of P with respect to Y to be the CSP with variables $\bigcup_{\langle \sigma, \rho \rangle \in Y} \sigma$, domain $\bigcup_{\langle \sigma, \rho \rangle \in Y} \left(\bigcup_{f \in \rho} \{f(v) \mid v \in \sigma\} \right)$, and constraints Y.*

In this paper we will be concerned with both the complexity of solving the components of a CSP and the way in which components interact.

2.1 The Complexity of Constraint Satisfaction

In this paper, we normally consider problems whose instances are CSPs.

Definition 3. *The **decision problem** for a set S of CSPs is:*
Instance: A CSP P in S
Question: Does P have a solution?

Definition 4. *The **search problem** for a set S of CSPs is:*
Instance: A CSP P in S
Question: What assignment (if any) is a solution for P?

We call a problem $\mathcal{P}$ *tractable* if there exists a uniform polynomial time algorithm for answering the question of $\mathcal{P}$ for all instances of $\mathcal{P}$. We call $\mathcal{P}$ *intractable* if it is NP-hard [5] to answer. The decision problem for the set of all CSPs is intractable [15]. We therefore concentrate our efforts on sets of CSPs for which the decision (or search) problem is tractable. This highlights an extra condition on classes of CSPs: can we tell whether some general CSP is in the class? This notion gives rise to the identification problem for sets of CSPs.

Definition 5. *The **identification problem** for a set S of CSPs is:*
Instance: A CSP P
Question: Is P in S?

We call a set S of CSPs *tractably identifiable* if the identification problem for S is tractable. We call S *tractably solvable* if the search problem for S is tractable. S is *tractable* if it is both tractably identifiable and tractably solvable, otherwise it is intractable.

For certain classes of CSPs the identification problem is intractable. In this case, it might appear as if we are left without a useful solution algorithm. However, Chen and Dalmau [2] have developed a framework for considering problems that combine identification and solution, the so called no-promise problem.

Definition 6. *The **no-promise problem** for a set S of CSPs is:*
Instance: A CSP P
Question: If $P \in S$ does P have a solution?

For an instance outside of S an algorithm solving the no-promise problem may decline to decide if the instance is satisfiable or not.

Typically, a no-promise algorithm is much less efficient than an algorithm which specifically identifies, then solves, the instances of a tractable class. This means that tractably identifiable classes are still of great importance.

2.2 Tractable Classes

There are many tractable classes of CSPs. Many of these can be described by limiting either the interaction of the constraints or else the types of constraint relation that we allow.

Definition 7. *A* ***hypergraph*** *H is a pair $\langle V, E\rangle$ where V is a set of vertices and E is a collection of subsets of V, called the* ***hyperedges*** *of H.*

The ***structure*** *of a CSP P, denoted $\sigma(P)$, is the hypergraph whose vertices are the variables of P and whose hyperedges are the scopes of the constraints of P. For a hypergraph H we denote by $\Psi(H)$ the set of CSPs whose structure is H. A class S of CSPs is called* ***structural*** *if it is defined by limiting the structure of the instances in S.*

A basic tractable structural class is those instances whose structure is acyclic [1]. However, this class has been extended by considering instances whose structure is "nearly acyclic" in the sense that there is a tractable reduction to an acyclic instance [7]. These reductions give rise to sets of CSPs whose reduction to the acyclic class is bound by a polynomial. Such classes have been well-studied and have made the use of tractable structural classes applicable to real-world examples. We will consider a framework for describing these reductions, or structural decompositions, in Sect. 2.3.

Definition 8. *A* ***constraint language*** *over a domain D is a set of relations over D. The* ***language*** *of a CSP P, denoted $\rho(P)$, is the constraint language formed by the set of defining relations of the constraints of P. The language of a set of instances is the union of the languages of the instances. For any constraint language Γ we will refer to the set of instances with language contained in Γ as* $\mathrm{CSP}(\Gamma)$*. A class S of CSPs is called* ***relational*** *if it is defined by limiting the language of the instances in S.*

A constraint language is called tractable if the set of instances defined over this language is tractably solvable. There are many known tractable constraint languages [12,14,13,11].

Definition 9. *Let D be a finite ordered domain. A relation ρ over D, with arity r, is called* ***max-closed*** *[14] if for every t_1 and t_2 in ρ we have that* $\langle \max(t_1[1], t_2[1]), \ldots, \max(t_1[r], t_2[r])\rangle$ *is also in ρ.*

We call a CSP max-closed if its language is max-closed. It has been shown [14] that the max-closed constraint language is tractable.

Definition 10. *We call a CSP $P = \langle V, D, C \rangle$* ***pairwise consistent*** *[16,10] if for every $\langle \sigma_1, \rho_1 \rangle, \langle \sigma_2, \rho_2 \rangle \in C$ we have that $\pi_{\sigma_1 \cap \sigma_2} \langle \sigma_1, \rho_1 \rangle = \pi_{\sigma_1 \cap \sigma_2} \langle \sigma_2, \rho_2 \rangle$.*

Pairwise consistency can be achieved in polynomial time [10] and is sufficient to decide some sets of CSPs.

Example 1. **Solving max-closed CSPs**
Let P be a max-closed CSP. We establish pairwise consistency by successively choosing constraints $c_1 = \langle \sigma_1, \rho_1 \rangle$ and $c_2 = \langle \sigma_2, \rho_2 \rangle$ from P and replacing c_1 with the projection onto σ_1 of the join of c_1 and c_2. Since no assignment not in the join of c_1 and c_2 can extend to a solution of P this can remove no solutions. Eventually this process must terminate since the domain is finite.

If there is now a constraint with empty constraint relation then there is no solution to P. Otherwise, we are guaranteed a solution which can be found by choosing, independently for each variable, the largest value consistent with all constraints.

There is an analogous set of min-closed CSPs with a similar solution algorithm (choose the smallest remaining value after pairwise consistency).

2.3 Guarded Decompositions

First we need to define a guarded decomposition.

Definition 11. [3] *A* ***guarded block*** *of a hypergraph H is a pair $\langle \lambda, \chi \rangle$ where the* ***guard****, λ, is a set of hyperedges of H, and the* ***block****, χ, is a subset of the vertices of the guard.*

For any CSP, P, and any guarded block $\langle \lambda, \chi \rangle$ of $\sigma(P)$, the constraint ***generated*** *by P on $\langle \lambda, \chi \rangle$ is the constraint $\langle \chi, \rho \rangle$, where ρ is the projection onto χ of the join of all the constraints of P whose scopes are elements of λ.*

A set of guarded blocks Ξ of a hypergraph H is called a ***guarded decomposition*** *of H if for every CSP $P = \langle V, D, C \rangle$ in $\Psi(H)$, the instance $P' = \langle V, D, C' \rangle$, where C' is the set of constraints generated by P on the members of Ξ, is solution equivalent to P.*

A guarded block $\langle \lambda, \chi \rangle$ of a hypergraph H ***covers*** *a hyperedge e of H if e is contained in χ. A set of guarded blocks Ξ of a hypergraph H is called a* ***guarded cover*** *for H if each hyperedge of H is covered by some guarded block of Ξ. A set of guarded blocks Ξ of a hypergraph H is called a* ***complete guarded cover*** *for H if each hyperedge e of H occurs in the guard of some guarded block of Ξ which covers e.*

It has been shown that a set of guarded blocks Ξ of a hypergraph H is a guarded decomposition of H if and only if it is a complete guarded cover [3]. It has also been shown that all useful structural decompositions, such as cycle-cutsets [4], hinges [9], hypertrees [6], etc., may be represented as guarded decompositions by imposing certain restrictions on the guarded blocks [3]. As such, the guarded decomposition is a highly useful framework for discussing structural decompositions.

A set of guarded blocks is acyclic if the set of blocks of these guarded blocks is an acyclic set of hyperedges over their set of vertices (the union of the blocks). All structural decompositions aim to tractably identify a set of CSPs for which there exists a polynomial time reduction into the set of acyclic CSPs. For guarded decompositions this reduction is tractable when we have a uniform bound on the number of hyperedges in any guarded block.

Definition 12. *The* ***width*** *of a set of guarded blocks is the maximum number of hyperedges in any of its guards.*

Definition 13. *A* ***join tree*** *of an acyclic set of guarded blocks $\mathcal{D}$ is any tree $J = \langle \mathcal{D}, E \rangle$ where the following* ***connectivity condition*** *holds:*

– *For every pair of guarded blocks $\langle \lambda_1, \chi_1 \rangle$ and $\langle \lambda_2, \chi_2 \rangle$ in $\mathcal{D}$ we have that $\chi_1 \cap \chi_2$ is contained in the block of every guarded block that exists on the unique path between $\langle \lambda_1, \chi_1 \rangle$ and $\langle \lambda_2, \chi_2 \rangle$ in J.*

Assume we have a CSP P together with a width k acyclic guarded decomposition, $\mathcal{D}$, of $\sigma(P)$. Firstly, we generate a join tree J of $\mathcal{D}$ and label each node with the constraint generated by P onto that guarded block. The set of constraints of the labels of J defines an acyclic CSP that is solution equivalent to P (it is the CSP generated by P on $\mathcal{D}$). We therefore solve P by finding a solution to this derived CSP using J. We apply pairwise consistency between the labels of the nodes of J and solve by assigning from any chosen root to the leaves. The connectivity condition ensures that we cannot conflict anywhere during the search whilst the pairwise consistency is sufficient to completely propagate partial solutions. The algorithm runs in polynomial time when k is bounded.

The No-Promise Algorithm for Guarded Decompositions. Chen and Dalmau [2] developed an algorithm for solving the decision problem for the set of instances with bounded width acyclic guarded decompositions. This algorithm correctly decides any instance whose structure has a bounded width acyclic guarded decomposition, without requiring that this guarded decomposition actually be generated (although we are promised that it exists). The algorithm, called the *projective k-consistency algorithm*, works by joining all k-sets of constraints and then performing pairwise consistency on the new set of constraints. It has been shown that the original instance is satisfiable if and only if, after the pairwise consistency has been performed, no new constraint relation has become empty (Theorem 20 of [2]).

Intuitively, since we are only performing consistency operations, we must preserve solutions during this algorithm (solutions cannot be removed by consistency techniques). The initial joining of the k-sets of constraints ensures that all possible guarded blocks in the successful guarded decomposition will have been utilized to generate new constraints. The pairwise consistency then ensures that the join tree of this guarded decomposition will be made consistent. The result follows immediately.

Chen and Dalmau then proceed to develop this algorithm into a no-promise algorithm using a controller algorithm. This controller repeatedly runs the projective k-consistency algorithm, then tries to find a compatible value for a particular variable. It re-runs the projective k-consistency algorithm after each assignment. If no compatible value can be found for the chosen variable then the algorithm returns 'I don't know'. The algorithm is shown to correctly solve all CSPs whose structure has an acyclic guarded decomposition of width at most k (Theorem 21 of [2]) and may, with luck, solve some other instances.

3 Generalizing Structural Decompositions

The limitation for decomposition techniques in that only easy to solve components can be used. For the structural decompositions this of course implies bounded size (resp. width) components (resp. guarded blocks). These are then "put together" in an easy to solve way, namely they form an acyclic structure.

We might have more power if we separate our concerns between the interaction of the components and the components themselves. The remainder of this paper is a framework for developing this separation of concerns.

Perhaps the most important part of the decomposition is the way in which the guarded blocks interact, that is, the intersection of the blocks. In the case of structural decompositions this is always an acyclic structure. If we are to allow arbitrary sized guarded blocks then we need some way to talk about their interaction.

Definition 14. *Let $\mathcal{D}$ be an acyclic guarded decomposition of a hypergraph $H = \langle V, E\rangle$ and let $\langle \lambda, \chi\rangle$ be a guarded block from $\mathcal{D}$. A set of hyperedges $\epsilon \subseteq E$ is called a **separating edge set** for $\langle \lambda, \chi\rangle$ in $\mathcal{D}$ if, for all $\langle \lambda_2, \chi_2\rangle$ in $\mathcal{D}$, $\langle \lambda_2, \chi_2\rangle \neq \langle \lambda, \chi\rangle$, we have that $\chi \cap \chi_2 \subseteq \bigcup \epsilon$. We define the **intersection vertices** of $\langle \lambda, \chi\rangle$ with respect to $\mathcal{D}$ to be the set of vertices given by $\bigcup_{\langle \lambda_2,\chi_2\rangle \in \mathcal{D}, \langle \lambda_2,\chi_2\rangle \neq \langle \lambda,\chi\rangle} (\chi \cap \chi_2)$.*

*When there exists a separating edge set for $\langle \lambda, \chi\rangle$ in $\mathcal{D}$ with size at most k we say that $\langle \lambda, \chi\rangle$ is **k-separated** in $\mathcal{D}$. We call $\mathcal{D}$ **k-separated** if every guarded block in $\mathcal{D}$ is k-separated in $\mathcal{D}$.*

*For a guarded block $\langle \lambda, \chi\rangle$ of $\mathcal{D}$ and a separating edge set ϵ for $\langle \lambda, \chi\rangle$ in $\mathcal{D}$ we define the guarded block $\langle \epsilon, X\rangle$ to be a **separator** for $\langle \lambda, \chi\rangle$ in $\mathcal{D}$, where X is the intersection vertices of $\langle \lambda, \chi\rangle$ in $\mathcal{D}$. When no smaller separating edge set exists we call the separator **minimal**. If ϵ has size at most k we call the separator a **k-separator**. We define a **separation structure** of $\mathcal{D}$ to be a set of minimal separators for the guarded blocks of $\mathcal{D}$.*

When the width of a separation structure is at most k (so that all separators are k-separators) we may call this a k-separation structure. A separation structure is a set of guarded blocks. We have the following useful result.

Proposition 1. *A guarded decomposition is acyclic if and only if its separation structure is acyclic.*

Proof. Omitted for brevity.

In the remainder of this paper we will consider classes of CSPs for which there exists an acyclic k-separation structure. These classes will use this acyclic k-separation structure to pass constraint information between their components.

Definition 15. *A **type**, τ, is a polynomial time solution algorithm for some set of CSPs denoted by S_τ. The input to a type is a CSP in S_τ and the output is either 'yes', together with a solution to this CSP, or else 'no'.*

The existence of a type τ for a set of CSPs S_τ provides a witness that S_τ is tractably solvable. We will use these types to generate hybrid tractability results which depend not only on the structure of the instances but also on the types of the components.

Example 2. **The max-closed type**
Recall the method used to solve the max-closed CSPs given in Ex. 1. This method gives rise to the max-closed type, which firstly performs pairwise consistency on the input CSP and then chooses the largest remaining domain value at each variable. As required, the max-closed type runs in polynomial time (in particular on the max-closed CSPs) and is a solution algorithm for the max-closed CSPs.

Since we are aiming for a solution algorithm which acts independently for each separated component we need to reconsider what it means for a guarded block to provide cover. The cover must now be provided on a per constraint basis, rather than on a per scope basis.

In the case of guarded decompositions we have that a set of guarded blocks $\mathcal{D}$ of a hypergraph H decomposes a CSP P if $\sigma(P) = H$ and $\mathcal{D}$ is a complete guarded cover of H. In the purely structural case we are allowed to join all constraints over common scopes. Then we require only a single guarded block to cover each scope. Now that we are concerned with the components of a CSP and the algorithms (types) that solve these components we require that each constraint itself is in need of cover by some guarded block. The following definitions formalize this idea for guarded decompositions. It is clear that for each constraint to be covered by the block of some guarded block we require that the set of guarded blocks is itself a guarded decomposition (a complete guarded cover).

Definition 16. *Let $P = \langle V, D, C\rangle$ be a CSP, $\mathcal{D}$ be a guarded decomposition of $\sigma(P)$ and μ be a mapping from C to (non-empty) subsets of $\mathcal{D}$. We say that $\mathcal{D}$ **decomposes P with respect to** μ if;*

- *for every c in C we have that the scope of c is;*
  - *in the guard of every guarded block in $\mu(c)$, and*
  - *in the block of at least one guarded block in $\mu(c)$, and*
- *for every guarded block β in $\mathcal{D}$ we have that for every hyperedge e in its guard there exists at least one constraint c in C whose scope is e and for which β is contained in $\mu(c)$.*

We call such a μ a ***decomposition function*** *for P with respect to $\mathcal{D}$. For a guarded block $\langle \lambda, \chi \rangle$ of $\sigma(P)$ we define the constraint* ***generated*** *by P on $\langle \lambda, \chi \rangle$* ***with respect to*** *μ to be the constraint $\langle \chi, \rho \rangle$ where ρ is the relation found by projecting onto χ the join of all those constraints c for which $\mu(c)$ contains $\langle \lambda, \chi \rangle$.*

Define P_μ to be the CSP $\langle V, D, C' \rangle$ where C' is the set of constraints generated by P on the guarded blocks of $\mathcal{D}$ with respect to μ.

Proposition 2. *Let $P = \langle V, D, C \rangle$ be a CSP, $\mathcal{D}$ be a guarded decomposition of $\sigma(P)$ and μ be a decomposition function for P with respect to $\mathcal{D}$. The CSP P_μ is solution equivalent to P.*

Proof. Since P_μ is generated only by joining and projecting constraints from P we find that any solution to P is also a solution to P_μ.

Next, consider any solution s to P_μ. By definition of μ and the construction of P_μ we have that each constraint c from P must have been joined to form some constraint in P_μ that covers c. As such, the restriction of s onto the scope of any constraint of P must be allowed by this constraint.

When we are only concerned with structure, as is the case with guarded decompositions, we may use a decomposition function which maps each constraint to the set of all guarded blocks whose guards contain its scope.

3.1 Typed Guarded Decompositions

In this section we complete our framework by defining the *typed guarded decomposition*. These decompositions will allow us to define new tractably solvable sets of CSPs that are not definable either by limiting only their structure or limiting only their language. In this sense, they provide new hybrid tractability results.

Definition 17. *A* ***typed guarded decomposition*** *of a hypergraph H is a pair $\langle \mathcal{T}, \mathcal{D} \rangle$ where $\mathcal{D}$ is a guarded decomposition of H and $\mathcal{T}$ is a function mapping each guarded block of $\mathcal{D}$ to a type. We call $\mathcal{D}$ the* ***associated guarded decomposition*** *of $\langle \mathcal{T}, \mathcal{D} \rangle$. For a particular guarded block β of $\mathcal{D}$ we call $\mathcal{T}(\beta)$ its* ***associated type****.*

Typed guarded decompositions place further restrictions of the set of CSPs which they decompose.

Definition 18. *Let $P = \langle V, D, C \rangle$ be a CSP, $\langle \mathcal{T}, \mathcal{D} \rangle$ a typed guarded decomposition of $\sigma(P)$ and μ a decomposition function for P with respect to $\mathcal{D}$. For any guarded block β of $\mathcal{D}$ we define the* ***component of*** *P* ***induced by*** *β* ***with respect to*** *μ to be the CSP derived from the set of all constraints c in C for which $\mu(c)$ contains β.*

We say that $\langle \mathcal{T}, \mathcal{D} \rangle$ ***decomposes*** *P* ***with respect to*** *μ if, for every guarded block β in $\mathcal{D}$, we have that the component of P induced by β with respect to μ is an instance of $S_{\mathcal{T}(\beta)}$.*

P_μ is defined as before and so the solution equivalence to P holds (by Proposition 2). In addition, typed guarded decompositions give us the restriction that the components induced by a CSP and decomposition function on the guarded blocks must be solvable by specified algorithms (the associated types).

4 Solving CSPs Using Typed Guarded Decompositions

In this section we consider a certain format for typed guarded decompositions that will allow us to develop an extension of the guarded decompositions of bounded width, the *acyclic k-separated typed guarded decomposition.*

Definition 19. *We call a typed guarded decomposition* ***acyclic*** *if its associated guarded decomposition is acyclic.*

A typed guarded decomposition $\langle \mathcal{T}, \mathcal{D} \rangle$ is called ***k-separated*** *if $\mathcal{D}$ is itself k-separated. Similarly, we define a* ***separation structure*** *for $\langle \mathcal{T}, \mathcal{D} \rangle$ to be a separation structure of $\mathcal{D}$.*

By Proposition 1 we know that a typed guarded decomposition is acyclic if and only if its separation structure is acyclic. We will use Proposition 1 in the sections that follow to demonstrate classes of acyclic k-separated typed guarded decompositions whose instances may be solved in polynomial time.

Example 3. **Acyclic k-separated max and min-closed components**
Consider the set of CSPs whose instances are an acyclic k-separated combination of max-closed and min-closed components. This set of CSPs is not tractable for any structural reason, since the components can be arbitrarily large. Nor is it tractable for any relational reason, since the union of the max-closed and min-closed constraint languages is not tractable. However, the instances of Ex. 3 are naturally described using acyclic k-separated typed guarded decompositions as described in this paper.

In Sect. 4.1 we demonstrate a sufficient condition on the acyclic k-separated typed guarded decompositions so that the decomposable CSPs may be solved in polynomial time, but only when we are also provided with the decomposition and the decomposition function as part of the input. Then, in Sect. 4.2, we show that a polynomial time algorithm exists that can solve the no-promise problem for the acyclic k-separated typed guarded decompositions. Once more, we require more restrictive conditions on the decompositions. We provide sufficient conditions for the solution to be polynomial. In particular, we show that the no-promise problem for the set of CSPs of Ex. 3 is tractable for bounded k.

4.1 A Simple Requirement for Polynomial Time Solution

In order to solve the components induced by the decomposable CSPs of the acyclic k-separated typed guarded decompositions (using their associated types) we will require that certain *constant constraints* can be added into the input CSP and maintain the property that the types are solution algorithms.

Definition 20. *A* ***constant constraint*** *is a constraint that only allows a single assignment.*

Definition 21. *For a constant k, consider the problem $\mathcal{A}_k$ whose instances are triples $\langle P, \langle \mathcal{T}, \mathcal{D} \rangle, \mu \rangle$ where;*

- *P is a CSP,*
- *$\langle \mathcal{T}, \mathcal{D} \rangle$ is an acyclic k-separated typed guarded decomposition of $\sigma(P)$, and*
- *μ is a decomposition function for P with respect to $\mathcal{D}$ such that $\langle \mathcal{T}, \mathcal{D} \rangle$ decomposes P with respect to μ.*

For an instance of $\mathcal{A}_k$ the question is: What is a solution to P?

Example 4. **Solving CSPs using acyclic k-separated typed guarded decompositions**
Let $\langle P, \langle \mathcal{T}, \mathcal{D} \rangle, \mu \rangle$ be an instance of $\mathcal{A}_k$ for some constant k. We solve this instance by finding a solution (if one exists) for P. We do this as follows.

1. Generate a k-separator for each guarded block of $\mathcal{D}$. At the same time, generate a mapping κ from the identified k-separators to the set of guarded blocks in $\mathcal{D}$ which have derived that k-separator.
   (By Prop. 1 the found k-separation structure is acyclic.)
2. Generate a join tree J of the k-separation structure. Label each node with the constraint generated by P on that node (k-separator).
   (There is as yet no guarantee that the solutions to the separation structure will propagate into the components of P induced by the guarded blocks with respect to μ.)
3. For each k-separator β take each allowed assignment to the label of β in J (the generated constraint) and generate a constant constraint on the block of β (which are the intersection variables for each guarded block in $\kappa(\beta)$).
   Then, for each guarded block α in $\kappa(\beta)$, pass the CSP formed by the constant constraint, together with the set of constraints which μ maps to α, into $\mathcal{T}(\alpha)$. If it returns that no solution exists, remove this assignment from the constraint generated on β (the label of β in J). Otherwise, if all applications of the types (with respect to their components and this constant) succeed, this assignment is kept.
   (At this point, the labels of the k-separators in the join tree directly capture the possible assignments that are acceptable for both this k-separator and its components.)
4. Apply pairwise consistency to the join tree.
   - If any constraint relation is wiped out then return 'no'.
   - Otherwise P does have a solution. Firstly, generate a solution on the k-separators. Then, to extend this solution, re-run the types on the components (as in Step 3) using the projected partial solution as an additional constant. Return 'yes'.
     (By definition of this partial solution, these constants must extend to the components.)

However, this algorithm raises one necessary condition. We require that the types are polynomial time solution algorithms for their respective components of the CSP. This is certainly true of the set of constraints that μ maps to a given guarded block, but in this algorithm we are also required to add an additional constant constraint. The following is a sufficient condition for this algorithm to work.

Definition 22. *Let $\langle P, \langle \mathcal{T}, \mathcal{D} \rangle, \mu \rangle$ be an instance of A_k (for some k).*

*We say that P is **good** if for every guarded block β in $\mathcal{D}$ we have that the component of P induced by β with respect to μ, together with any constant constraint on the intersection vertices of β, is an instance of S_τ (that is, polynomial time solvable by τ).*

It is possible to verify that the algorithm of Ex. 4 runs in polynomial time (in the size of the CSP P). As such, it allows us to prove the following theorem.

Theorem 1. *For a constant k the set of instances of $\mathcal{A}_k$ that are good is tractably solvable.*

Proof. Omitted for brevity.

Theorem 1 demonstrates that the search problem for the set of good instances from A_k is tractable (for bounded k). However, it is quite likely that determining whether an appropriate typed acyclic k-separated guarded decomposition and decomposition function exists for a given CSP is hard. Fortunately, there is an algorithm we can use that solves the no-promise problem for a large subset of these instances.

4.2 A No-Promise Algorithm for Acyclic k-Separated Typed Guarded Decompositions

In previous work [8] the authors identified a property of sets of CSPs that make them more amenable to being used as components for use in typed guarded decompositions. If a tractable set S of CSPs remains tractable after we are allowed to add any number of constant constraints into any instance of S then we call S *constant additive tractable.* Solution algorithms to such useful tractable sets of CSPs can often be used as types for typed guarded decompositions.

Classes such as that of Ex. 3 naturally generalize the *bounded interaction constant additive tractable* classes identified previously by the authors [8]. For these restricted classes, identification was shown to be tractable. For the more general classes of this paper identification may well be hard. For this reason, a no-promise algorithm for our new decompositions is useful.

Definition 23. *A type τ is called a **no-promise type** if the following properties hold:*

1. *τ terminates on any input CSP, not just those in S_τ, and is polynomial time for any input.*
2. *τ allows all constant constraints to be added to the input whilst still being a polynomial time solution algorithm. This implies that S_τ is constant additive tractable.*
3. *τ preserves solutions, so that if τ replies 'no' then there really is no satisfying assignment to the input CSP (including any constants).*
4. *τ is monotonic (with respect to inclusion). By this we mean that, for any CSP $P = \langle V, D, C \rangle$, if $\tau(P)$ replies 'no' then, for every $P' = \langle V', D', C' \rangle$ with $V' \supseteq V$, $D' \supseteq D$ and $C' \supseteq C$ we have that $\tau(P')$ also replies 'no'.*

These properties might seem like a harsh requirement to place on types. However, many solution algorithms for tractable sets of CSPs have these properties. In particular, when a solution algorithm is based on a consistency procedure, such as pairwise consistency, these properties are (almost) certainly satisfied. For example, the max-closed type of Ex. 2 satisfies all of the required properties (as does the analogous min-closed type).

These properties for no-promise types are sufficient to allow us to derive a no-promise algorithm for the acyclic k-separated typed guarded decompositions over a pre-specified set of no-promise types.

Definition 24. *Let T be a set of no-promise types. We define a T-**guarded decomposition** to be a typed guarded decomposition whose types are all in T.*

Example 5. **A no-promise algorithm for acyclic k-separated T-guarded decompositions**

Let $P = \langle V, D, C\rangle$ be a CSP and T be a finite set of no-promise types. This algorithm takes P as input and returns 'yes' (together with a solution), 'no' or 'unsure'. It only returns 'unsure' if P is not decomposable by any acyclic k-separated T-guarded decomposition.

For any set C of constraints we define a k-join constraint over C to be the join of any k constraints from C.

1. Generate a binary CSP $P' = \langle V', D', C'\rangle$ where:
   - V' are all possible k-join constraints over C.
   - D' is the set of all allowed assignments from the constraint relations of the k-join constraints.
   - There is a binary constraint in C' for every pair of variables (k-joins) which allows a pair of domain values (assignments to k-joins) exactly when;
     - The two assignments are allowed by the respective k-join constraints and also do not conflict with each other (that is, they do not assign more than one value in D to each variable in V), and
     - The application (in turn) of every type in T to P, together with the two constant constraints generated by this pair of assignments, returns 'yes' (for all types in T).
2. Perform pairwise consistency on P'.
3. If any constraint relation in C' is wiped out then return 'no'.
4. Let W be an empty partial solution to P'. Repeat the following until W is a solution to P' (that is, has assigned all variables in V'):
   - Choose any unassigned variable in V' and find a consistent value for it in the following way:
     - Add any value (assignment to the respective k-join constraint over C) still remaining in its unary projection as a unary constant constraint in P'.
     - Remove (and remember) pairs of assignments from the constraint relations of C' that are found to be inconsistent with any type from T, when run on P together with the constant constraints generated by both this pair of assignments and all assignments in W.

        - Perform pairwise consistency on the modified P'. If a constraint relation is wiped out then try another value for this variable (by rolling back the removed assignments and replacing the constant constraint) until a consistent value is found.
    - If a consistent value is found add this assignment to W.
    - If no consistent value for the current variable exists then return 'unsure'.
5. Finally, if this backtrack-free search manages to successfully find a solution to P' then;
    - generate a solution for P by taking the assignments to V from the k-join assignments defined by W, and
    - return 'yes'.

Whilst the algorithm of Ex. 5 is non-trivial, it can be verified to run in polynomial time in the size of the input CSP (for bounded k and finite T).

Theorem 2. *Let k be a constant and T be a finite set of no-promise types. The no-promise problem for the set of all CSPs that are decomposable by some acyclic k-separated T-guarded decomposition is tractable.*

Proof. Omitted for brevity.

Theorem 2 provides a direct reason why the no-promise problem for the class of Ex. 3 (for bounded k) is tractable, since the set containing the max-closed type and min-closed type (both of which are no-promise types) is finite.

5 Conclusion

In this paper, we have seen that by modifying our view of decomposition we open up the idea of different ways to solve instances containing limited interaction between (tractable) components. We have developed a framework, the typed guarded decomposition, which we hope will provide a foundation to study these generalized decompositions. It seems that the key to generalizing decomposition methods is in identifying classes for which only a polynomial amount of propagation is necessary.

Unfortunately, we cannot yet say anything about the tractability of the identification problem for useful typed guarded decompositions. However, we have seen that we can derive an algorithm that solves the no-promise problem for large classes of typed guarded decompositions. We are at least able to solve decomposable instances even if we cannot tractably identify to which class they belong.

References

1. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30:479–513, 1983.
2. H. Chen and V. Dalmau. Beyond hypertree width: Decomposition methods without decompositions. In *Principles and Practice of Constraint Programming - CP 2005*, Lecture Notes in Computer Science, pages 167–181. Springer-Verlag, 2005.

3. D. Cohen, P. Jeavons, and M. Gyssens. A unified theory of structural tractability for constraint satisfaction and spread cut decomposition. In *Proceeedings of IJCAI'05*, pages 72–77, 2005.
4. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1988.
5. M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA., 1979.
6. G. Gottlob, L. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124:243–282, 2000.
7. G. Gottlob, L. Leone, and F. Scarcello. Hypertree decompositions: A survey. In *Proceedings 26th International Symposium on Mathematical Foundations of Computer Science, MFCS'01*, volume 2136 of *Lecture Notes in Computer Science*, pages 37–57. Springer-Verlag, 2001.
8. M.J. Green. *New Methods for the Tractability of Constraint Satisfaction Problems.* PhD thesis, University of London, Department of Computer Science, Royal Holloway, Egham, Surrey, UK, June 2005.
9. M. Gyssens, P.G. Jeavons, and D.A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.
10. P. Janssen, P. Jegou, B Nouguier, and M.C. Vilarem. A filtering process for general constraint satisfaction problems: achieving pair-wise consistency using an associated binary representation. In *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.
11. P.G. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200:185–204, 1998.
12. P.G. Jeavons and D.A. Cohen. An algebraic characterization of tractable constraints. In *Computing and Combinatorics. First International Conference COCOON'95 (Xi'an,China,August 1995)*, volume 959 of *Lecture Notes in Computer Science*, pages 633–642. Springer-Verlag, 1995.
13. P.G. Jeavons, D.A. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of the ACM*, 44:527–548, 1997.
14. P.G. Jeavons and M.C. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79(2):327–339, 1995.
15. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
16. D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

Propagation in CSP and SAT

Yannis Dimopoulos[1] and Kostas Stergiou[2]

[1] Department of Computer Science
University of Cyprus, Nicosia, Cyprus
yannis@cs.ucy.ac.cy

[2] Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece
konsterg@aegean.gr

Abstract. Constraint Satisfaction Problems and Propositional Satisfiability, are frameworks widely used to represent and solve combinatorial problems. A concept of primary importance in both frameworks is that of constraint propagation. In this paper we study and compare the amount of propagation that can be achieved, using various methods, when translating a problem from one framework into another. Our results complement, extend, and tie together recent similar studies. We provide insight as to which translation is preferable, with respect to the strength of propagation in the original problem and the encodings.

1 Introduction

CSPs and SAT are closely related frameworks that are widely used to represent and solve combinatorial problems. It is well known that there exist several ways to translate a problem expressed in one framework into the other framework (for example [2,14]).

One of the most important concepts in CSP and SAT is the concept of constraint propagation. Solvers in both frameworks utilize propagation algorithms both prior to and during search to prune the search space and save search effort. Recently there have been several studies exploring and comparing the amount of propagation that can be achieved in each framework using standard techniques, such as arc consistency (in CSPs) and unit propagation (in SAT), under various encodings from one framework to another [3,4,9,14]. A general lesson learned from these studies is that the choice of encoding is very important when comparing propagation methods in different frameworks. For example, arc consistency in a binary CSP is equivalent to unit propagation in the *support encoding* of the CSP into SAT [9,10]. On the other hand, arc consistency is stronger than unit propagation under the *direct encoding* [14].

Apart from the variety of ways to translate problems from one framework into another, a second source of complexity (and confusion) when comparing propagation techniques in different frameworks is the large number of such techniques that have been proposed. So far, the comparisons between propagation methods in CSPs and SAT have only considered standard techniques like arc consistency and forward checking, on one hand, and unit propagation on the other hand. Although these techniques remain at the core of most CSP and SAT solvers respectively, other stronger propagation methods are also attracting considerable interest in both communities in recent years. For example, some SAT solvers (e.g. `kcnfs`, `march_dl`, `Dew_Satz`, `2CLS+EQ`) employ strong reasoning techniques such as failed literal detection and variants of binary resolution [1,6,11]. Also, strong consistencies such as singleton and inverse consistencies are beginning to emerge as possible alternatives to arc consistency in CSPs [5,8].

In this paper we make a detailed theoretical investigation of the relationships between strong propagation techniques in CSPs and SAT. We consider various encodings of SAT into CSP and binary CSPs into SAT. In some cases we prove that, under certain encodings, there is a direct correspondence between a propagation method for CSPs and another one for SAT. For example, failed literal detection applied to a SAT problem achieves the same pruning as singleton arc consistency applied to the literal encoding of the problem into a CSP. In other cases, where no direct correspondence can be established, we identify conditions that must hold so that certain consistencies can achieve pruning in a SAT or CSP encoding, and/or place bounds in the pruning achieved by a propagation technique in one framework in terms of the other framework. For example, we show that failed literal detection applied to the direct encoding of a CSP is strictly stronger than restricted path consistency and strictly weaker than singleton arc consistency applied to the original CSP. Finally, we introduce new propagation techniques for SAT that capture the pruning achieved by certain CSP consistencies. For example, we introduce *subset resolution*; a form of resolution that captures the pruning achieved by arc consistency in the dual encoding of a SAT problem into a CSP.

Our results provide insight and better understanding of propagation in CSPs and SAT and complement recent similar studies. Also, we give indications as to when encoding a problem is beneficial and which encoding should be preferred, with respect to the strength of propagation in the original problem and the encodings. Note that, due to space restrictions, we only give some of the proofs (or sketches of proofs) for our theoretical results.

2 Preliminaries

A CSP, P, is defined as a triple $(\mathtt{X}, \mathtt{D}, \mathtt{C})$, where: $\mathtt{X} = \{x_1, \ldots, x_n\}$ is a finite set of n variables, $\mathtt{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, and $\mathtt{C}$ is a set of constraints. Each constraint $C \in \mathtt{C}$ is defined over a set of variables $\{x_{j_1}, \ldots, x_{j_k}\}$ and specifies the allowed combinations of values for these variables.

A binary constraint C on variables $\{x_i, x_j\}$ is *Arc Consistent* (AC) iff $\forall a \in D(x_i)\ \exists\, b \in D(x_j)$ such that the assignments (x_i, a) and (x_j, b) are compatible. In this case we say that b is a *support* for a on constraint C. A non-binary constraint is Generalized Arc Consistent (GAC) iff for every variable in the constraint and each one of its values there exist compatible values in all other variables involved in the constraint. A CSP is (G)AC iff all constraints are (G)AC and there are no empty domains.

Several consistencies have been defined for binary CSPs. Most can be described as specializations of (i, j)-consistency. A problem is (i, j)-*consistent* iff it has non-empty domains and any consistent instantiation of i variables can be extended to a consistent instantiation involving j additional variables [7]. Under this definition, a problem is AC iff it is $(1, 1)$-consistent. A problem is *path-consistent* (PC) iff it is $(2, 1)$-consistent. A problem is *path inverse consistent* (PIC) iff it is $(1, 2)$-consistent. In addition, a number of consistencies that cannot be described as (i, j)-consistencies have been defined. A problem P is *singleton arc consistent* (SAC) iff it has non-empty domains and for any instantiation (x, a) of a variable x, the resulting subproblem, denoted by $P_{(x,a)}$, can be made AC. A problem is *restricted path consistent* (RPC) iff any pair of instantiations (x, a), (y, b) of variables x and y, such that (y, b) is the only support of (x, a) on the constraint between x and y, can be consistently extended to any third variable.

Following [5], we call a consistency property A *stronger* than B iff in any problem in which A holds then B holds, and *strictly stronger* iff it is stronger and there is at least one problem in which B holds but A does not. We call a local consistency property A *incomparable* with B iff A is not stronger than B nor vice versa. Finally, we call a local consistency property A *equivalent* to B iff A is stronger than B and vice versa.

A propositional theory T is a set (conjunction) of CNF clauses of the form $l_1 \vee l_2 \vee \ldots \vee l_n$, where each l_i, $1 \leq i \leq n$, is a *literal*, ie. an atom or its negation. A clause can be alternatively denoted as $\{l_1, l_2, \ldots, l_n\}$. Finally the notation $x_1 x_2 \ldots x_n L$ denotes the clause $\{x_1\} \cup \{x_2\} \cup \ldots \cup \{x_n\} \cup L$. If c is a clause, $at(c)$ denotes the set of atoms of c. We assume that the reader is familiar with the basics of propositional satisfiability.

The most common propagation method used in SAT algorithms is *Unit Propagation* (*UP*) that repeatedly applies *unit resolution* (*UR*) to the clauses of the input theory. Among stronger propagation methods, one of the earliest is the *Failed Literal Detection* rule [6] denoted as *FL rule* or simply *FL*. Given a literal l in T, s.t. $\{\neg l\} \notin T$ and $\{l\} \notin T$, the FL rule assigns the value true (false) to l iff $UP(T \cup \{\neg l\})$ $(UP(T \cup \{l\}))$ derives the empty clause. We call *FL-prop* the propagation scheme that repeatedly applies the FL rule until no more variable values can be inferred or the empty clause is derived.

Another class of methods that are employed in state-of-the-art SAT solvers and preprocessing algorithms is *binary resolution*, in its general or various restricted forms. Binary resolution resolves two clauses of the form xy and $\neg xZ$ and generates the clause yZ. A restricted form of binary resolution, called BinRes, has been introduced in [1], that requires that both resolvents are binary. The application of BinRes as a propagation method, denoted by BinRes-prop, consists of repeatedly adding to the theory all new binary and unit clauses produced by resolving pairs of binary clauses and performing UP on any unit clauses that appear until nothing new is produced (or a contradiction is achieved). Note that BinRes-prop is a weaker propagation method than FL-prop [1]. Another restricted form of binary resolution is *Krom-subsumption resolution* (*KromS*) [13] that takes as input two clauses of the form xy and $\neg xyZ$ and generates the clause yZ. Note that yZ subsumes $\neg xyZ$, therefore $\neg xyZ$ can be deleted. *Generalized Subsumption resolution* (*GSubs*) takes two clauses xY and $\neg xYZ$ and generates YZ. The propagation methods derived by repeatedly applying KromS or GSubs are denoted by KromS-prop and GSubs-prop respectively.

3 Encodings

From CSP to SAT We restrict our study to binary CSPs. Many ways to translate a binary CSP into SAT have been proposed [10,14,9,12]. We focus on two of the most widely studied ones; the direct and the support encodings.

Direct Encoding: In the *direct encoding* a propositional variable x_{ia} is introduced for each value a of a CSP variable x_i. For each $x_i \in X$, there is an at-least-one clause $x_{i1} \vee \ldots \vee x_{id}$ to ensure that x_i takes a value. We optionally add at-most-one clauses that ensure that each CSP variable takes at most one value: for each i, a, b with $a \neq b$, we add $\neg x_{ia} \vee \neg x_{ib}$. Finally, for each constraint C on variables $\{x_i, x_j\}$ and for each a, b, s.t. tuple $<(x_i, a), (x_j, b)>$ is not allowed, we add $\neg x_{ia} \vee \neg x_{jb}$.

Support Encoding: The *support encoding* also introduces a propositional variable x_{ia} for each value a of a CSP variable x_i. We also have all the at-least-one clauses and (optionally) the at-most-one clauses. To capture the constraints, there are clauses that express the supports that values have in the constraints. For each binary constraint C on variables $\{x_i, x_j\}$ and for each $a \in D(x_i)$, we add $x_{jb_1} \vee \ldots \vee x_{jb_s} \vee \neg x_{ia}$, where $x_{jb_1}, \ldots, x_{jb_s}$ are the propositional variables that correspond to the s supporting values that a has in $D(x_j)$.

From SAT to CSP The following three are standard ways to translate a SAT instance into a CSP.

Literal Encoding: In the *literal encoding* of a SAT problem T a variable v_i is introduced for each clause c_i in T. $D(v_i)$ consists of those literals that satisfy c_i. A binary constraint is posted between two variables v_i and v_j iff clause c_i contains a literal l whose complement is contained in clause c_j. This constraint rules out incompatible assignments for the two variables (e.g. (v_i, l) and $(v_j, \neg l)$).

Dual Encoding: In the *dual encoding* of a SAT problem T a dual variable v_i is introduced for each clause c_i in T. $D(v_i)$ consists of those tuples of truth values that satisfy c_i. A binary constraint is posted between any two dual variables which correspond to clauses that share propositional variables. Such a constraint ensures that shared propositional variables take the same values in the tuples of both dual variables, if they appear with the same sign in the original clauses, and complementary values if they appear with opposite signs.

Non-Binary Encoding: In the *non-binary encoding* of a SAT problem T there is 0-1 variable for each propositional variable. A non-binary constraint is posted between variables that occur together in a clause. This constraint disallows the tuples that fail to satisfy the clause.

4 Encoding SAT as CSP

4.1 Literal Encoding

We denote by $L(T)$ the translation of a propositional theory T under the literal encoding. From [3,14] we know that there is a direct correspondence between UP and AC. We now study stronger consistency levels.

From [5] we know that PIC and RPC are stronger than AC for general CSP problems. Below we provide a characterization of the cases where these consistency methods lead to domain reductions in the literal encoding of a propositional theory.

Proposition 1. *Value l in the domain of a variable in the literal encoding $L(T)$ of a propositional theory T is not PIC iff T either contains the unit clause $\neg l$ or the unit clauses m and $\neg m$ or the clauses m and $\neg l \vee \neg m$ or the clauses $\neg l \vee m$ and $\neg l \vee \neg m$.*

Proof. The "if" part is straightforward. We will prove the "only if" part. Assume that the value l from the domain of variable v_i, corresponding to the clause c_i in T, is not PIC because it can not be extended to the variables v_j and v_k. If the value l has no support in any of the variables v_j or v_k, T must contain the unit clause $\{\neg l\}$. Assume now that this is not the case. Suppose that the domain of variable v_j contains only the value m (ie. corresponds to a unit clause in T). If the pair of values l and m can not be extended to variable v_k, it must be the case that the domain of v_k is either $\{\neg m\}$, or $\{\neg l\}$, or $\{\neg l, \neg m\}$. Consider the case now where the domain of v_j contains more than one value. Furthermore, assume that it does not contain the value $\neg l$. Then, l can form a consistent triple of values involving variables v_i, v_j and v_k. Therefore, for l to be not PIC, v_j must contain the value $\neg l$. Moreover, if v_j has more than two values in its domain (including $\neg l$) the value assignment of l to v_i can be extended to the other two variables. Therefore, v_j must have exactly two values in its domain, one of which is $\neg l$, ie., it is of the form $\{\neg l, m\}$. Since the values l and m cannot be extended to v_k, we conclude that the domain of v_k is of the form $\{\neg l, \neg m\}$. □

A similar result holds for RPC. The proof is similar to the above.

Proposition 2. *Value l in the domain of a variable in the literal encoding $L(T)$ of a propositional theory T is not RPC iff T either contains the unit clause $\neg l$ or the unit clauses m and $\neg m$ or the clauses m and $\neg l \vee \neg m$ or the clauses $\neg l \vee m$ and $\neg l \vee \neg m$.*

From the above analysis we conclude that on the literal encoding of a SAT problem, PIC collapses down to RPC. Note that in general binary CSPs with more than 3 variables, PIC is strictly stronger than RPC [5]. These results also imply that BinRes-prop applied to T is at least as strong as enforcing PIC (and therefore RPC) on $L(T)$:

Proposition 3. *If enforcing PIC on the literal encoding $L(T)$ of a propositional theory T deletes the value l form the domain of a variable of $L(T)$ then BinRes-prop in T generates the unit clause $\neg l$ or determines that T is unsatisfiable.*

Proof. Inductively on the values $l_1, l_2, \ldots, l_n$ deleted by PIC enforcement.
Base case: From proposition 1 follows that if l_1 is not PIC, BinRes either derives $\neg l_1$ or determines that T is unsatisfiable (in the case where T contains m and $\neg m$).
Inductive Hypothesis: Assume that the proposition holds for all deletions l_i, $1 \leq i < k < n$. That is, if PIC on $L(T)$ deletes the values $\{l_1, l_2, \ldots, l_{k-1}\}$, BinRes-prop on T derives the unit clauses $\{\neg l_1, \neg l_2, \ldots, \neg l_{k-1}\}$.
Inductive Step: Assume that l_k is not PIC in $L(T)$. By analysis of the cases of proposition 1 we conclude that BinRes-prop either generates $\neg l_k$, or determines that T is unsatisfiable.
Assume that l_k is PIC in $L(T)$, but is not PIC in the problem resulting after the deletion of the values $\{l_1, l_2, \ldots, l_{k-1}\}$. There are two cases.
First, there is a variable v_j such that all the supports of value l_k have been deleted from the domain of v_j by PIC. The domain of variable v_j is either $\{\neg l_k, l'_1, l'_2, \ldots, l'_n\}$ or $\{l'_1, l'_2, \ldots, l'_n\}$, and $\{l'_1, l'_2, \ldots, l'_n\} \subseteq \{l_1, l_2, \ldots, l_{k-1}\}$. From the inductive hypothesis follows that BinRes-prop entails the unit clauses $\neg l'_1$, $\neg l'_2, \ldots, \neg l'_n$. Then, BinRes-prop, either derives the unit clause $\neg l_k$, or determines that T is unsatisfiable.

Assume now that l_k has support in domain of v_j after the deletions $\{l_1, l_2, \ldots, l_{k-1}\}$ performed by PIC. This means that c_j, the clause of T that correspond to v_j, is of the form $A \cup D \cup R$, where $A \subseteq \{\neg l_k\}$, $D \subseteq \{l_1, l_2, \ldots, l_{k-1}\}$, $R = c_j - (A \cup D)$, with $|R| > 0$. Assume that $R = \{m\}$. If m has no support in the initial domain of variable v_k, then this domain must be $\{\neg m\}$. Then, BinRes-prop, either derives the unit clause $\neg l_k$, if $A = \{\neg l_k\}$, or determines that T is unsatisfiable, if $A = \emptyset$.
Assume now that $|R| > 1$, and let $R = \{m_1, m_2, \ldots, m_n\}$, with $n \geq 2$. Observe that all elements of R are supports for l_k in v_j. Suppose now that l_k is not PIC because none of the pairs of values (l_k, m_f), $1 \leq f \leq n$, can be extended to a consistent triple involving a value from some variable v_g. Let the domain of v_g be of the form $B \cup S$ where $B \subseteq \{\neg l\}$. Note that any value of S is support for some value m_f, $1 \leq f \leq n$. Since l_k is not PIC after the deletions of values $\{l_1, l_2, \ldots, l_{k-1}\}$, we conclude that $S \subseteq \{l_1, l_2, \ldots, l_{k-1}\}$. From the inductive hypothesis we know that BinRes-prop entails the unit clauses $\neg l_1$, $\neg l_2, \ldots, \neg l_{k-1}$. Therefore, BinRes-prop will generate, using the clause that corresponds to variable v_g, either the unit clause $\neg l_k$ or the empty clause. □

The following example shows that BinRes-prop is strictly stronger than PIC.

Example 1. Consider the propositional theory $T = \{l_1 \vee l_2, \neg l_1 \vee l_3, \neg l_2 \vee l_3, \neg l_3 \vee l_4\}$. BinRes derives the unit clause l_3, but enforcing PIC on $L(T)$ does not lead to any domain reductions.

Not surprisingly, we can exploit the direct correspondence between AC and UP shown in [14] to prove that FL-prop on T is equivalent to enforcing SAC in $L(T)$. The proof proceeds by induction on the number of deletions performed by SAC and the number of assignments made by FL.

Proposition 4. *Enforcing SAC on the literal encoding $L(T)$ of a propositional T deletes value l from the domains of all variables of $L(T)$ iff FL-prop on T assigns false to l.*

4.2 Dual Encoding

We denote by $D(T)$ the translation of a propositional theory T under the dual encoding. From [14] we know that AC applied to the dual encoding can achieve more propagation than UP in the original SAT instance.

As we will show, AC on the dual encoding can achieve a very strong level of consistency that cannot be captured by known propagation methods in the original SAT problem. For instance, the following example demonstrates that FL on T and AC on $D(T)$ are incomparable.

Example 2. Consider the theory $T = \{l_1 \vee l_4, \neg l_1 \vee l_2, \neg l_1 \vee l_3, \neg l_2 \vee \neg l_3\}$. Note that FL will assign the value F to l_1. The problem $D(T)$ is AC, therefore no domain reduction is performed by AC.
Consider now the theory T that contains all possible clauses in three variables, ie., $l_1 \vee l_2 \vee l_3, l_1 \vee l_2 \vee \neg l_3, \ldots, \neg l_1 \vee \neg l_2 \vee \neg l_3$. AC on $D(T)$ will lead to a domain wipeout, whereas FL does not lead to any simplifications.

It is not difficult to show that FL on T is weaker than *SAC* on $D(T)$.

Proposition 5. *If FL assigns true to literal l of a propositional theory T, then all variable values of $D(T)$ that correspond to valuations that assign false to l are not SAC.*

To precisely identify the propagation achieved by AC on the dual encoding, we first provide two characterizations of propositional theories that are not AC under the dual encoding. The first is a general characterization based on the form of the propositional theory T, whereas the second describes the form of the values that are not AC.

Proposition 6. *A value in the domain of a variable x_i of $D(T)$ that corresponds to the clause c_i in T is not AC iff T contains a clause c_j such that $at(c_j) \subseteq at(c_i)$ and there exists l such that $l \in c_j$ and $\neg l \in c_i$.*

Let x_i be a variable in the dual encoding $D(T)$ of a propositional theory T, that corresponds to a clause c_i of T defined on the set of atoms $at(c_i) = \{a_1, a_2, \ldots, a_n\}$. We assume an order on the set of elements of $at(c_i)$ which, if not otherwise stated, corresponds to the order of appearance of the atoms in c_i. We denote this by $at(c_i) = (a_1, a_2, \ldots, a_n)$. We use this order to refer to the values of variable x_i of $D(T)$ as follows. A value in the domain of x_i is a tuple $v = (v_1, v_2, \ldots, v_n)$, where $v_i \in \{T, F\}$ denotes the value of atom a_i, with $1 \leq i \leq n$. Alternatively a value for variable x_i is a tuple $v = (l_1, l_2, \ldots, l_n)$, where $l_i = a_i$ if a_i is assigned true in v and $l_i = \neg a_i$ if a_i is assigned false in v.

Proposition 7. *Value $v = (l_1, l_2, \ldots, l_n)$ in the domain of variable x_i in the dual encoding $D(T)$ of a propositional theory T is not AC iff T contains a clause $c_j = l'_1 \vee l'_2 \vee \ldots \vee l'_m$ such that for each l'_k, $1 \leq k \leq m$ it holds that $l'_k = \neg l_p$ for some $1 \leq p \leq n$.*

Note that the above results are valid for a CSP $D(T)$ that is the dual encoding of a propositional theory T. Enforcing AC on $D(T)$ may lead to a sequence of domain reductions, leading to a CSP that does not necessarily correspond to the dual encoding of an associated simplification of theory T.

We now introduce *subset resolution*, a form of resolution that is stronger than GSubs, and is intended to capture the domain reductions performed when enforcing AC.

Definition 1. Subset resolution *resolves two clauses c_i and c_j of a theory T iff T contains a clause c such that $at(c_i) \subseteq at(c)$ and $at(c_j) \subseteq at(c)$.*

We denote by *SubRes-prop* the propagation algorithm obtained by repeatedly applying subset resolution. The following result shows that SubRes-prop is at least as strong as enforcing AC.

Proposition 8. *Let x_i be a variable in the dual encoding $D(T)$ of a propositional theory T that corresponds to clause c_i of T such that $at(c_i) = (a_1, a_2, \ldots, a_n)$. If enforcing AC deletes the value $v = (l_1, l_2, \ldots, l_n)$ from the domain of variable x_i, then either T contains or SubRes-prop generates the clause $l'_1 \vee l'_2 \vee \ldots \vee l'_m$ such that for each l'_k, $1 \leq k \leq m$ it holds that $l'_k = \neg l_p$ for some $1 \leq p \leq n$.*

Proof. By induction on the sequence of values $v_1, v_2, \ldots, v_f$ deleted by the AC enforcing algorithm.
Base Case: Follows from Proposition 7.
Inductive Hypothesis: Assume that the proposition holds for all v_i ,$1 \leq i < k < f$.
Inductive Step: Assume that the AC enforcing algorithm, after deleting the values $v_1, v_2, \ldots, v_{k-1}$, deletes the value v_k from the domain of variable x_i of $D(T)$ because it has no support in the domain of variable x_j, that corresponds to clause c_j in T. The case where the original domain of x_j contained no support for v_k is covered by Proposition 7. Consider now the case where value v_k has the supports $S = \{s_1, s_2, \ldots, s_r\}$ in the original domain x_j, but these supports are deleted by AC. Define $A = at(c_i) \cap at(c_j)$. Consider first the case where $A = \emptyset$. Then, S coincides with the domain of x_j, ie. it contains all possible valuations on $at(c_j)$, except the valuation that falsifies c_j. The fact that v_k has no support in x_j means that the domain of x_j is empty. Therefore, the AC enforcing algorithm must have been terminated, which contradicts the assumption that it deletes v_k. Hence, this situation can never arise.

Consider now the case where $A \neq \emptyset$, and let v_k^A be the projection of value v_k on the atoms of A, defined as $v_k^A = (l_1, l_2, \ldots, l_m)$. The set S contains all possible valuations on the set of atoms $at(c_j)$ that assign the same values as v_k to the variables of A, except the valuation that falsifies c_j. Since the AC enforcing algorithm deletes value v_k, all the supports of v_k in the domain of variable x_j (ie. the elements of S) must have been deleted by the algorithm, and therefore belong to the set of values $v_1, v_2, \ldots, v_{k-1}$. From the inductive hypothesis we know that SubRes-prop derives a set of clauses $R = \{c'_1, c'_2, \ldots, c'_q\}$, $q \leq r$, that satisfy the properties stated in the proposition. Therefore, for each possible assignment on the atoms of $at(c_j)$ that assigns the same value as v_k on the atoms of A, the set $R \cup \{c_j\}$ contains a clause c' such that the assignment is not a model of c'. Hence, $R \cup \{c_j\}$ has no models where all the atoms of A are assigned the same values as in v_k. Therefore, $R \cup \{c_j\} \models \neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m$. Since for each clause c_R of $R \cup \{c_j\}$ it holds that $at(c_R) \subseteq at(c_j)$, SubRes-prop is able to derive, from the set of clauses $R \cup \{c_j\}$, a prime implicant that subsumes $\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m$. □

The next result is the reciprocal of the previous proposition and both together imply that enforcing AC on $D(T)$ is equivalent to SubRes-prop in T.

Proposition 9. *Given a propositional theory T, if SubRes-prop on T derives a clause $l_1 \vee l_2 \vee \ldots \vee l_m$, then enforcing AC on $D(T)$ deletes all values $(l'_1, l'_2, \ldots, l'_n)$ from the domains of the variables of $D(T)$ such that for each l_i, $1 \leq i \leq m$ there is some l'_j, $1 \leq j \leq n$, such that $l'_j = \neg l_i$.*

Proof. By induction on the the set of clauses $S = \{c_1, c_2, \ldots, c_k\}$ defined inductively as follows: $c_i \in S$ if $c_i \in T$ or c_i is the subset resolvent of two clauses $c_a, c_b \in S$.

Base Case: The proposition follows for all clauses of T from proposition 7.

Inductive Step: Let $c = c_1 \cup c_2$ be the clause that is generated by subset resolution from the clauses $c'_1 = \{l\} \cup c_1$ and $c'_2 = \{\neg l\} \cup c_2$. Our inductive hypothesis is that the proposition holds for clauses c'_1 and c'_2. Let $c_1 = \{l_1^1, l_2^1, \ldots, l_x^1\}$ and $c_2 = \{l_1^2, l_2^2, \ldots, l_y^2\}$. Since c'_1 and c'_2 are resolved by subset resolution, it must hold that T contains a clause c_g such that $at(c'_1) \subseteq at(c_g)$ and $at(c'_2) \subseteq at(c_g)$. Let x_g be the corresponding variable of $D(T)$. From the inductive hypothesis we know that enforcing AC deletes all values of x_g with projection $(\neg l, \neg l_1^1, \neg l_2^1, \ldots, \neg l_x^1)$ and $(l, \neg l_1^2, \neg l_2^2, \ldots, \neg l_y^2)$ on $at(c'_1)$ and $at(c'_2)$ respectively. Therefore, x_g can not contain a value in its domain with projection[1] $(\neg l_1^1, \neg l_2^1, \ldots \neg l_x^1, \neg l_1^2, \neg l_2^2, \ldots, \neg l_y^2)$ (no value at all if $c_1 = c_2 = \emptyset$). Moreover, no domain of the other variables can include a value with projection $(\neg l_1^1, \neg l_2^1, \ldots, \neg l_x^1, \neg l_1^2, \neg l_2^2, \ldots, \neg l_y^2)$ on the set $at(c'_1) \cup at(c'_2)$, as it has no support in the domain of x_g. □

Having introduced a new resolution technique to that is equivalent to AC on the dual encoding, we can also define similar methods that capture even higher consistency levels. We now introduce *extended subset resolution*, a slightly extended form of subset resolution, that is intended to capture the domain reductions performed by PIC on the dual encoding. Note that if $c = \{l_1, l_2, \ldots, l_n\}$ is a clause, $\overline{c} = \{\neg l_1, \neg l_2, \ldots, \neg l_n\}$.

[1] To facilitate the discussion we assume that $c_1 \cap c_2 = \emptyset$.

Definition 2. Extended Subset resolution *(ESR) resolves two clauses c_i and c_j of a theory T if either T contains a clause c such that $at(c_i) \subseteq at(c)$ and $at(c_j) \subseteq at(c)$ or $c_i \cap \overline{c_j} = \{l\}$ and T contains a clause c such that $at(c_i) - at(\{l\}) \subseteq at(c)$ and $at(c_j) - at(\{l\}) \subseteq at(c)$.*

We now characterize the cases where PIC performs domain reductions.

Proposition 10. *Value $v = (l_1, l_2, \ldots, l_n)$ in the domain of a variable x of the dual encoding $D(T)$ of a theory T is not PIC iff T either contains a clause $l'_1 \vee l'_2 \vee \ldots \vee l'_m$ or a pair of clauses $l'_1 \vee l'_2 \vee \ldots l'_a \vee l'$ and $l'_{a+1} \vee l'_{a+2} \vee \ldots l'_m \vee \neg l'$ such that for each l'_i, $1 \leq i \leq m$ there is some l_j, $1 \leq j \leq n$, such that $l'_i = \neg l_j$.*

Based on the above, the following result shows the relation between the clauses generated by the propagation algorithm *ESR-prop* that repeatedly applies ESR on T, and the domain reductions performed by a PIC enforcing algorithm on $D(T)$. The proof proceeds in a similar fashion as the proof of Proposition 9.

Proposition 11. *If ESR-prop on a propositional theory T derives a clause $l_1 \vee l_2 \vee \ldots \vee l_n$, then enforcing PIC on $D(T)$ deletes all values $(l'_1, l'_2, \ldots, l'_m)$ from the domains of the variables of $D(T)$ such that for each l_i, $1 \leq i \leq n$ there is some l'_j, $1 \leq j \leq m$, such that $l'_j = \neg l_i$.*

4.3 Non-binary Encoding

We denote by $N(T)$ the translation of a propositional theory T under the non-binary encoding. Note that in $N(T)$ all clauses of T involving the same set of variables are encoded together in one constraint. From [14] we know that under the non-binary encoding, GAC is stronger than UP. We first show thatif the propositional theory T does not contain clauses that are on the same variables, GAC on $N(T)$ can do no more pruning than UP on T. The proof uses induction in the number of deletions performed by GAC.

Proposition 12. *Let T be a propositional theory that does not contain two clauses c_i and c_j such that $at(c_i) = at(c_j)$, and let x be a variable in T. If GAC deletes the value a from $D(x)$ in $N(T)$ then UP assigns the value $\neg a$ to x in T.*

If the above restriction does not hold then GAC enforced on $N(T)$ can achieve a high level of consistency. The following example shows that FL-prop, BinRes-prop, and KromS-prop are all incomparable to enforcing GAC.

Example 3. Let T be the theory containing all possible clauses in three variables: $l_1 \vee l_2 \vee l_3, l_1 \vee l_2 \vee \neg l_3, \ldots, \neg l_1 \vee \neg l_2 \vee \neg l_3$. FL-prop, BinRes-prop and KromS-prop on this theory do not lead to any simplifications, whereas GAC on $N(T)$ shows that the problem is not solvable. Note that GSubs-prop applied to T also determines insolubility.

Now consider the theory $l_1 \vee l_4, \neg l_1 \vee l_2, \neg l_1 \vee l_3, \neg l_2 \vee \neg l_3$. FL-prop and BinRes-prop determine that l_1 must be assigned false, whereas GAC leads to no reductions. Finally, consider the theory $l_1 \vee l_2 \vee l_3, \neg l_1 \vee l_2, \neg l_2 \vee l_3$. KromS-prop determines that l_3 must be assigned true, whereas GAC leads to no reductions.

To put an upper bound in the pruning power of GAC, we first characterize the cases where a value is not GAC in $N(T)$.

Proposition 13. *Value 0 (1) of a variable x_i in the non-binary encoding $N(T)$ of a propositional theory T is not GAC iff T contains either the unit clause x_i ($\neg x_i$) or all possible clauses in variables $x_i, x_1, \ldots, x_k$ ($k \geq 1$) that include literal x_i ($\neg x_i$).*

Using the above proposition, we can prove that GSubs-prop applied to T is strictly stronger than enforcing GAC on $N(T)$.

Proposition 14. *Let T be a propositional theory. If enforcing GAC deletes the value 0 (1) from the domain of a variable in $N(T)$ then GSubs-prop assigns the value 1 (0) to the corresponding variable in T.*

Proof. Suppose GAC makes a sequence of value deletions $(x_1, a_1), (x_2, a_2), \ldots, (x_j, a_j)$. The proof uses induction on j.
Base Case: The first value $a_1 \in x_1$ will be deleted because it has no support in some constraint C in $N(T)$. From Proposition 13 we know that either T contains the unit clause x_1, if $a_1 = 0$ ($\neg x_1$ if $a_1 = 1$), or all possible clauses in the variables of the constraint that include literal x_1, if $a_1 = 0$, and $\neg x_1$ if $a_1 = 1$. In both cases, if we apply GSubs-prop on the clauses, we will entail x_1 ($\neg x_1$).
Inductive Hypothesis: We assume that for any $1 \leq m < j$ the proposition holds.
Inductive Step: Consider the final deletion of value a_j from $D(x_j)$. This value is deleted because it has no supporting tuple in some constraint C on variables $x_j, y_1, \ldots, y_k$, which corresponds to one or more clauses on the corresponding propositional variables in T. This means that for each tuple τ that supported value a_j, at least one of the values in τ has been deleted from a variable among $y_1, \ldots, y_k$. According to the hypothesis, for every such deletion, the corresponding propositional variable was set to the opposite value by GSubs-prop, and the associated literals were set to false in all the associated clauses. Now consider the subset $y_l, \ldots, y_m$ of $y_1, \ldots, y_k$ which consists of the variables that have not been set. Assume that the (reduced) clauses associated with constraint C do not contain all possible combinations of literals for variables $y_l, \ldots, y_m$. Then C will contain at least one supporting tuple for a_j which contradicts the fact that a_j is deleted. Therefore, the clauses associated with constraint C contain all possible combinations of literals for variables $y_l, \ldots, y_m$. If we apply GSubs-prop on these clauses, we will entail x_j, if $a_j = 0$, and $\neg x_j$ if $a_j = 1$. □

From the above proposition and the last problem in Example 3, it follows that GSubs-prop is strictly stronger than enforcing GAC.

5 Encoding CSP as SAT

5.1 Direct Encoding

We denote by $Di(P)$ the translation of a CSP P into SAT under the direct encoding. From [14] we know that AC enforced on a CSP P can achieve more pruning than UP applied to $Di(P)$.

The following example demonstrates that FL-prop is incomparable to PIC while BinRes-prop is incomparable to AC, RPC, and PIC.

Example 4. Consider a CSP with four variables x_1, x_2, x_3, x_4, all with $\{0, 1\}$ domains, and constraints $x_1 = x_2$, $x_2 = x_3$, $x_3 = x_4$ and $x_4 \neq x_1$. This problem is AC, RPC, and PIC. However, by setting x_{10} or x_{11} to true in the direct encoding, UP generates the empty clause. Therefore, FL-prop sets both x_{10} and x_{11} to false and determines that the problem is unsatisfiable. Accordingly, BinRes-prop generates the unit clauses $\neg x_{10}$ and $\neg x_{11}$ and therefore determines unsatisfiability.

Now consider a CSP with three variables x_1, x_2, x_3, all with $\{0, 1, 2, 3\}$ domain, and three constraints. Assume that values 0 and 1 (2 and 3) of x_1 are supported by 0 and 1 (2 and 3) in $D(x_2)$ and $D(x_3)$, and that values 0 and 1 (2 and 3) of $D(x_2)$ are supported by 2 and 3 (0 and 1) in $D(x_3)$. FL-prop on the direct encoding of the problem does not generate the empty clause by setting any propositional variable to true or false. Therefore is does not achieve any inference. However, the problem is not PIC.

Finally, consider a CSP with two variables x_1 and x_2, both with three values in their domains, where one of the values in $D(x_1)$ has no support in $D(x_2)$. AC (and all stronger CSP consistencies) will delete this value. However, BinRes-prop cannot resolve any clauses and therefore infers nothing.

We now show that if a value in a CSP is not RPC then the FL rule will set the corresponding variable to false in $Di(P)$.

Proposition 15. *If a value $a \in D(x_i)$ of a CSP P is not RPC then the FL rule assigns x_{ia} to false when applied to $Di(P)$.*

Proof. Assume that value $a \in D(x_i)$ of a CSP P is not RPC. This means that (x_i, a) has a single support (say $b \in D(x_j)$) in the constraint between x_i and x_j and the assignments (x_i, a) and (x_j, b) cannot be consistently extended to some third variable x_l. That is, there is no value in $D(x_l)$ that supports both (x_i, a) and (x_j, b). Now consider the direct encoding of the problem and assume that FL sets x_{ia} to true. UP will immediately set each $x_{jb'}$, where $b' \in D(x_j)$ and $b' \neq b$, to false, and as a result x_{jb} will be set to true. UP will then set all propositional variables corresponding to the values in $D(x_l)$ to false. Therefore, an empty clause is generated and as a result x_{ia} will be set to false. □

A corollary of the above proposition is that FL sets the corresponding propositional variable to false for any value of the original CSP that is not AC. The first problem in Example 4 together with Proposition 15 can help prove that FL-prop is strictly stronger than RPC and AC. The complete proof involves induction in the number of deletions performed by the algorithms. We now state that if FL-prop makes an inference in $Di(P)$ then SAC also makes the corresponding inference in P.

Proposition 16. *If FL-prop sets a propositional variable x_{ia} to false in the direct encoding of a CSP P then SAC deletes value a from $D(x_i)$ in P. If FL-prop sets x_{ia} to true then SAC deletes all values in $D(x_i)$ apart from a.*

Now consider the second problem in Example 4. This problem is not SAC, but FL-prop applied to its direct encoding infers nothing. This, together with the above proposition, prove that SAC is strictly stronger than FL-prop, and therefore also BinRes-prop.

We now show that GSubs applied to $Di(P)$ can do no more work than Unit Resolution.

Proposition 17. *Given a CSP P, GSubs applied to $Di(P)$ draws an inference iff only Unit Resolution does.*

Proof. It suffices to show that GSubs can only be applied to pairs of clauses of the form x, $\neg xY$. Consider two clauses of the form xY, $\neg xYC$. First assume that both clauses are non-binary. In this case both clauses are necessarily at-least-one clauses. But if x is present in one of the clauses, $\neg x$ cannot be present in the other, since the two clauses encode domains of different variables. Now assume that at least one of the clauses is of the form xy. There are three cases depending on what kind of clause xy is. If xy is a conflict clause then $\neg x$ can only be found in an at-least-one clause. However, literal y cannot be present in this clause. If xy is an at-most-one clause then $\neg x$ can again only be found in the corresponding at-least-one clause. However, this clause will contain $\neg y$ and not y. Finally, if xy is an at-least-one clause then $\neg x$ and y cannot occur together in any conflict or at-most-one clause. Hence, in all cases GSubs cannot be applied. □

5.2 Support Encoding

We denote by $Sup(P)$ the translation of a CSP P into SAT under the support encoding. From [10,9] we know that AC applied to a CSP P is equivalent to UP applied to $Sup(P)$. We now elaborate on the relationship between FL-prop and SAC.

Proposition 18. *FL-prop sets a propositional variable x_{ia} to false in the support encoding of a CSP P iff value SAC deletes value $a \in D(x_i)$ in P. FL-prop sets x_{ia} to true iff SAC deletes all values in $D(x_i)$ apart from a.*

The proof is rather simple and proceeds by induction in the number of assignments made by FL-prop and the number of value deletions performed by SAC, exploiting the equivalence between AC and UP.

It is easy to see that BinRes-prop is strictly stronger than AC. Consider a problem with three variables x,y,z, each with domain $\{0, 1\}$ and constraints $x = y$, $x = z$ and $y \neq z$. This problem is AC, but BinRes-prop applied to its support encoding shows that it is inconsistent. This example, coupled with the fact that BinRes-prop subsumes UP (which is equivalent to AC), proves that BinRes-prop is strictly stronger than AC.

The following example demonstrates that BinRes-prop is incomparable to RPC, and PIC.

Example 5. Consider the first problem in Example 4. This problem is AC, RPC, and PIC. However, BinRes-prop generates the unit clauses $\neg x_{10}$ and $\neg x_{11}$ and therefore determines unsatisfiability.

Now consider a CSP with three variables x_1, x_2, x_3, all with $\{0, 1, 2, 3\}$ domain, and three constraints. Assume that value 0 of x_1 is supported by 0 in $D(x_2)$ and 0,1 in $D(x_3)$, and that value 0 of $D(x_2)$ is supported by 2,3 in $D(x_3)$. All other values have at least two supports in any constraint. The problem is not RPC (or PIC), but BinRes-prop applied to the support encoding does not detect the inconsistency.

As in the direct encoding, GSubs applied to $Sup(P)$ can do no more work than Unit Resolution. The proof proceeds by case analysis and is similar to the proof of Proposition 17.

Proposition 19. *Given a CSP P, GSubs applied to $Sup(P)$ draws an inference iff only Unit Resolution does.*

6 Conclusion

In this paper we presented theoretical results concerning the relative propagation power of various local consistency methods for CSP and SAT. More specifically, we studied AC, PIC, RPC and SAC for CSP, and FL, BinRes, KromS and GSubs for SAT. The results we obtained complement and tie together recent similar studies [3,9,14].

As it may be expected, in cases where AC is equivalent to Unit Propagation, SAC is equivalent to Failed Literal Detection. Under both translations of CSP to SAT we consider, FL can achieve a level of consistency that is higher than RPC in the original problem. Among the less powerful methods, that are nevertheless stronger than AC, BinRes arises as an appealing method that can achieve a relative high level of local consistency. Indeed, BinRes is stronger than PIC under the literal encoding, and stronger than AC under the support encoding. Generalized subsumption resolution finally achieves an intermediate level of consistency between GAC in $N(T)$ and AC in $D(T)$.

Comparing among different encodings, in the case of translating a SAT to CSP, the dual encoding appears to achieve the highest level of local consistency among the different approaches studied. Indeed, AC in the dual encoding corresponds to subset resolution, a method that is stronger than generalized subsumption resolution. Although the cost of applying such a local consistency method in its general form may be prohibitive, restricted versions (eg. restricting the number of literals in the clauses considered) of the method may have some practical value. The non-binary encoding appears weaker than the dual, and enforcing GAC on the translated problem can achieve better propagation than UP only if the original theory contains clauses on the same variables. Finally, under the literal encoding, techniques that have been used in SAT and CSP are roughly equivalent, with the exception of BinRes which achieves a level of consistency between PIC and SAC. When translating from CSP to SAT, the direct encoding appears to be rather weak. Indeed, strong resolution methods such as generalized subsumption resolution are weaker than AC in the original problem. The support encoding appears to behave better propagation-wise, as already suggested in [9]. A general conclusion that can been drawn from our analysis is that translating a CSP to SAT can be beneficial only if the support encoding is used, coupled with a propagation method that is at least as strong as BinRes.

An extended version of this paper contains a more detailed study of propagation in CSP and SAT, including CSP consistency methods such as Neighbor Inverse Consistency and SAT techniques such as Hyper-resolution. In the future we intend to extend our study to cover encodings of non-binary constraints into SAT, and also consider the relationship between learning techniques in CSP and SAT under the various encodings. Finally, but very importantly, an empirical evaluation of the different encodings and propagation methods is required.

References

1. F. Bacchus. Enhancing Davis Putnam with Extended Binary Clause Reasoning. In *Proceedings of AAAI-02*, pages 613–619, 2002.
2. H. Bennaceur. The satisfiability problem regarded as a constraint satisfaction problem. In *Proceedings of ECAI-1996*, pages 125–130, 1996.
3. H. Bennaceur. A Comparison between SAT and CSP Techniques. *Constraints*, 9:123–138, 2004.
4. C. Bessière, E. Hebrard, and T. Walsh. Local Consistencies in SAT. In *Proceedings of SAT-2003*, pages 400–407, 2003.
5. R. Debruyne and C. Bessière. Domain Filtering Consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
6. J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, 1995.
7. E. Freuder. A Sufficient Condition for Backtrack-bounded Search. *JACM*, 32(4):755–761, 1985.
8. E. Freuder and C. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proceedings of AAAI'96*, pages 202–208, 1996.
9. I. Gent. Arc Consistency in SAT. In *Proceedings of ECAI-2002*, pages 121–125, 2002.
10. S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, 1990.
11. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI-97*, pages 366–371, 1997.
12. S. Prestwich. Full Dynamic Substitutability by SAT Encoding. In *Proceedings of CP-2004*, pages 512–526, 2004.
13. A. van Gelder and Y. Tsuji. Satisfiability testing with more reasoning and less guessing. In *Cliques, Coloring and Satisfiability*, pages 559–586, 1996.
14. T. Walsh. SAT v CSP. In *Proceedings of CP-2000*, pages 441–456, 2000.

The *Minimum Spanning Tree* Constraint

Grégoire Dooms[1,⋆] and Irit Katriel[2,⋆⋆]

[1] Department of Computing Science and Engineering
Université catholique de Louvain, Belgium
`dooms@info.ucl.ac.be`
[2] BRICS[⋆⋆⋆], University of Aarhus, Denmark
`irit@daimi.au.dk`

Abstract. The paper introduces the $MST(G, T, W)$ constraint, which is specified on two graph variables G and T and a vector W of scalar variables. The constraint is satisfied if T is a minimum spanning tree of G, where the edge weights are specified by the entries of W. We develop algorithms that filter the domains of all variables to bound consistency.

1 Introduction

Complex constraints and variable types simplify the modelling task while providing the constraint solver with a better view of the structure of the CSP. The recently introduced CP(Graph) framework [4] allows the programmer to define *graph variables*, i.e., variables whose assigned values are graphs. The usefulness of graph variables depends on the existence of filtering algorithms for constraints defined on them. That is, a constraint solver that supports graph variables ideally supports a collection of constraints that describe fundamental graph properties.

In this paper we introduce the $MST(G, T, W)$ constraint, which is specified on two graph variables G and T and a vector W of scalar variables. The constraint is satisfied if T is a minimum spanning tree (MST) of G (i.e., the minimum-weight connected subgraph that contains all nodes of G), where the positive weights of the edges in G (and hence also in T) are specified by the entries of W.

Finding the MST of a graph takes almost-linear time, but several interesting variants of the MST problem, such as minimum k-spanning tree [8] (finding a minimum-weight tree that spans any set of k nodes) and Steiner tree [10] (finding a minimum-weight tree that spans a given set of nodes) are known to be NP-hard. Such problems can be modelled by a combination of MST and other constraints.

In other applications, there is uncertainty in the input, e.g., when the exact weight of an edge is not known, but can be assumed to belong to a given interval of values. In the widely-studied MST sensitivity analysis problem, we are given a fixed graph g with fixed weights and an MST t of g. We need to determine, for each edge, the amount by which its weight can be changed without violating $t = MST(g)$. The robust spanning tree problem [1] addresses uncertainty from a different point of view. Its input is a graph

⋆ Supported by the Walloon Region, project BioMaze (WIST 315432).
⋆⋆ Supported by the Danish Research Agency (grant # 272-05-0081).
⋆⋆⋆ Basic Research in Computer Science, funded by the Danish National Research Foundation.

with a set of possible edge weights for every edge. A *scenario* is a selection of a weight for each of the edges. For a spanning tree t and a scenario s, the *regret* of t for s is the difference between the weight of t and the weight of the MST of the graph under scenario s. The output is a spanning tree that minimizes that worst-case regret.

Finally, in inverse parametric optimization problems [7] we are interested in finding parameters of an optimization problem, given its solution. For instance, our MST constraint allows us to receive a tree and determine which graphs have this tree as their MST.

Our results are summarized in the table below. We look at various restrictions of the MST constraint, and develop a bound consistency algorithm for each of them. Let n and m be, respectively, the number of nodes and edges in the upper bound of the domain of G (and hence also of T). Let $Sort(m)$ be the time it takes to sort m edge-weights and α the slow-growing inverse-Ackerman function. As the table indicates, our algorithm computes bound consistency for the most general case in cubic time but whenever the domain of one of the variable is fixed, bound consistency can be computed in almost-linear time. The table indicates in which section of the paper each case is handled.

	Fixed Edge-Weights		**Non-Fixed Edge-Weights**	
	Fixed Graph	Non-Fixed Graph	Fixed Graph	Non-Fixed Graph
Fixed Tree	MST verification $O(m+n)$ [11]	$O(m+n)$ [Section 4.1]	$O(m\alpha(m,n))$ [Section 5.1]	$O(m\alpha(m,n))$ [Section 5.2]
Non-Fixed Tree	$O(Sort(m)+ m\alpha(m,n))$ [Section 4.2]	$O(Sort(m)+ m\alpha(m,n))$ [Section 4.3]	$O(Sort(m)+ m\alpha(m,n))$ [Section 5.3]	$O(mn(m+\log n))$ [Section 5.4]

Related Work. Filtering algorithms have been developed for several constraints on graphs, such as Sellmann's shorter paths constraints [14], unweighted forest constraints (directed [2] and undirected [13]) and the Weight-Bounded Spanning-Tree constraint [5], $WBST(G, T, W, I)$, which specifies that T is a spanning tree of G of weight at most I, where W is again a vector of edge weights. Although $WBST$ is semantically close to MST, the structure of the solution set is different and the bound consistency algorithms that we have developed are very different from the ones described in this paper. Perhaps the most obvious difference between the two constraints is that for $WBST$, in the most general case (when all three variables are not fixed) it is NP-hard to decide whether a solution exists (and hence also to filter the constraint to bound consistency). For MST, it is possible to do this in polynomial time. From the application point of view, $WBST$ is a cost-based filtering constraint (i.e., an optimization constraint) and it can be used in conjunction with the MST constraint to get more pruning.

In [1], I. Aron and P. Van Hentenryck addressed the robust spanning tree problem with interval data. They describe an algorithm that partially solves a special case of the filtering problem that we address in this paper. More precisely, for the case in which the graph is fixed, they show how to detect which edges must be removed from the upper bound of the tree domain. Finally, our filtering algorithms apply techniques that were previously used in King's MST verification algorithm [11] and Eppstein's algorithm for computing the k smallest spanning trees [6] of a graph.

Roadmap. The rest of the paper is structured as follows. Section 2 contains some preliminaries, definitions and conventions used throughout the paper. In Section 3 we describe a preprocessing step and invariants that are maintained during execution of the algorithms described in subsequent sections. Sections 4 and 5 form the core of the paper and describe the bound consistency algorithms for increasingly complicated cases.

2 Preliminaries and Notation

2.1 Set and Graph Variables

The domain $D(x)$ of a *set variable* x is specified by two sets of elements: The set of elements that must belong to the set assigned to x (which we call the *lower bound* of the domain of x and denote by $\underline{D}(x)$) and the set of elements that may belong to this set (the *upper bound* of the domain of x, denoted $\overline{D}(x)$). The domain itself has a lattice structure corresponding to the partial order defined by set inclusion. In other words, for a set variable x with domain $D(x) = [\underline{D}(x), \overline{D}(x)]$, the value $v(x)$ that is assigned to x in any solution must be a set such that $\underline{D}(x) \subseteq v(x) \subseteq \overline{D}(x)$.

A *graph variable* can be modelled as two set variables V and E with an inherent constraint specifying that $E \subseteq V \times V$. Alternatively, the domain $D(G)$ of a graph variable G can be specified by two graphs: A lower bound graph $\underline{D}(G)$ and an upper bound graph $\overline{D}(G)$, such that the domain is the set of all subgraphs of the upper bound which are supergraphs of the lower bound. We will assume the latter because it is more convenient when describing filtering algorithms.

2.2 Assumptions and Conventions

The constraint considered in this paper is defined on graph variables as well as scalar variables. The latter represent weights and are assigned numbers. We will assume that the domain of each scalar variable is an interval, represented by its endpoints. The number of entries in the vector W is equal to the number of edges in G. We will abstractly refer to $W[e]$ as the entry that contains the weight of edge e, ignoring implementation details. We also allow those intervals to be filtered to an empty interval when the corresponding edge does not belong to any solution, ignoring the fact that some constraint systems do not allow empty domains.

When the cardinality of the domain of a variable is exactly 1, we say that the variable is *fixed*. We will denote fixed variables by lowercase letters and non-fixed variables by capital letters.

2.3 Bound Consistency

For a constraint that is defined on variables whose domains are intervals, specified by their endpoints, computing bound consistency amounts to shrinking the variable domains as much as possible without losing any solutions. If the domain is an interval from a total order (which is the case for the entries of W), the bound consistent domain is specified simply by the smallest and the largest value that the variable can assume in a solution.

On the other hand, when the domain is an interval of a partial order (which is the case with a graph variable), the lower bound is the intersection of all values that the variable can assume and the upper bound is the union of all such values. Note that the domain endpoints might not be values that the variable can assume in a solution. Since a bound consistency algorithm may only remove elements from a variable domain but never add new ones, filtering the domain of a graph variable to bound consistency amounts identifying which of the nodes and edges in its upper bound graph must belong to the graph in all solutions (and placing them in the lower bound graph) as well as which nodes and edges may not belong to the graph in any solution (and removing them from the upper bound graph).

3 Side Constraints and Invariants

Throughout the paper, we will assume that the following filtering tasks have been performed in a preprocessing step: First, the algorithm applies the bound consistency algorithms described in [4,5] for the constraints $\mathit{Nodes}(G) = \mathit{Nodes}(T)$, $\mathit{Subgraph}(T, G)$ (which specifies that T is a subgraph of G) and $\mathit{Tree}(T)$ (which specifies that T is connected and acyclic). Filtering for the $Tree(T)$ constraint removes from $\overline{D}(T)$ arcs whose endnodes belong to the same connected component of $\underline{D}(T)$, and enforces that T is connected: If there are two nodes in $\underline{D}(T)$ that do not belong to the same connected component of $\overline{D}(T)$ then the constraint has no solution. Otherwise, any bridge or cut-node in $\overline{D}(T)$ whose removal disconnects two nodes from $\underline{D}(T)$ is placed in $\underline{D}(T)$.

Note that the conjunction of $\mathit{Subgraph}(T, G)$ and $\mathit{Tree}(T)$ enforces $\mathit{Connected}(G)$ and that we may assume that bound consistency for $\mathit{Subgraph}(T, G)$ is maintained dynamically at no asymptotic cost, as described in [5].

Finally, since $\underline{D}(T)$ is contained in the MST in any solution, we reduce the problem to the case in which $\underline{D}(T)$ is initially empty, as follows. We contract all edges of $\underline{D}(T)$ in g and obtain the graph g'. For any MST t' of g', the edge-set $t' \cup \underline{D}(T)$ is a minimum-weight spanning tree of g that contains $\underline{D}(T)$.

4 Fixed Edge Weights

In this section we assume that the edge weights are fixed. That is, the domain of W contains exactly one vector. We separately handle each of the three subcases where at least one other variable is not fixed.

4.1 Non-fixed Graph and Fixed Tree

We first tackle the simple case $MST(G, t, w)$, where $dom(G) = [\underline{D}(G), \overline{D}(G)]$ is not fixed while the minimum spanning tree and the edge weights are.

After applying the filtering described in Section 3 for $Subgraph(T, G)$ and node-set equality, it remains to enforce that t is an MST of the value assigned to G. This means that we need to remove from $\overline{D}(G)$ any edge (u, v) which is not in t and which is lighter than all edges on the path p in t between u and v; by the cycle property, the heaviest edge on the cycle $p \cup \{(u, v)\}$ (which is in t), cannot belong to any MST, so the cycle must not be in G. The only way to exclude the cycle is to remove (u, v) from $\overline{D}(G)$. This can be done as follows in linear time: For each edge $e = (u, v)$ in $\overline{D}(G) \setminus t$, compute the maximum edge weight $w*$ of an edge on the path from u to v in t using King's algorithm [11], which receives a weighted tree and, in linear time, constructs a data structure that supports constant-time queries of the form "which is the heaviest edge on the tree path between u and v?" If $w(e) < w*$, e may not belong to G. If it is in $\underline{D}(G)$ the constraint has no solution. Otherwise, remove it from $\overline{D}(G)$.

4.2 Fixed Graph and Non-fixed Tree

We turn to the case $MST(g, T, w)$ where the variables G and W of the constraint are fixed. The tree T is constrained to be a minimum spanning tree of the given graph g and the bound-consistency problem amounts to finding the union and the intersection of all MSTs of g.

Analysis of g to filter $D(T)$. We now describe a variant of Kruskal's algorithm [12] that constructs an MST of g while partitioning its edge-set into the sets $Mandatory(g)$, $Possible(g)$ and $Forbidden(g)$, defined as follows.

Definition 1. *Let g be a connected graph. The sets $Mandatory(g)$, $Possible(g)$ and $Forbidden(g)$ contain, respectively, the edges that belong to all, some or none of the MSTs of g.*

For an unconnected graph g whose maximal connected components are $g_1, \ldots, g_k$, we extend this definition to be the union of the respective set for each maximal connected component of g. Formally,

$$\begin{aligned} Mandatory(g) &= \cup_{i=1}^{k} Mandatory(g_i), \\ Possible(g) &= \cup_{i=1}^{k} Possible(g_i), \\ Forbidden(g) &= \cup_{i=1}^{k} Forbidden(g_i), \end{aligned}$$

As in Kruskal's original version, the algorithm begins with a set of n singleton nodes and grows a forest by repeatedly inserting a minimum weight edge that does not create a cycle. The difference is that instead of considering one edge at a time, in each iteration we extract from the queue all edges of minimal weight, determine which of them are mandatory, possible or forbidden, and only then attempt to insert them into the forest. Let t_k be the forest constructed by using edges of weight less than k and let E_k be the set of edges of weight k. Let $(u,v) \in E_k$ and let $C(u)$ and $C(v)$ be the connected components in t_k of u and v, respectively. If $C(u) = C(v)$, then by the cycle property (u,v) does not belong to any MST of g (i.e., $(u,v) \in Forbidden(g)$). If $C(u) \neq C(v)$ and (u,v) is a bridge in $t_k \cup E_k$, then by the cut property (u,v) belongs to all MSTs of g (i.e., $(u,v) \in Mandatory(g)$).

The running time of this algorithm is $O(Sort(m) + m\alpha(m,n))$ where $Sort(m)$ is the time required to sort the edges by weight and α is the inverse-Ackerman upper bound of the union-find data structure [16] that represents the trees of the forest. When a batch of edges is extracted from the queue we need to perform a bridge computation in the graph composed of these edges to distinguish between possible and mandatory edges. Bridge detection takes time which is linear in the number of edges [15] and each edge of g participates in one bridge computation.

Filtering the Domain of T. We are now ready to use the results of the analysis of g to filter the domain of T. This entails the following steps: (1) For each mandatory edge $e \in Mandatory(g)$, if $e \notin \overline{D}(T)$ then there is no solution. Otherwise, place e in $\underline{D}(T)$. (2) For each forbidden edge $e \in Forbidden(g)$, if $e \in \underline{D}(T)$ then there is no solution. Otherwise, remove e from $\overline{D}(T)$.

Since $Mandatory(g)$ and $Forbidden(g)$ are disjoint, the two steps have no effect on each other, so they may be applied in any order, and it suffices to apply each of them only once. But could we achieve more filtering by repeating the whole algorithm again, from the preprocessing step through the analysis of g to the filtering steps? We will now show that we cannot.

Let e be an edge that was placed in $\underline{D}(T)$ in the first filtering step. Then $e \in Mandatory(g)$, which means that it belongs to all MSTs of g. Let t_1 and t_2 be the two trees that e merges together when it is inserted into the forest by our variant of Kruskal's algorithm on g. Then the edges that were extracted from the queue before e do not contain an edge between t_1 and t_2, because in that case e would have been placed in either $Possible(g)$ or $Forbidden(g)$. This means that if e is in $\underline{D}(T)$ from the start, all edges lighter than e would be classified as before. Clearly, the edges heavier than e see the same partition of the graph into trees whether e is in $\underline{D}(T)$ or not. Since e is mandatory, the edges that have the same weight as e do not belong to a path between e's endpoints that uses only edges with weight at most equal to that of e. Hence, placing e in $\underline{D}(T)$ cannot change their classification.

Now, let e be an edge that was removed from $\overline{D}(T)$ in the second filtering step. Then $e \in Forbidden(g)$, which means that it does not belong to any MST of g. Then its removal from $\overline{D}(T)$ does not have any effect on the classification of other edges as $Forbidden$, $Possible$ or $Mandatory$.

We do not need to apply the filtering steps of Section 3. The nodes of T are fixed so we cannot detect new cut nodes. Clearly an impossible edge does not disconnect $\overline{D}(T)$ (otherwise it would be mandatory by definition). We show that the pruning of $\overline{D}(T)$ cannot creates new bridges then these bridges are already in $Mandatory(g)$. We consider an impossible edge e is removed and creates a bridge in $\overline{D}(T)$. As it is impossible, its endpoints are connected by a path composed of edges lighter than e. Then when the bridge e' of weight k was processed, the edge e was not in the graph g_k, and e' was a bridge in g_k. Hence it was classified as mandatory.

In conclusion, if we apply the algorithm again, the analysis of g would classify all edges in the same way as before. In other words, applying the algorithm again will not result in more filtering.

4.3 Non-fixed Graph and Tree

We now turn to the case $MST(G, T, w)$, in which both the graph and the tree are not fixed (but the edge weights are). Recall that we begin by applying the preprocessing step described in Section 3.

Analyzing $D(G)$ to filter $D(T)$. The main complication compared to the fixed-graph case is in the analysis of the set of graphs described by $D(G)$ in order to filter $D(T)$. We extend the definition of the sets $Mandatory$, $Possible$ and $Forbidden$ for a set of graphs as follows:

Definition 2. *For a set S of graphs, the set $Mandatory(S)$ contains the edges that belong to every MST of any connected graph in S, the set $Forbidden(S)$ contains the edges that do not belong to any MST of a connected graph in S and the set $Possible(S)$ contains all other edges in the union of the graphs in S.*

We need to identify the sets $Mandatory(D(G))$ and $Forbidden(D(G))$. We will show that it suffices to analyze the two bounds of the graph domain, namely the graphs $\underline{D}(G)$ and $\overline{D}(G)$.

Lemma 1 (Downgrade lemma). *The addition of an edge to a graph can only downgrade the state of the other edges of this graph. Here, downgrading means staying the same or going from "mandatory" to "possible" to "forbidden". Formally: Let $g^+ = g \cup \{e = (u, v)\}$ where $e \notin g$ and let $k = w(e)$. Then:*

$$\forall a \in g : \big(a \in Mandatory(g^+) \Rightarrow a \in Mandatory(g)\big) \wedge$$
$$\big(a \in Possible(g^+) \Rightarrow a \in Mandatory(g) \cup Possible(g)\big)$$

Proof. We compare the classification of the edges obtained by running the algorithm of Section 4.2 twice in parallel, one copy called A running on the graph g and one copy called A^+ running on g^+. Clearly, as long as the edge e is not popped from the queue, the edges are classified in the same way in both graphs.

If e is an impossible edge of g^+, then the edges popped after it will still be classified in the same way in both graphs. Otherwise, it can affect the fate of the edges that are popped at the same time or later:

Case 1. If u and v belong to trees that are also merged by another edge e' with weight equal to that of e, then both e and e' are classified in g^+ as possible, while in g the edge e' was classified as either possible or mandatory (depending on whether it was the only path connecting these trees in the batch). After this, the partition of the nodes of the graphs into trees is the same for g and g^+, and the algorithms classify the remaining edges in the same way.

Case 2. Otherwise, A leaves u and v in different trees while A^+ merges the two trees. At some point, both algorithms see a batch whose insertion connects u and v in g. These edges will be classified by A as either possible (if there is more than one) or mandatory (if there is only one). On the other hand, A^+ will classify them as impossible because their endpoints already belong to the same tree. In all other steps, both algorithms classify the edges in the same way. □

Note that the addition of a node of degree 1 and its incident edge to a graph does not change the status of the other edges in the graph. This edge just becomes mandatory.

We now show how to identify the sets $Forbidden(D(G))$ and $Mandatory(D(G))$. The set $Possible(D(G))$ consists of the remaining edges in $\overline{D}(G)$. Recall that the set $Forbidden(\underline{D}(G))$ is the union of the forbidden edges of each maximal connected component of $\underline{D}(G)$.

Theorem 1. *The set $Forbidden(D(G))$ of edges that do not belong to any MST of a connected graph in $D(G)$ is*

$$Forbidden(D(G)) = Forbidden(\underline{D}(G)).$$

Proof. A direct consequence of the downgrade lemma. □

We now turn to computing the $Mandatory$ set. The following lemma states that the mandatory edges belong to the mandatory set of $\underline{D}(G)$. Note that this does not follow from the definition of $Mandatory(G)$, because if $\underline{D}(G)$ is not connected, it does not belong to $D(G)$.

Lemma 2. *The set of edges that belong to all MSTs of all connected graphs in $D(G)$ is contained in the set of mandatory edges for $\underline{D}(G)$, i.e.,*

$$Mandatory(D(G)) \subseteq Mandatory(\underline{D}(G)).$$

Proof. If $\underline{D}(G)$ is empty, $Mandatory(D(G)) = \emptyset$. Otherwise, there are two cases: If $\underline{D}(G)$ is a connected graph, then it belongs to $D(G)$ and the lemma holds by definition. Otherwise, we may assume that $\overline{D}(G)$ is connected. This follows from the pruning rules of the connected constraint in the preprocessing step: Otherwise either the constraint has no solution or $\underline{D}(G)$ is empty or we can remove all but one connected component of $\overline{D}(G)$. Let $\overline{D}(G)'$ be the graph obtained from $\overline{D}(G)$ by contracting every connected component of $\underline{D}(G)$.

Let g be a minimal connected graph in $D(G)$, i.e., a graph in $D(G)$ such that the removal of any node or edge from g results in a graph which is either not connected or not in $D(G)$. Then g consists of the union of $\underline{D}(G)$ with a set t' of additional nodes and edges in $\overline{D}(G) \setminus \underline{D}(G)$. The set of mandatory edges of g, then, is the union of $Mandatory(\underline{D}(G))$ and the mandatory edges in t'.

Since we assume that the preprocessing step was performed, we know that every edge in $\overline{D}(G) \setminus \underline{D}(G)$ is excluded from at least one connected graph in $D(G)$: Otherwise, it is a bridge in $\overline{D}(G)$ that connects two mandatory nodes and was not included into $\underline{D}(G)$ in the preprocessing step, a contradiction. We get that the intersection of the mandatory sets over all minimal connected graphs in $D(G)$ is equal to $Mandatory(\underline{D}(G))$.

Since every connected graph in $D(G)$ is a supergraph of at least one minimal connected graph of $D(G)$, we get by the downgrade lemma that the mandatory set for any graph is contained in the mandatory set for some minimal graph. This concludes the proof. □

Theorem 2. *The set $Mandatory(D(G))$ of edges that belong to all MSTs of graphs in $D(G)$ is:*

$$Mandatory(D(G)) = Mandatory(\overline{D}(G)) \cap Mandatory(\underline{D}(G))$$

Proof. If an edge is present in all MSTs of all connected graphs in $D(G)$ then it is present in all MSTs of $\overline{D}(G)$ and all MSTs of the minimal connected graphs of $D(G)$. Hence, by lemma 2, $Mandatory(\underline{D}(G)) \cap Mandatory(\overline{D}(G)) \supseteq Mandatory(D(G))$. Assume that there exists an edge e in $Mandatory(\underline{D}(G)) \cap Mandatory(\overline{D}(G))$ which is not in $Mandatory(D(G))$. Then there is a graph $g \in D(G)$ such that e is not in $Mandatory(g)$. Since $g \subset \overline{D}(G)$, we can obtain $\overline{D}(G)$ by a series of edge insertions. One of these insertions turned e from a non-mandatory edge into a mandatory one, in contradiction to the downgrade lemma. □

Example 1. Assume that the domain of G is the graph shown in Figure 1, where the solid edges are in $\underline{D}(G)$ and the dashed edge is in $\overline{D}(G) \setminus \underline{D}(G)$. Then $Mandatory(\overline{D}(G))$ contains the edges weighted 1 and 2, while the edge of weight 3 is forbidden. On the other hand, $Mandatory(\underline{D}(G))$ contains the edges weighted 2 and 3, because the edge of weight 1 is not in $\underline{D}(G)$. Hence, only the edge of weight 2 is mandatory in all graphs of $D(G)$.

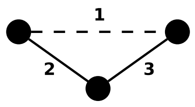


Fig. 1. The domain of G in Example 1

Filtering the Domains of G and T. Using the results of the previous sections, we derive a simple algorithm to filter the domains of T and G to bound consistency.

As before, we begin by applying the preprocessing step of Section 3. We then proceed as follows.

Step 1: We filter $D(T)$ according to $Mandatory(D(G))$ and $Forbidden(D(G))$ as in Section 4.2. By Theorems 1 and 2, we have these two sets if we know $Forbidden(\underline{D}(G))$, $Mandatory(\underline{D}(G))$ and $Mandatory(\overline{D}(G))$. The latter can be computed by applying the algorithm described in Section 4.2 to both bounds of $D(G)$. Once these sets have been computed, we use them as in Section 4.2 to filter $D(T)$.

Step 2: To filter $D(G)$, we need to identify edges that cannot be in G because otherwise T would not be the minimum spanning tree of G. An edge (u, v) has this property iff on every path P in $\overline{D}(T)$ between u and v there is an edge which is heavier than (u, v).

Example 2. To illustrate this condition, assume that the domain of T is as described in Figure 2 and that $\overline{D}(G)$ contains the edge (u, v) with weight 2. Although u and v are not connected in $\underline{D}(T)$, any path from u to v in a tree in $D(T)$ contains an edge which is heavier than (u, v), so (u, v) must not be in G. On the other hand, if the weight of the edge (x, y) is 1, then (u, v) can be in G if $(x, y) \in T$.

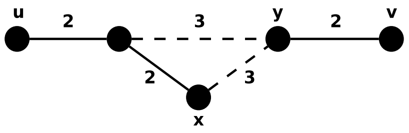


Fig. 2. The domain of T in Example 2. Edges of $\underline{D}(T)$ and solid and those of $\overline{D}(T) \setminus \underline{D}(T)$ are dashed.

To find these edges, we apply a modified version of the algorithm of Section 4.2 to $\overline{D}(G)$: We reverse the contraction of the edges of $\underline{D}(T)$ and create a sorted list of all edges of $\overline{D}(G)$, including those of $\underline{D}(T)$. As before, we begin with a graph H that contains the nodes of $\overline{D}(G)$ as singletons and at each step we extract from the queue the batch B of all minimum-weight edges. We first insert the edges of $B \cap \overline{D}(T)$ into H and contract each of them by merging the connected components that its endpoints belong to. Then, we remove from $\overline{D}(G)$ every edge in $B \setminus \overline{D}(T)$ that connects two different connected components of H; If any of these edges are in G, then at least one of them must belong to the MST of G. But they cannot be in T, so we must make sure that they are not in G either.

After applying Step 2, we need to apply Step 1 again. However, we show that after doing so we have reached a fixpoint. We first show that the second application of the first step does not change $\overline{D}(T)$. Since the second step depends only on $\overline{D}(T)$, we are done if we use the $\overline{D}(G)$ computed by the second step to update $\underline{D}(T)$.

Consider the impact of the removal of a non-tree edge e (i.e., an edge which is not in $\overline{D}(T)$) from $\overline{D}(G)$ during Step 1. The set $Forbidden$ is not affected because it depends only on $\underline{D}(G)$. Assume that the removal of e causes the insertion of an edge e' into $\underline{D}(G)$. Then e' was a bridge in $\overline{D}(G) \setminus \{e\}$. But since $e \notin \overline{D}(T)$, e' is a bridge in $\overline{D}(T)$ so it already was in $\underline{D}(T)$, and hence also in $\underline{D}(G)$, a contradiction. This proves that we have reached a fixed-point.

5 Handling Non-fixed Weights

We now turn to the case where the edge weights are not fixed, i.e., the domain of each entry in W is an interval of numbers, specified by its endpoints. Once again, we begin with simple cases where some of the variables are fixed and gradually build up to the most general case.

5.1 Fixed Graph and Tree

When the graph and tree are fixed ($MST(g, t, W)$), the filtering task is to compute the minimum and maximum weight that each edge can have such that t is an MST of g. [1] A non-tree edge (u, v) must not be lighter than any edge on the path in t between u and v. We can apply King's algorithm to t, while assuming that each tree edge has the minimum possible weight. Then, if the data structure returns the edge e' for the query $e = (u, v)$, we filter the domain of the weight of e, denoted $W[e]$, by setting $D(W[e]) \leftarrow D(W[e]) \cap [\underline{D}(W[e']), \infty]$.

For a tree edge e, let $t_1(e)$ and $t_2(e)$ be the two trees obtained by removing e from t. Then e must not be heavier than any other edge in g that connects a node from $t_1(e)$ and a node from $t_2(e)$. Let $r(e)$ be the minimum weight edge between $t_1(e)$ and $t_2(e)$ in the graph $g \setminus e$. Assuming that every non-tree edge has the maximal possible weight, we can find the $r(e)$'s for all tree edges within a total of $O(m\alpha(m, n))$ time [6,17]. Then, for every $e \in t$ we set $D(W[e]) \leftarrow D(W[e]) \cap [-\infty, \overline{D}(W[r(e)])]$.

Now, if there is an entry $W[e]$ in the weights vector with $D(W[e]) = \emptyset$, the constraint has no solution.

5.2 Fixed Tree and Non-fixed Graph

When the graph is not fixed but the tree is, the node-set of the graph is determined by the tree. After filtering $D(G)$ to equate the node-sets and to contain all edges of t, we have that the endpoints of all non-tree edges, i.e., edges of $\overline{D}(G) \setminus t$, belong to t.

We apply the filtering step of the previous section to the weights of the non-tree edges. If this results in $D(W[e]) = \emptyset$ for some edge e, there are now two options: If $e \in \overline{D}(G) \setminus \underline{D}(G)$ we remove e from $\overline{D}(G)$, and if $e \in \underline{D}(G)$ then the constraint has no solution.

Next, we filter the weights of tree edges by applying the algorithm of the previous section on $\underline{D}(G)$. That is, for each tree edge e we find the weight of the lightest edge $r(e)$ in $\underline{D}(G)$ that connects $t_1(e)$ and $t_2(e)$, and shrink $D(W[e])$ as before. To see why it suffices to consider $\underline{D}(G)$, note that an edge e' in $\overline{D}(G) \setminus \underline{D}(G)$ is excluded from at least one graph in $D(G)$, and in this graph, of course, e may be heavier than e'.

5.3 Fixed Graph and Non-fixed Tree

In Section 4.2, we handled the same problem with fixed weights by a variant of Kruskal's algorithm that required sorting the edges by weight. Since one weight interval can now overlap another, there is no longer a total order of the edge weights. We will show how to adapt the Kruskal-based algorithm to this case.

Phase 1: First, the algorithm considers edges in g. Instead of a list of edge-weights, we create a list of the endpoints of these edges' domains. We sort them in non-decreasing

[1] This problem is tightly related to the *MST sensitivity analysis problem*, where we are given a graph with fixed edge weights and its MST and need to determine, for each edge, the amount by which its weight can be perturbed without changing the property that the tree is an MST of the graph.

order, breaking ties in favor of lower bounds. Now, $\underline{D}(W[e])$ or $\overline{D}(W[e])$ is between $\underline{D}(W[e'])$ and $\overline{D}(W[e'])$ in the list if and only if $D(W[e]) \cap D(W[e']) \neq \emptyset$.

We then sweep over this list and examine each domain endpoint in turn. We say that an edge e is *unreached* before $\underline{D}(W[e])$ was processed, *open* if $\underline{D}(W[e])$ was already processed but $\overline{D}(W[e])$ was not, and *closed* after $\overline{D}(W[e])$ was processed. We maintain a graph H which is initially a set of n singleton nodes. In addition, we maintain a union-find data structure UF that represents the connected components of the subgraph H' of H that contains only closed edges. Initially, the union-find data structure also has n singletons. During the sweep, when processing $\underline{D}(W[e])$ where $e = (u, v)$, we mark e as *open*. If $Find(UF, u) = Find(UF, v)$, we place e in the *forbidden set*. Otherwise, we insert it into H. When processing $\overline{D}(W[e])$ where $e = (u, v)$, we mark e as *closed*. If e is a bridge in H, we place it in the *mandatory set* Finally, we perform $Union(UF, u, v)$.

After the sweep, all edges of the mandatory set are included in $\underline{D}(T)$ and all of the impossible edges are removed from $\overline{D}(T)$. Naturally, if this violates $\underline{D}(T) \subseteq \overline{D}(T)$, the constraint is inconsistent.

Phase 2: Next, the algorithm filters the weights of all edges. For a non-tree edge e, i.e., an edge $e \in g \setminus \overline{D}(T)$, the weight must be high enough so that e does not belong to any MST of g. In other words, the weight of e must be higher than the maximum weight of an edge on the tree path connecting its endpoints. In Section 4.1 we mentioned that King's algorithm can find the desired threshold when the tree is fixed. But how do we find the MST in $D(T)$ that minimizes the weight of the heaviest edge on the path between u and v? Clearly, the desired weight is the lower bound of the domain of this edge's weight. The following lemma implies that it suffices to find any MST in $D(T)$ (while assuming that each edge has the minimum possible weight), and apply King's algorithm to this MST.

Lemma 3. *Let g be a graph with a fixed weight $w(e)$ for each edge e and let t_1 and t_2 be two MSTs of g. Let u and v be two nodes in g, let p_1 be the path between u and v in t_1 and let p_2 be the path between u and v in t_2. Then*

$$\max_{e \in p_1} w(e) = \max_{e \in p_2} w(e)$$

Proof. Consider the symmetric difference $p_1 \oplus p_2$ of the two paths. It consists of a collection of simple cycles, each of which consists of a subpath of p_1 and a subpath of p_2. Let c be one of these cycles. By the cycle property, any MST of g excludes a maximum weight edge from c (see Figure 3). Hence, there are at least two maximum edge weights c, one in $c \cap p_1$ and one in $c \cap p_2$. Since this is true for every cycle, we get that the maximum weights on each of the paths must be equal. □

For an edge $e \in \underline{D}(T)$, i.e., an edge which belongs to all MSTs, the weight must not be so high that there is a cycle in g on which this is the heaviest edge. In other words, we need to find an MST t of g that contains $\underline{D}(T)$ and which maximizes the minimum weight of a non-tree edge r_e that together with t forms a cycle that contains e. In Section 5.1 we mentioned that the desired threshold for all tree edges can be found

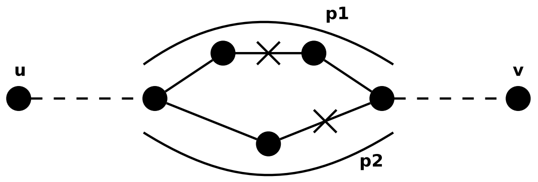


Fig. 3. Illustration of the proof of lemma 3. A cycle and its p_1 and p_2 subpaths, the maximum weight edges excluded from each MST.

in $O(m\alpha(m,n))$ time when the tree is fixed. Once again, we show that it suffices to apply the same algorithm to one of the MSTs in $D(T)$. Clearly, the desired weight is the upper bound of the domain of an edge weight. Furthermore, reducing the weight of any edge in g can only decrease the value of $w(r_e)$. The following lemma implies that it suffices to find any MST of g that contains $\underline{D}(T)$ (while assuming that each edge has the maximum possible weight), and use this MST to compute the thresholds for all tree edges.

Lemma 4. *Let g be a graph with a fixed weight $w(e)$ for each edge e and let t_1 and t_2 be two MSTs of g. Let e be an edge in $t_1 \cap t_2$, let r_1 be the minimum weight edge that together with t_1 closes a cycle that contains e and let r_2 be the minimum weight edge that together with t_2 closes a cycle that contains e. Then*

$$w(r_1) = w(r_2)$$

Proof. It is known (see, e.g., Lemma 3 in [6]) that each of $t_1 \cup \{r_1\} \setminus \{e\}$ and $t_2 \cup \{r_2\} \setminus \{e\}$ is an MST of $g \setminus \{e\}$. Since all MSTs of a graph have equal weight, $w(r_1) = w(r_2)$. □

The time complexity of the algorithm described in this section is $O(Sort(m))$ to sort the endpoints of the domains of the edge weights and $O(m\alpha(m,n))$ for the modified Kruskal algorithm, incremental connectivity [16] and bridge computation [18], two MST computations and filtering of the edge weights.

5.4 General Case: Non-fixed Graph and Non-fixed Tree

We now turn to the most general case, in which all variables are not fixed. In the case of the $WBST$ constraint, we were able to find efficient bound consistency algorithms for special cases, but the most general case is NP-hard. In contrast, we will show that the MST constraint is not NP-hard in its most general form. The naïve bound consistency algorithm that we sketch in this section has a running time of $O(mn(m+\log n))$, which cannot be considered practical. We leave it as an open problem to find a more efficient method to approach the general case.

As before, we apply the filtering steps that follow from equality of the node-sets, inclusion of T in G and the connectedness of G and T. The main complication compared to the cases considered in the previous sections is in filtering the domains of the edge weights, because now the node-sets of the graph and the tree are not fixed. Again, we need to filter the lower bound weight of non-tree edges and the upper bound weight of tree edges.

Filtering the weights of non-tree edges. An edge $e = (u, v)$ in $\overline{D}(G) \setminus \overline{D}(T)$ cannot belong to the tree and therefore it must not be lighter than the maximum weight edge on the tree path from u to v. Let t be a tree and let $t(u, v)$ be the maximum weight edge on the path in t between u and v. We need to determine the minimum possible value of $t(u, v)$ over all trees in $D(T)$. Clearly, this weight would be a lower bound of the domain of some edge weight.

We set edge weights of all edge in $\overline{D}(T)$ to their lower bounds and contract the edges of $\underline{D}(T)$. In the remaining graph, we need to find a simple path from u to v that minimizes the maximum weight of an edge along it. This can be done in $O(m + n \log n)$ time by a dynamic programming approach that uses a variation of Djikstra's shortest-paths algorithm [3], where the sum of edge weights is replaced by a maximum computation. Since this computation needs to be repeated m times, once for every non-tree edge, the total time is $O(m(m + n \log n))$.

Filtering the weights of tree edges. A tree edge e must not be heavier than any non-tree edge that connects two nodes u and v such that e is on the tree path between u and v. To find the maximum possible weight of an edge $(u, v) \in \underline{D}(T)$, we will show how to check, for each edge $(x, y) \in \overline{D}(G) \setminus \underline{D}(T)$, whether there is a tree $t \in D(T)$ that contains (u, v), such that (x, y) is the minimum weight edge connecting the two components of $t \setminus \{(u, v)\}$.

Let $E' = (\underline{D}(T) \setminus \{(u, v)\}) \cup \{e | e \in \underline{D}(G) \cap \overline{D}(T) \wedge w(e) < w(x, y)\}$. If t exists, then each edge of E' is either in t or connects nodes that belong to the same connected component of $t \setminus \{(u, v)\}$. So we compute connected components of $G' = (Nodes(\overline{D}(G)), E')$. If x and y are in the same component, there is no such t. Otherwise, contract each connected component and merge the component of u with the component of v. Let G'' be the resulting graph. For a node $w \in G'$, we will refer to the node of G'' that represents the connected component of w by $CC(w)$. We will say that a node in G'' is mandatory if it represents a component in G' that contains at least one node from $\underline{D}(T)$.

Insert into G'' all the edges of $\overline{D}(T)$ which are not heavier than (x, y). It remains to determine whether we can make a tree that spans $CC(x)$, $CC(y)$, $CC(u) = CC(v)$ and all the mandatory nodes of G'', such that $CC(u)$ is on the path from $CC(x)$ to $CC(y)$. To do this, root G'' at the $CC(u)$ and find its dominators (in linear time) [9]. If $CC(x)$ and $CC(y)$ have a common dominator, such a tree does not exist. Otherwise, find two disjoint paths, one from $CC(u)$ to $CC(x)$ and one from $CC(u)$ to $CC(y)$. Then add edges to the tree to make it span all mandatory nodes.

This test takes linear time, and needs to be repeated at most m times for every edge in $\underline{D}(T)$, i.e., $O(mn)$ times. The total running time is therefore $O(m^2 n)$.

6 Conclusion

We have shown that it is possible to compute bound consistency for the MST constraint in polynomial time. For the special cases in which at least one of the variables is fixed, we found linear or almost-linear time algorithms. For the most general case, our upper bound is cubic and relies on a brute-force algorithm. It remains open whether the techniques we used for the simpler cases can be generalized to an efficient solution for MST in its most general form.

References

1. I. D. Aron and P. Van Hentenryck. A constraint satisfaction approach to the robust spanning tree problem with interval data. In *UAI*, pages 18–25, 2002.
2. N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *CP-AI-OR 2005*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.
3. E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. 1959.
4. G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In *CP 2005*, volume 3709 of *LNCS*, pages 211–225. Springer-Verlag, 2005.
5. G. Dooms and I. Katriel. Graph constraints in constraint programming: Weighted spanning trees. Research Report 2006-1, INGI, Belgium, 2006.
6. D. Eppstein. Finding the k smallest spanning trees. In *SWAT '90*, pages 38–47, London, UK, 1990. Springer-Verlag.
7. D. Eppstein. Setting parameters by example. *SIAM J. Comput.*, 32(3):643–653, 2003.
8. N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *FOCS 1996*, page 302, Washington, DC, USA, 1996. IEEE Computer Society.
9. L. Georgiadis and R. E. Tarjan. Finding dominators revisited: extended abstract. In *SODA 2004*, pages 869–878. SIAM, 2004.
10. R. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*. Volume 53 of Annals of Discrete Mathematics., North Holland, Amsterdam, 1992.
11. V. King. A simpler minimum spanning tree verification algorithm. In *WADS*, volume 955 of *LNCS*, pages 440–448. Springer, 1995.
12. J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. American Mathematical Society*, 7:48–50, 1956.
13. I. Katriel N. Beldiceanu and X. Lorca. Undirected forest constraints. In *CP-AI-OR 2006*, volume 3990 of *LNCS*, pages 29–43. Springer-Verlag, 2006.
14. M. Sellmann. Cost-based filtering for shorter path constraints. In F. Rossi, editor, *CP 2003*, volume 2833 of *LNCS*, pages 694–708. Springer, 2003.
15. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
16. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
17. R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.
18. J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

Impact of Censored Sampling on the Performance of Restart Strategies

Matteo Gagliolo[1,2] and Jürgen Schmidhuber[1,3]

[1] IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland
[2] University of Lugano, Faculty of Informatics,
Via Buffi 13, 6904 Lugano, Switzerland
[3] TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany

Abstract. Algorithm selection, algorithm portfolios, and randomized restarts, can profit from a probabilistic model of algorithm run-time, to be estimated from data gathered by solving a set of experiments. Censored sampling offers a principled way of reducing this initial training time. We study the trade-off between training time and model precision by varying the censoring threshold, and analyzing the consequent impact on the performance of an optimal restart strategy, based on an estimated model of runtime distribution.

We present experiments with a SAT solver on a graph-coloring benchmark. Due to the "heavy-tailed" runtime distribution, a modest censoring can already reduce training time by a few orders of magnitudes. The nature of the optimization process underlying the restart strategy renders its performance surprisingly robust, also to more aggressive censoring.

1 Introduction

The interest in algorithm performance modeling is twofold. An accurate model can provide useful insights for analyzing algorithm behavior [1]; it can as well be used to automate selection [2], or, more generally, allocation of computational resources among different algorithms. In the context of computational performance analysis of solvers for constraint satisfaction, an interest has been recently growing around the existence, in some structured domains, of a second phase transition, in the under-constrained region [3], where, for some problems, the run-time distribution exhibits "heavy tails", i.e., is Pareto for both very large and very small values of time [4]. In this case, the probability mass of the algorithm's runtime distribution can effectively be shifted towards lower time values, by simply running multiple copies of the same algorithm in parallel (algorithm *portfolios* [5]), or by repeatedly restarting a single algorithm, each time with a different initialization or random seed [6], as this allows to avoid the "unlucky" long runs, and profit from the very short ones.

Both strategies can be rendered more efficient if a model of run-time distribution (RTD) is available for the algorithm/problem combination at hand.[1] In the context of

[1] In the following, for the sake of readability, we will often refer to the RTD of a problem instance, meaning the RTD of different runs of the randomized algorithm of interest on that instance; and the RTD of a problem set, meaning the RTD of different runs of the randomized algorithm of interest, each run on a different instance, uniform randomly picked without replacement from the set.

algorithm portfolios, models can be used to allocate resources to different competing algorithms, or to evaluate the optimal number of components for a homogeneous portfolio [7]. In the case of restarts, it has been shown that the knowledge of the RTD allows to determine an optimal strategy, based on a constant cutoff [6].

What makes these approaches problematic is the huge computation time required, not for training the model, but for gathering the training data itself, as one might need to solve a large number of problems, in order to obtain a reliable sample of run-time values. One way of countering this is offered by *censored sampling*, a technique commonly used for lifetime distribution estimation (see, e.g., [8]), which allows to bound the duration of each training experiment, and still exploit the information conveyed by runs that reach the censoring threshold.

This obviously has a cost, to be paid in terms of the precision of the obtained model. This cost can in principle be measured according to traditional statistical goodness-of-fit tests, but if the sole purpose of the model is to set up a portfolio, or a restart strategy, in order to gain on future performance, then the only quantity of practical interest is the loss in performance induced by the censored sampling.

The main objective of this work is to analyze this trade-off between training time and efficiency, in the case of optimal restarts. To this aim we present experiments with a randomized version [4] of a well known complete SAT solver [9], on a benchmark of graph coloring problems from SATLIB [10], on which heavy-tailed behavior can be observed.

In the following, after some additional references (Sect. 2), censored sampling (Sect. 3) and restart strategies (Sect. 4) are briefly introduced, followed by a description (Sect. 5) and discussion (Sect. 6) of experimental results.

2 Previous Work

As an extensive review of the literature on modeling run-time distribution is beyond the scope of this paper, we will limit to a few examples. The behavior of complete SAT solvers on solvable and unsolvable instances near phase transition have been shown to be approximable by Weibull and lognormal distributions respectively [11]. Heavy-tailed behavior is observed for backtracking search on structured underconstrained problems in [3,4], but also in many other problem domains, such as computer networks [12]. Interesting hypothesis on the mechanism behind it are explored in [1]. The performance of local search SAT solvers is analyzed in [13,14], and modeled in [15] using a mixture of exponential distributions.

The literature on algorithm portfolios [5,7], anytime algorithms [16,17], and restart strategies [18,6,19,20,12] provides many examples of the application of performance modeling to resource allocation. In [21] we presented a dynamic approach, in which a conditional model of performance is updated and progressively exploited during training. See also [22] for more references.

3 Censored Sampling

Consider a "Las-Vegas" algorithm [6] running k times on instances of a family of problems, on which we believe that the algorithm will display a similar behavior. Be

$\mathcal{T} = \{t_1, t_2, ..., t_k\}$ the set of outcomes of the experiments. In order to model the probability density function (pdf) of solution time t, we can choose a parametric function $g(t|\theta)$, with parameter θ, and then express the likelihood of $\mathcal{T}$ given θ, as

$$\mathfrak{L}(\mathcal{T}|\theta) = \prod_{i=1}^{k} \mathfrak{L}(t_i|\theta) = \prod_{i=1}^{k} g(t_i|\theta) \tag{1}$$

We can then search the value of θ that maximizes (1), or, in a Bayesian approach, introduce a prior $p(\theta)$ on the parameter, and maximize the posterior $p(\theta|\mathcal{T}) \propto \mathfrak{L}(\mathcal{T}|\theta)p(\theta)$.

Type I censored sampling (see, e.g., [8]) consists in stopping experimental runs that exceed a cutoff time t_c, and replacing the corresponding multiplicative term in (1) with

$$\mathfrak{L}_c(t_c|\theta) = \int_{t_c}^{\infty} g(\tau|\theta)d\tau = [1 - G(t_c|\theta)] \tag{2}$$

where $G(t|\theta) = \int_0^t g(\tau|\theta)d\tau$ is the conditional cumulative distribution function (CDF) corresponding to g. This allows to limit the computational cost of running the experiments, while exploiting the information carried by unsuccessful runs. In *Type II censored sampling*, k experiments are run in parallel, and stopped after a desired number u of uncensored samples is obtained. In this way the fraction of censored samples $c = (k - u)/k$ is set in advance, while the censoring threshold t_c for the remaining $k - u$ runs is determined by the outcome of the experiments, as the ending time of the u-th fastest experiment.

4 Optimal Restart Strategies

A restart strategy consists in executing a sequence of runs of a randomized algorithm, in order to solve a same problem instance, stopping each run k after a time $T(k)$ if no solution is found, and restarting the algorithm with a different random seed; it can be operationally defined by a function $T : \mathbb{N} \rightarrow \mathbb{R}^+$ producing the sequence of thresholds $T(k)$ employed. Luby et. al [6] proved that the optimal restart strategy is *uniform*, i. e., one in which a constant $T(k) = T$ is used to bound each run. They show that, in this case, the expected value of the total run-time t_T, i. e., the sum of runtimes of the successful run, and all previous unsuccessful runs, can be evaluated as

$$E(t_T) = \frac{T - \int_0^T F(\tau)d\tau}{F(T)} \tag{3}$$

where $F(t)$ is the cumulative distribution function (CDF) of the run-time t for an unbounded run of the algorithm, i. e., a function quantifying the probability that the problem is solved before time t. If such distribution is known, an optimal cutoff time T^* can be evaluated minimizing (3). Otherwise, they suggest a universal non-uniform restart strategy U, with cutoff sequence $\{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, ...\}$,[2] proving

[2] The sequence is composed of powers of 2: when 2^{j-1} is used twice, 2^j is the next. More precisely, $k = 1, 2, ...$, $T(k) := 2^{j-1}$ if $k = 2^j - 1$; $T(k) := T(k - 2^{j-1} + 1)$ if $2^{j-1} \leq k < 2^j - 1$.

that its computational performance t_U is, with high probability, within a logarithmic factor worse than the *expected* total run-time $E(t_{T^*})$ of the optimal strategy.

5 Experiments

A model $\hat{F}$ of the RTD on a set of problems can be obtained from censored and uncensored runtime samples; the performance of the corresponding sub-optimal uniform strategy $\hat{T}$, evaluated minimizing (3) with $\hat{F}$ in place of the real F, will depend on the precision of the estimated $\hat{F}$, which will in turn depend on the number of samples used to estimate it, and the amount of censoring. If we fix the number of samples, and vary the fraction of censored samples, we expect to observe a trade-off between the time spent running the training experiments, from whose outcomes $\hat{F}$ is estimated, and the performance of the corresponding $\hat{T}$. It is precisely this trade-off that we intend to analyze here.[3]

In order to do so, we set up a simple learning scheme. Given a set of problems, and a randomized solver s, we first randomly pick a subset of n problems. For each problem, we start r runs of the algorithm s, differing only for the random seed, for a total of $k = nr$ parallel runs. We control the duration of these "training" experiments with Type II censored sampling (see Sect 3), fixing a censoring fraction $c \in [0, 1)$ in advance: as the first $\lfloor(1 - c)k\rfloor$ runs terminate, we stop also the remaining $\lceil ck\rceil$ runs. The gathered runtime samples are then used to train a model $\hat{F}$ of the RTD, from which a uniform strategy $\hat{T}$ is evaluated, by minimizing (3) numerically. The performance of $\hat{T}$ is then tested on the remaining problems of the set. Varying c, we can measure the corresponding variations in training time, and in the performance of $\hat{T}$ on the test set.

The experiments were conducted using Satz-Rand [4], a version of Satz [9] in which random noise influences the choice of the branching variable. Satz is a modified version of the complete DPLL procedure, in which the choice of the variable on which to branch next follows an heuristic ordering, based on first and second level unit propagation. Satz-Rand differs in that, after the list is formed, the next variable to branch on is randomly picked among the top h fraction of the list. We present results with the heuristic starting from the most constrained variables, as suggested also in [9], and noise parameter set to 0.4.

The benchmark used (obtained from SATLIB [10]) consists of different sets of SAT-encoded "morphed" graph-coloring problems [23] (100 vertices, 400 edges, 5 colors, resulting in 500 variables and 3100 clauses when encoded as a CNF3 SAT problem): each graph is composed of the set of common edges among two random graphs, plus fractions $p \in [0, 1]$ and $1 - p$ of the remaining edges for each graph, chosen as to form regular ring lattices. Each of the 9 problem sets contains 100 instances, generated with a logarithmic grid of 9 different values for the parameter p, from 2^0 to 2^{-8}, to which we henceforth refer with labels $0, ..., 8$. This benchmark is particularly interesting, as the parameter p controls the amount of structure in the problem, and the heavy-tailed behavior of Satz-Rand varies accordingly on the different sets.

[3] Note that a given problem set might contain instances which follow sensibly different RTDs: in this case, the obtained model would capture the overall behavior of the algorithm on the set, and the corresponding restart strategy would be suboptimal for each single instance. We ignore this issue here, as we are only interested in comparing among different $\hat{F}$, and not with the real F.

For each problem set, we repeated the simple scheme described above, running $r = 20$ copies of Satz-Rand on each of $n = 50$ randomly picked training instances, and evaluating $\hat{T}$ on the remaining 50. The process was repeated for 10 different levels of the censored fraction c during training, from $c = 0$ to $c = 0.9$. For practical reasons, experiments with $c = 0$ were actually run with a very high censoring threshold[4] (10^6): only a few runs of Satz-Rand exceeded this value.

As for the model, we tried different alternatives, including Weibull,[5] lognormal, and the novel double and right-hand Pareto lognormal, introduced in [24] to model heavy tailed distributions. The double Pareto-lognormal distribution (DPLN) describes the distribution of the product of two independent random variables, one with lognormal distribution, one with Double Pareto distribution, whose pdf can be written as $\gamma t^{\beta-1}$ for $t < 1$ (left tail), and $\gamma t^{-\alpha-1}$ for $t > 1$, $\gamma = \alpha\beta/(\alpha + \beta)$. The advantage of DPLN is that it can adapt to both lognormal and heavy tailed distributions. We also tested various mixtures of two distributions, with pdf of the form $f(t|\theta) = wf_1(t|\theta_1) + (1 - w)f_2(t|\theta_2)$, with parameter $\theta = (w, \theta_1, \theta_2)$, $w \in [0, 1]$.

The models were trained by maximum likelihood, as described in Sect. 3. To compare with a non-parametric approach, we repeated the experiments using the Kaplan-Meier estimator [25], which can also account for censored samples. Among the parametric models, we obtained the best results with a mixture including one lognormal and either one double-Pareto lognormal, or only the heavy-tailed component, the Double Pareto distribution described above.

We present the results for this latter mixture, and for the Kaplan-Meier estimator. All quantities reported are upper 95% confidence bounds obtained from results of 10 runs. In Figg. 1, 2, right column, we present the tradeoff between training cost, labeled `train`, and restart performances on the test set, for the two models, respectively labeled `logndp` and `kme`, at different values of the censoring fraction c. We also plot the cost of the universal strategy, labeled U, on the test set (the performance on the training set is similar, as both are composed of 50 randomly picked problem instances). For the test time, we can appreciate some degradation of performance only for very heavy censoring ($c = 0.8, 0.9$), for which the advantage in training time is negligible anyway. Note that this does not mean that the accuracy of the model is unaffected: to highlight this apparent contradiction, we also plotted, in left column, the value of a χ^2 statistic of the parametric model `logndp`.[6] While for the uncensored estimate this is near or below the 95% acceptance threshold (indicated by the white bars in the plot), the value of the χ^2 test degrades rapidly even for low values of c.

[4] As we are also conducting experiments with parallel portfolios of heterogeneous algorithms, and thus we need a common measure of time, we modified the original code of Satz-Rand adding a counter, that is incremented at every loop in the code. The resulting time measure was consistent with the number of backtracks. All runtimes reported are in millions of loop cycles.

[5] It is interesting to see that the Weibull distribution, reported in [11] as having a good fit on satisfiable problems near the sat-unsat phase transition, has instead a very poor fit in this case.

[6] Measured as in [11], dividing the *uncensored* data into m bins (on a logarithmic scale), and comparing the number of samples o_i in each bin to the one e_i predicted by the fitted distribution, according to the formula $\chi^2 = \sum_i [(o_i - e_i)/e_i]^2$. A high value indicates a poor fit: the model passes the test with confidence α if χ^2 is lower than the $1 - \alpha$ quantile of the χ^2 distribution with $m - k$ degrees of freedom, k being the number of parameters in the model.

In Fig. 3 to 6 we display, on the top row, the average of the resulting censoring threshold t_c, for different censoring fractions c. This, along with the training cost, allows to appreciate the tail behavior of Satz-Rand on the different problem set. On problem set 1, most runtimes have a similar value, and the remaining few are very large. t_c is greatly reduced by a modest $c = 0.1$, but further censoring does not decrease it much: the same obviously applies to training cost. On problem set 8, runtime values are spread along two orders of magnitude. Increasing c has a more gradual impact on t_c, and on training cost. This situation varies gradually for intermediate problem sets. Problem 0 is less interesting, as all runs of Satz-Rand end in a similar time, and heavy tailed behavior is not observed. The resulting plots are similar to problem 1, without the heavy tail effects. In the second row, we display the CDF $\hat{F}$ estimated by `dplogn`, on a single run, for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with a better approximation of the real F of the set, the empirical Kaplan-Meier CDF evaluated on 200 uncensored runs for each problem (labeled `real`). To further investigate the tail behavior of Satz-Rand RTD, in the third row we display the log-log plot of its survival function $1 - F(x)$, where a Pareto tail ($\propto t^{-\alpha}$) is displayed as a straight line. In both cases, one can visually appreciate the degradation of the model, induced by censored sampling, especially for values of t larger than t_c, which are not seen by the model. So why the performance of the restart strategy is not affected? The fourth row of Figg. 3 to 6 seems to suggest an answer. It plots the expected cost (3) of a uniform restart strategy, against the restart threshold T, evaluated using `dplogn` at different levels of censoring. The comparison term `real` is the *actual* performance of a restart strategy T, evaluated *a-posteriori* on the same run: averaging this on multiple runs, one would obtain an estimate of the real $E(t_T)$ for the problem set. We can see that the estimated and real cost differ greatly, but have a similar minimum: this allows $\hat{T}$ to be efficient also with a poor $\hat{F}$, obtained from a heavily censored runtime sample.

Fig. 7 plots the CDF $\hat{F}$ obtained with the non-parametric Kaplan-Meier estimator. This simple model proved similar in performance to `dplogn`, also in the few cases where this latter failed to converge (see again Figg. 1, 2).

For what concerns the universal restart strategy U, its performance on the test set is consistently worse. Its advantage on the training set was predictable, as in our simple scheme 20 copies of Satz-Rand are run in parallel on each training problem, and obviously decreases with c: on sets 7, 8, training cost is actually lower for $c = 0.8, 0.9$. We expect to further reduce training cost, again with a low impact on test performance, by simply running less parallel copies.

6 Discussion

There is only an apparent contradiction between the rapid degradation of the model, following the increase in censored data, and the stability of the performance of the estimated optimal restart strategy. Traditional statistical tests are in fact intended to measure the fit of a pdf along the whole spectrum of possible values. The formula for the restart performance (3) is instead based on the *cumulative distribution* function, which is the integral of the pdf; and on its further integration up to T, which is usually small. This means that the actual shape of a large portion of the distribution is irrelevant,

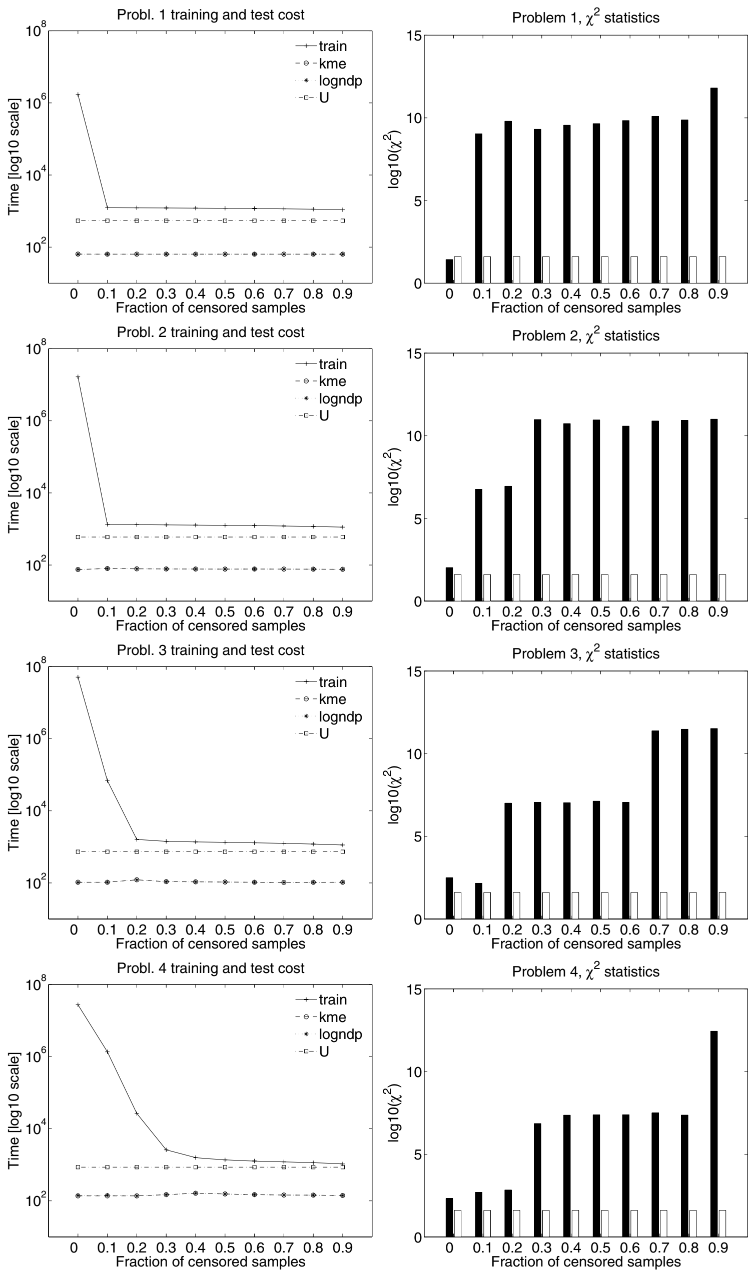


Fig. 1. Problems 1 to 4. Left: the trade-off between training cost (`train`) and test performances of the parametric mixture lognormal-double Pareto (`logndp`), and the non-parametric Kaplan-Meier estimator (`kme`), for different censoring fractions c. `U` labels the performance t_U of the universal strategy on the test set. Right: $\log_{10}$ of the χ^2 statistics for `logndp` (black), compared to $\log_{10}$ of the acceptance threshold (white).

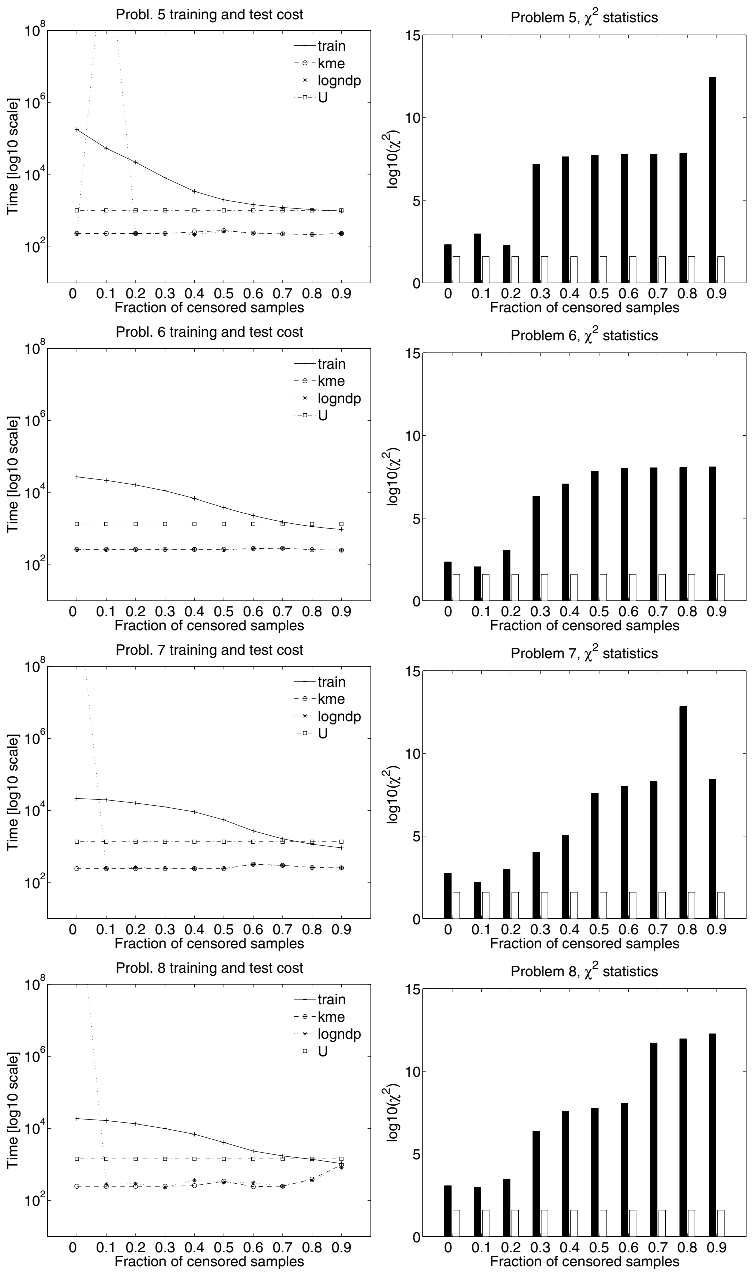


Fig. 2. Problems 5 to 8. Left: the trade-off between training cost (`train`) and test performances of the parametric mixture lognormal-double Pareto (`logndp`), and the non-parametric Kaplan-Meier estimator (`kme`), for different censoring fractions c. `U` labels the performance t_U of the universal strategy on the test set. Right: $\log_{10}$ of the χ^2 statistics for `logndp` (black), compared to $\log_{10}$ of the acceptance threshold (white).

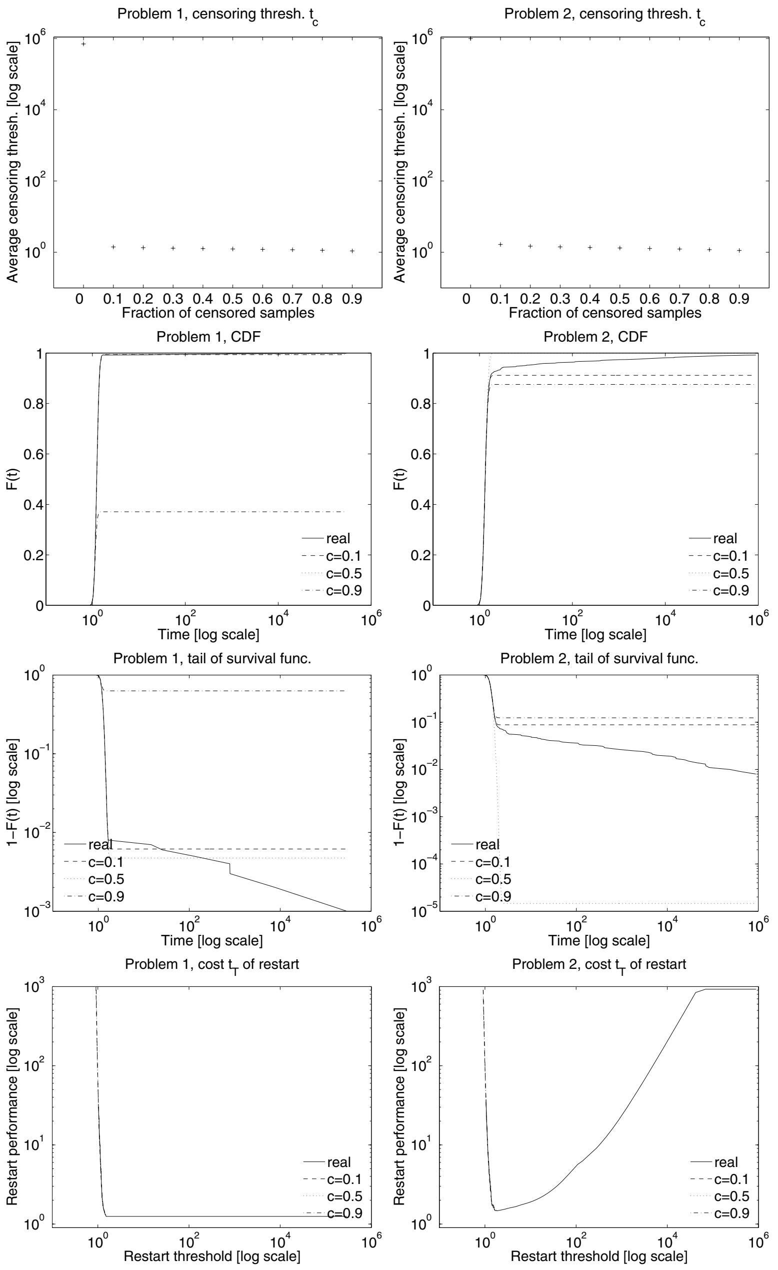


Fig. 3. Problems 1 (left column) and 2 (right column). From top to bottom: average censoring threshold t_c for different fractions of censoring; CDF; tail of the survival function; estimated expected cost of restart for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last three rows refer to a single run. Note the similar minima in last row.

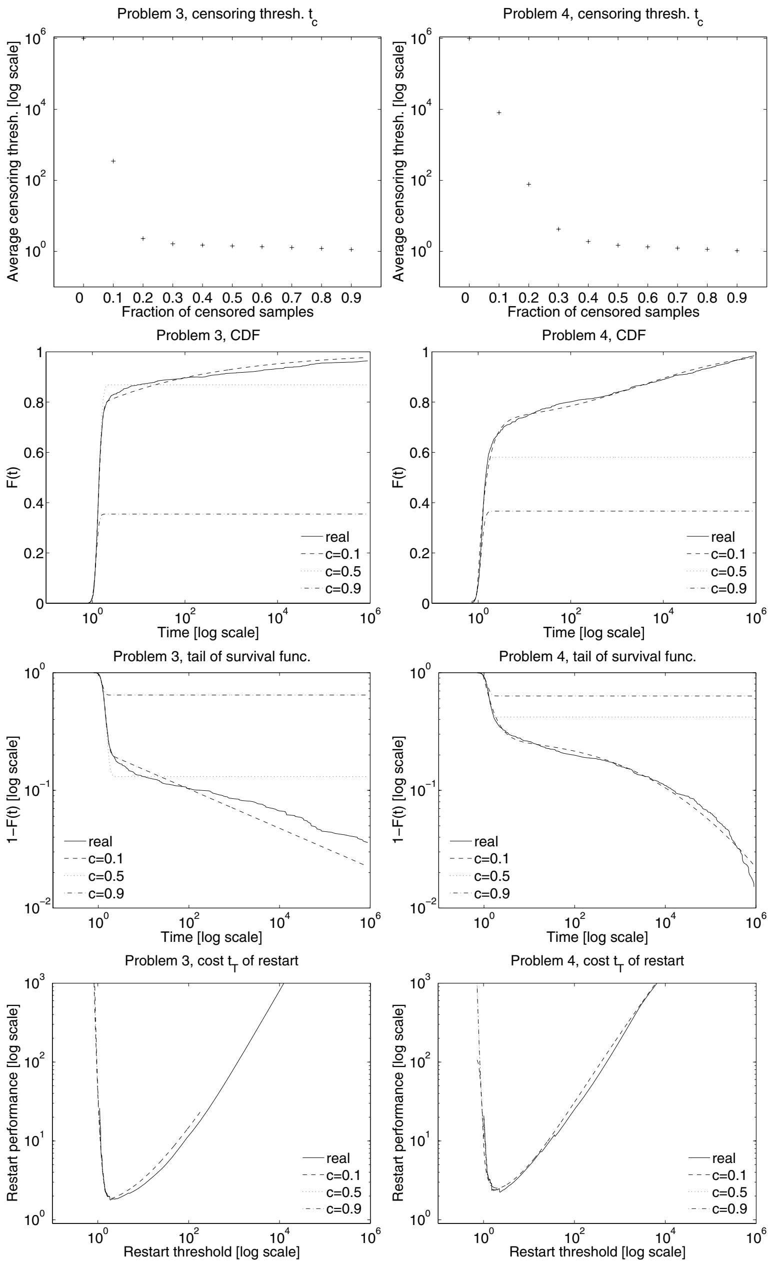


Fig. 4. Problems 3 (left column) and 4 (right column).From top to bottom: average censoring threshold t_c for different fractions of censoring; CDF; tail of the survival function; estimated expected cost of restart for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last three rows refer to a single run. Note the similar minima in last row.

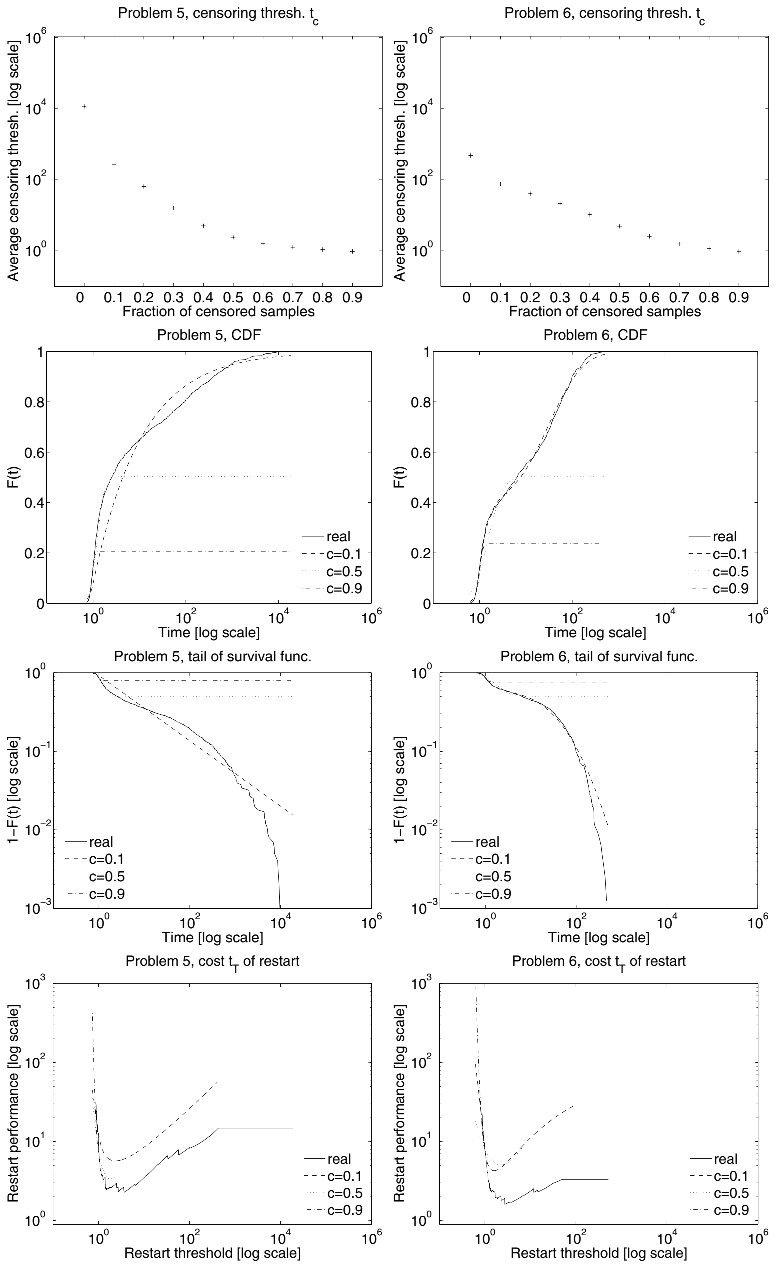


Fig. 5. Problems 5 (left column) and 6 (right column). From top to bottom: average censoring threshold t_c for different fractions of censoring; CDF; tail of the survival function; estimated expected cost of restart for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last three rows refer to a single run. Note the similar minima in last row.

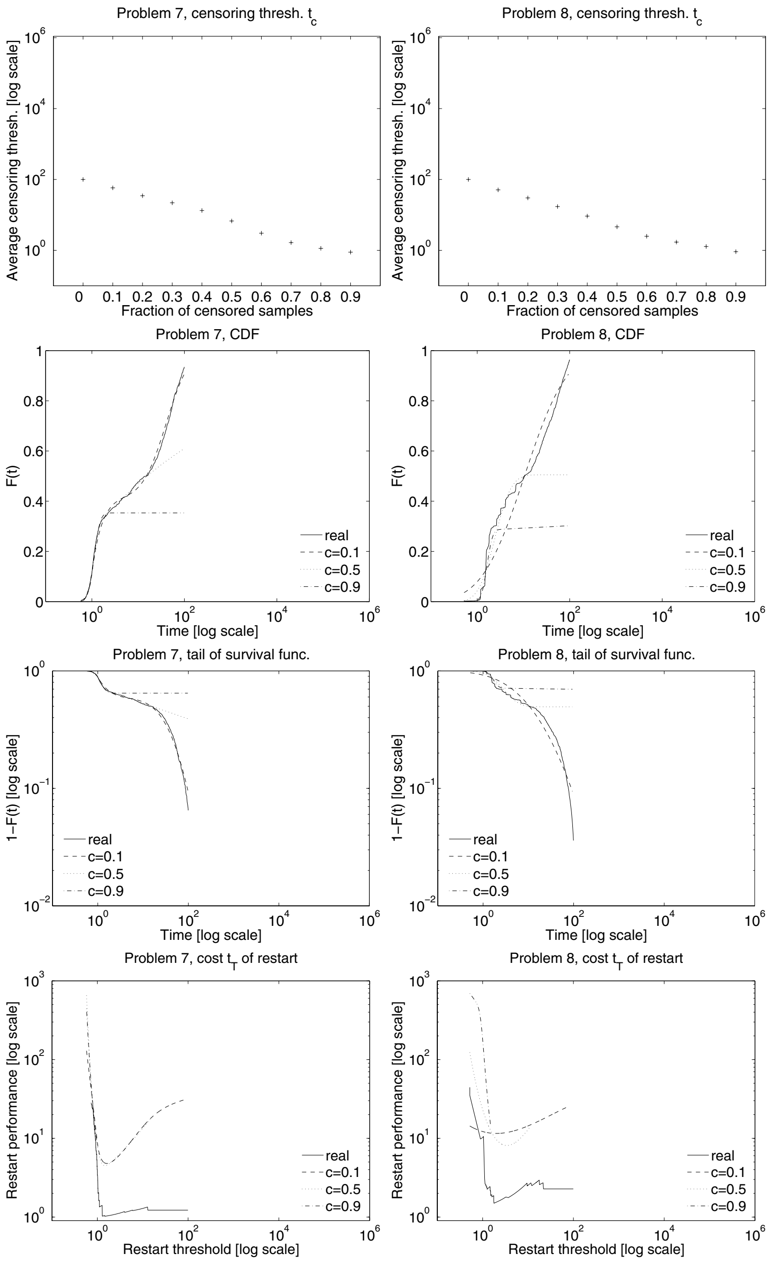


Fig. 6. Problems 7 (left column) and 8 (right column). From top to bottom: average censoring threshold t_c for different fractions of censoring; CDF; tail of the survival function; estimated expected cost of restart for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last three rows refer to a single run. Note the similar minima in last row.

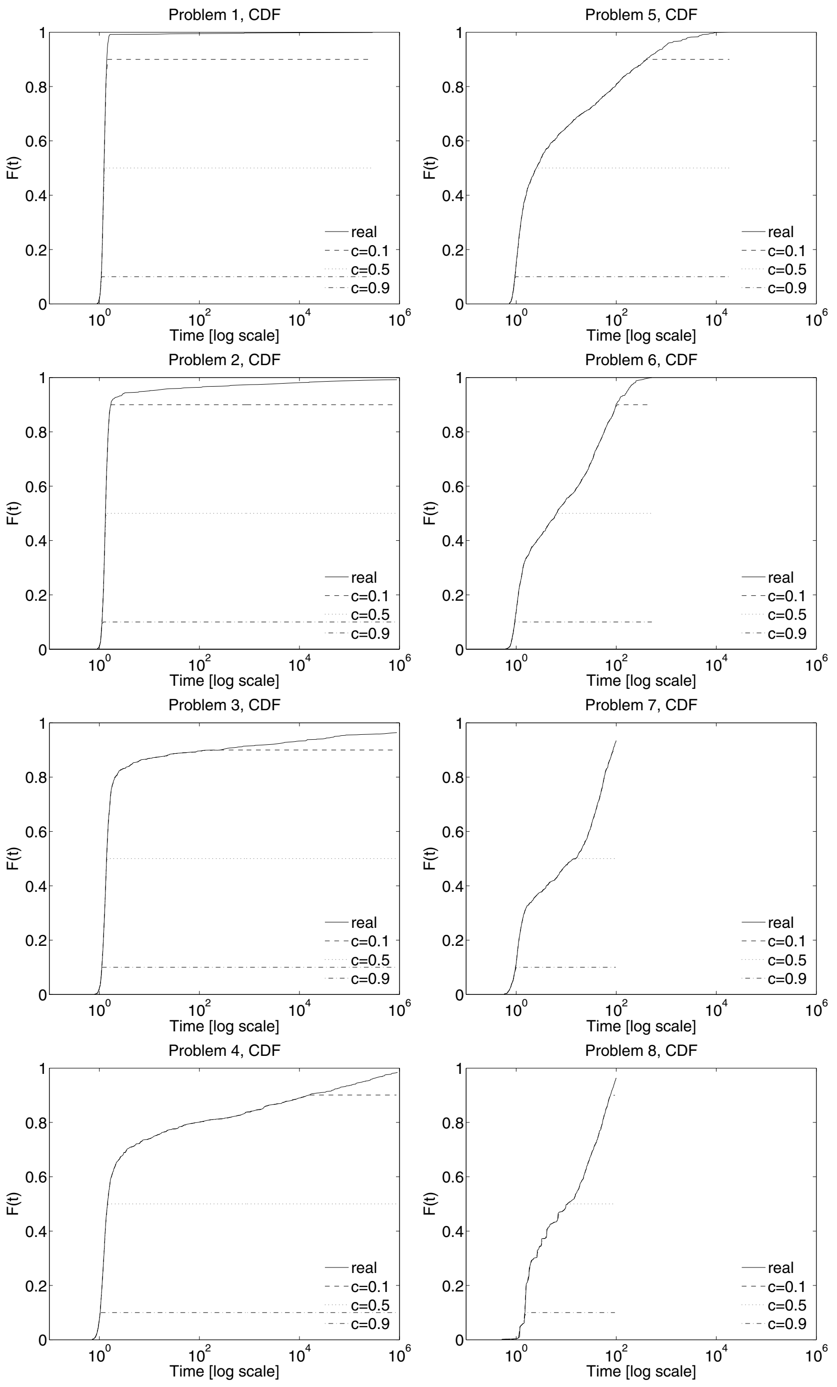


Fig. 7. Problems 1 to 8, Kaplan-Meier estimator

as long as its mass does not vary; while for values lower than T, the integration involved in the CDF acts as a “denoising” filter, making (3) more robust to loss of fit of the model.

From a more practical point of view, our experiments show that even a sub-optimal restart strategy can have a relevant advantage over the universal. This advantage can be obtained for an additional training effort, which can be greatly reduced by censored sampling. On a larger test set, this initial training effort would pay off quite rapidly. Note also that there is no reason to stop the training process, as it is relatively cheap compared to problem solving, and each restart can also be interpreted as a censored sample.

We expect analogous results with any heavy-tailed randomized algorithm. This motivates us to further investigate methods to merge training and exploitation of models of algorithm performance, as in [22]. Current research is aimed at a companion analysis of the effect of censoring in the case of algorithm portfolios, and the combination of universal and estimated optimal restart strategies, the latter based on censored and uncensored samples gathered on a sequence of problems, the former limiting cost in the initial phase, when the model is still unreliable, in a *life-long learning* fashion.

References

1. Gomes, C.P., Fernandez, C., Selman, B., Bessiere, C.: Statistical regimes across constrainedness regions. Constraints **10**(4) (2005) 317–337
2. Rice, J.R.: The algorithm selection problem. In Rubinoff, M., Yovits, M.C., eds.: Advances in computers. Volume 15. Academic Press, New York (1976) 65–118
3. Hogg, T., Williams, C.P.: The hardest constraint problems: a double phase transition. Artif. Intell. **69**(1-2) (1994) 359–377
4. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reason. **24**(1-2) (2000) 67–100
5. Huberman, B.A., Lukose, R.M., Hogg, T.: An economic approach to hard computational problems. Science **275** (1997) 51–54
6. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. **47**(4) (1993) 173–180
7. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1–2) (2001) 43–62
8. Nelson, W.: Applied Life Data Analysis. John Wiley, New York (1982)
9. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: IJCAI97. (1997) 366–371
10. Hoos, H.H., Stützle, T.: SATLIB: An Online Resource for Research on SAT. In I.P.Gent, H.v.Maaren, T.W., ed.: SAT 2000, IOS press (2000) 283–292 `http://www.satlib.org`.
11. Frost, D., Rish, I., Vila, L.: Summarizing csp hardness with continuous probability distributions. In: AAAI/IAAI. (1997) 327–333
12. van Moorsel, A.P.A., Wolter, K.: Analysis and algorithms for restart. In: QEST ’04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference on (QEST’04), Washington, DC, USA, IEEE Computer Society (2004) 195–204
13. Hoos, H.H.: On the run-time behaviour of stochastic local search algorithms for sat. In: Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI’99), Orlando, Florida (1999) 661–666

14. Hoos, H.H., Stützle, T.: Local search algorithms for SAT: An empirical evaluation. Journal of Automated Reasoning **24**(4) (2000) 421–481
15. Hoos, H.H.: A mixture-model for the behaviour of sls algorithms for sat. In: Eighteenth national conference on Artificial intelligence, Menlo Park, CA, USA, American Association for Artificial Intelligence (2002) 661–667
16. Boddy, M., Dean, T.L.: Deliberation scheduling for problem solving in time-constrained environments. Artificial Intelligence **67**(2) (1994) 245–285
17. Hansen, E.A., Zilberstein, S.: Monitoring and control of anytime algorithms: A dynamic programming approach. Artificial Intelligence **126**(1–2) (2001) 139–157
18. Alt, H., Guibas, L., Mehlhorn, K., Karp, R., Widgerson, A.: A method for obtaining randomized algorithms with small tail probalities. Research Report MPI-I-92-110, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany (1992)
19. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence, Menlo Park, CA, USA, American Association for Artificial Intelligence (1998) 431–437
20. Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., Selman, B.: Dynamic restart policies (2002)
21. Gagliolo, M., Schmidhuber, J.: A neural network model for inter-problem adaptive online time allocation. In Duch, W., et al., eds.: Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005, Part 2, Springer (2005) 7–12
22. Gagliolo, M., Schmidhuber, J.: Dynamic algorithm portfolios. In: Ninth International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida, January 2006. (2006)
23. Gent, I., Hoos, H.H., Prosser, P., Walsh, T.: Morphing: Combining structure and randomness. In: Proc. of AAAI-99. (1999) 654–660 — Benchmark avaliable at `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/SW-GCP/descr.html`.
24. Reed, W.: The double pareto-lognormal distribution - a new parametric model for size distribution (2001) — Available at http://www.math.uvic.ca/faculty/reed/index.html.
25. Kaplan, E., Meyer, P.: Nonparametric estimation from incomplete samples. J. of the ASA **73** (1958) 457–481

Watched Literals for Constraint Propagation in Minion[⋆]

Ian P. Gent[1], Chris Jefferson[2], and Ian Miguel[1]

[1] School of Computer Science, University of St Andrews, St Andrews, Fife, UK
[2] Oxford University Computing Laboratory, University of Oxford, Oxford, UK
{ipg, ianm}@dcs.st-and.ac.uk, Chris.Jefferson@comlab.ox.ac.uk

Abstract. Efficient constraint propagation is crucial to any constraint solver. We show that *watched literals*, already a great success in the satisfiability community, can be used to provide highly efficient implementations of constraint propagators. We describe three important aspects of watched literals as we apply them to constraints, and how they are implemented in the MINION constraint solver. We show three successful applications of to constraint propagators: the sum of Boolean variables; GAC for the 'element' constraint; and GAC for the 'table' constraint.

1 Introduction

Efficient constraint propagation is the bedrock of any implementation of constraint programming. We show that *watched literals* can be used to provide faster implementations of constraint propagators. Watched literals are one of the main reasons for the dramatic improvements seen in SAT solvers in this decade [12]. They allow the key progagation algorithm, unit propagation, to run much faster than previous implementations. Used as triggers to fire constraint propagations, watched literals have three features different to triggers as normally used. Watched literals only cause propagation when a given *variable-value pair* is deleted; their triggering conditions can be changed *dynamically* during search; and they remain *stable* on backtracking so do not use memory for restoration. These features can be used separately, and we report on doing so.

We first identify what we consider to be the two key benefits of watched literals: the reduction of work when no propagation is possible, and the reduction in work on backtracking. We detail the three aspects of watched literals mentioned above. Then we report implementations of three different constraints using watched literals. The first is a slight generalisation of the SAT clause, and so it is no surprise that this allows us to improve the speed of propagating SAT clauses in a constraint solver. In fact, our results show that SAT clauses propagate almost as fast as in a state-of-the-art SAT solver. Next, we show that

⋆ Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Fellowship. This research was supported by EPSRC Grants GR/S30580 and EP/D032636 and the EPSRC funded SymNET. We thank Peter Jeavons, Steve Linton, Iñes Lynce, Angela Miguel, Karen Petrie, Judith Underwood, and our CP reviewers.

watched literals are ideally suited to implementing generalised arc consistency (GAC) for the 'Element' constraint. Using classical triggers, it is hard to achieve GAC efficiently. Our third example is to implement GAC for the 'Table' constraint, i.e. an arbitrary constraint specified by an explicit list of allowed tuples, based on GAC-2001/3.1 [4]. By using watched literals, we automatically convert this from being a constraint-based to a support-based algorithm. Furthermore, by the nature of watched literals, we do not need to restore the state of support data structures on backtracking.

We report on how we provide the infrastructure to allow watched literals to work in a practical constraint solver. All our implementation and experimentation is based on the MINION constraint solver [7]. MINION's design principles are to reduce user choice to allow greater optimisation; and to maintain a low memory profile, especially for state information that needs to be restored on backtracking. We have shown that this does lead to exceptionally fast runtimes [7]. Thus, the run-time improvements we report here are compared to the state of the art. Watched literals have been incorporated into MINION. Since MINION is open source, our implementations are available for use and research.[1]

2 Motivation and Terminology

Conventionally, constraints are triggered by a particular variable's domain being changed in one of three ways: by being assigned, by having any value in its domain removed, or by having either its lower or upper bound removed. We call these kinds of triggers **classical**. A watched literal can trigger on the removal of any given variable-value pair in the problem. We use the word "literal" for a variable-value pair. This is completely consistent with its usage in SAT, where a literal is a variable together with a polarity. A **literal trigger** is one which causes its constraint to act when the given value is removed from that variable. Literal triggers can be found in the cc(FD) language, in which constraints can be attached directly to individual domain elements [9]. We use the word 'literal' is to avoid a clash with the classical 'domain trigger', which will fire when *any* value of its variable is removed. After a watched literal is triggered, its constraint can move the trigger to another literal, delete it, or add a new watched literal. We say that a **dynamic trigger** is one which can be moved in this fashion. Dynamic triggers are a valid concept on non-literal triggers. We can have dynamic triggers which fire on domain removal, bounds removal, or variable assignment, and these are implemented in MINION. By contrast, a **static trigger** is one whose firing conditions are guaranteed to remain fixed during the lifetime of a constraint.[2] Watched literals have to remain valid on backtracking, so that we can avoid storing their state information in backtrackable memory. A trigger with this property is a **backtrack-stable trigger**, or just 'stable trigger' for short.[3] We

[1] `http://minion.sourceforge.net`. We used revision 138 in this paper.

[2] Under this definition a classical bounds trigger is *static* although the exact domain removals that trigger it will change.

[3] We thank Steve Linton for suggesting this name.

explain this concept in more detail below. For brevity and to emphasise the historical linkage with the SAT concept, we say that a **watched trigger** is a dynamic and backtrack-stable one. Thus a **watched literal** is just a dynamic, stable, literal trigger.

It might be helpful to think of a dynamic trigger being a generalisation of a classical bounds trigger. A bounds trigger fires, say, on the value 2, but after it propagates it will fire on the new lower bound, perhaps 7. For a dynamic trigger, the constraint itself decides where the new trigger will be, rather than being fixed at the new lower bound. It can move the trigger to another value of the same variable, to any value of any other variable in the constraint. It can move some or all dynamic triggers associated with a constraint to different places. It can even add or delete dynamic triggers.

We see the key advantage of watched literals and dynamic triggers as the flexibility it gives constraint propagators to avoid unnecessary work. In many situations where a constraint cannot propagate, it causes *no work at all.* To enable this, each constraint using dynamic triggers must provide what we call the **propagation guarantee**: that its intended level of consistency is satisfied *unless* at least one of its dynamic triggers fires. For a watched literal, when a domain value is deleted, we only need to wake up constraints which are watching that literal (i.e. that value of its variable.) The advantage extends to, for example, dynamic assignment triggers. A constraint on ten variables may be able to set up dynamic assignment triggers on just two of them: this constraint will cause no work when of the other eight are assigned.

Watched triggers (but not dynamic triggers) have another significant advantage because they are backtrack-stable. This lets us reduce memory management overheads greatly. Conventionally, on backtracking, we ensure that the state of all propagators are exactly as they were when we left that node going forwards, necessitating some data structures and maintenance to support this.[4] Backtrack-stability lets us lift this restriction. It is enough to ensure that when we backtrack, all constraints are in a state which lets them provide the propagation guarantee. It is perfectly acceptable for a constraint's set of watched triggers to be completely different on returning to a node to the set when we left the node, as long as both sets provide the propagation guarantee in that search state. The ideal way to achieve this is for constraints to define sets of watched triggers which do not need to be changed on backtracking, i.e. backtrack-stable triggers. Where we can do this perfectly, the constraint needs put nothing into backtrackable memory. This means that no copying is done on backtracking, so once again we have reduced the cost associated with the constraint to zero.

Defining backtrack-stable triggers can require nonintuitive thinking, but some general rules apply and we will see examples in the detailed descriptions to follow. Many cases are simple, because the notion of support monotonic in many constraints: i.e. if a set of values supports some literal, they still support it as we restore values to domains on backtracking. However, more subtle cases arise,

[4] This is not always true. For example, Régin describes how versions of the MAC algorithms can be modified to avoid state restoration upon backtracking [13].

as in GAC for the table constraint in Section 5. When we fail to find support for a value and so the constraint forces its deletion, the natural thing to do is to remove the watch on the literal. This is correct for dynamic triggers but not for watched triggers. When we backtrack, the value we have just deleted will be restored to its domain, as will the literal whose deletion caused its removal. If we delete the trigger, we will lose the propagation guarantee on backtracking. Instead, we leave the watched literal in place.

There is one general disadvantage that should be mentioned with watched triggers, although not with dynamic triggers. Because triggers may move arbitrarily between leaving a node and returning to it, it is often not possible to use a propagation algorithm which is optimal in the worst case in terms of propagation work performed down a single branch. An example is our variant of GAC-2001/3.1 below. This has to be set against the great potential for efficiency gains we have outlined, so we do not think this will often be a major disadvantage. The same problem arises in SAT but watched literals are very widely used, and we will see experimentally that our non-optimal watched version of GAC outperforms an optimal but unwatched one.

3 Watched Literals for Sum of Booleans

The first constraint we consider is the sum of an array of Boolean variables B being greater than or equal to a constant value c. Where $c = 1$, this is just a SAT clause and our method is exactly the standard use of watched literals in SAT [12]. As such it serves as a good introduction to watched literals for constraint programming. Our approach is only novel in being incorporated in a constraint solver: watched literals for the (more general) weighted sum of a Boolean array was reported by Chai and Kuehlmann [5].

We watch $c + 1$ different literals in the domain of $c + 1$ different variables in B. In each case we watch the value 1. If, at initialisation, we can find only c such literals we immediately set them all to be 1, and if we cannot find even that many we fail: in neither case is any more work on this constraint required. In general, the watched literals are $B[i_1] = 1, B[i_2] = 1, \ldots, B[i_{c+1}] = 1$. The watched literals, on their own, more than satisfy the constraint, so all other literals could be set to 0 without any propagation happening.

When one of the values being watched is removed, we have some $B[i_j] = 0$. In this case, we have to find a new value i'_j so that $B[i'_j]$ still has 1 in its domain and is not one of the other k literals currently being watched. In the worst case, we have to scan each unwatched variable in the array to see if it has 1 in its domain. If we find such a literal, we simply stop watching $B[i_j] = 1$ and start watching $B[i'_j] = 1$: we call this moving the watch from one literal to the other. From the propagation guarantee, we cannot propagate until one of these literals is deleted. On backtracking, we can only make the constraint looser by enlarging domains, so the watched literals are backtrack-stable. The more interesting case is if we cannot find any other literal to watch. In this case, we know there are at most c literals with 1 in their domain, and all must be set to 1 to satisfy the constraint.

GAC Propagator for Sum of Booleans $\sum B[1..n] \geq \mathsf{c}$

```
OneRemovedFromWatchedVar($i$)
        Triggered by DomainRemovalOf $B[i] = 1$
A01     $j :=$ Last
A02     repeat $j := j + 1$ mod $n - c - 1$
A03     until ($j =$ Last) or ($1 \in Domain(B[$UnWatched$[j]])$)
A04     if ($j \neq$ Last) then          // We found a new literal to watch, so update data structures
A04.1       MoveWatchFrom $B[i] = 1$ To $B[$UnWatched$[j]] = 1$;
A04.2       UnWatched$[j]$ := $i$; Last := $j$
A05     else                             // No new literal found to watch, so at most c 1's possible in B
A05.1       foreach $k$ such that WATCHED$[B[k] = 1]$
A05.2           if($k \neq i$) then AssignVariable($B[k] = 1$)
```

Fig. 1. Propagator for the Boolean Sum constraint. `UnWatched` is an array of integers. Its values are all different and represent the set of unwatched literals: in general it will not remain sorted during search. `Last` points to the array element in `UnWatched` that was set to be watched the last time we updated this constraint. Neither `UnWatched` nor `Last` needs to be restored on backtracking: therefore both reside in non-backtrackable memory. AssignVariable sets a variable to a value and triggers necessary propagations while MoveWatchFromTo updates the watched literal store.

So we can immediately set all of them to be 1. This will either cause immediate failure (if another one of the watched literals can no longer be 1) or guarantees to satisfy the constraint. However, we continue to watch all the $c + 1$ literals we were watching previously. All watched literals are now set and will cause no propagation, so there is no cost in leaving them in place. Pseudocode for this is shown in Fig. 1. As mentioned above, it is essential to leave the watches in place so that they will be correct on backtracking. In the cases studied in this paper it is always correct to leave watches in place when a constraint is either satisfied or unsatisfied. The last time the constraint was fired, the propagation guarantee held and the set of watched triggers were backtrack-stable. Therefore, if we backtrack past the current point in search, we return to a state where we know a set of backtrack-stable triggers satisfied the propagation guarantee.

Our first set of experiments are on SAT problems. These are encoded into a constraint problem with a Boolean variable for each SAT variable and a Boolean sum greater than 1 for each clause. In no way can Minion compete with specialist SAT solvers because it does not have specialised heuristics or techniques such as nogood learning. However, these experiments demonstrate the value of incorporating watched literals into a constraint solver for this kind of constraint. We compare Minion using its sum-$\geq$ constraint using classical triggers against our new implementation using watched literals. For reference we also compare these against the same model encoded into ILOG Solver 5.3 (a more recent version was not available to us) and a state-of-the-art SAT solver, MiniSAT [6]. We ran on two sets of SAT benchmarks from SATLib [10], uniformly unsatisfiable random instances with 150 variables, and the QG benchmark set. Results are

Table 1. Experiments with Random SAT instances (mean of 100) and QG SAT instances. Bold indicates which of the three constraint solvers searched most nodes per second – nodes searched in each case were identical. Italics in the Solver column indicate timeouts after 1 hour. The number of SAT variables in QG$i.n$ is n^3.

	Classical			Watched		ILOG Solver 5.3		MiniSat		
Problem	Time(s)	Nodes	Nds./s	Time(s)	Nds./s	Time(s)	Nds./s	Time(s)	Decisions	Dec./s
Random	96.20	621,798.1	**63,360**	100.0	62,180	462.9	13,225	0.0858	3,577.1	41,614
QG1.7	8.13	58	7	7.83	7	2.77	**21**	0.14	121	864
QG1.8	628.47	188,270	300	101.89	**1,848**	*3604.09*	27	0.75	7,951	10,601
QG2.7	8.16	32	4	7.89	4	2.20	**14**	0.125	54	432
QG2.8	1,137.31	340,747	300	165.16	**2,063**	*3604.09*	25	10.328	48,933	4,737
QG3.8	1.08	150	139	1.06	141	0.36	**415**	0.046	283	6,152
QG3.9	18.42	82,404	4,474	14.09	**5,847**	107.63	766	6.703	55,003	8,205
QG4.8	1.06	909	855	1.03	882	0.92	**987**	0.078	1,003	12,859
QG4.9	1.56	461	295	1.52	304	0.94	**491**	0.046	28	608
QG5.9	2.83	187	66	2.75	68	1.30	**143**	0.62	26	419
QG5.10	5.44	1,453	267	4.63	**314**	10.84	134	0.093	74	795
QG5.11	6.77	506	75	6.20	**82**	7.97	63	0.14	79	564
QG5.12	269.84	139,581	517	76.33	**1,829**	2132.67	65	0.296	1,440	4,864
QG5.13	4,847.28	1,798,176	371	1,125.83	**1,597**	*3602.88*	48	6.156	30,776	4,999
QG6.9	2.19	28	13	2.13	13	0.53	**51**	0.046	16	348
QG6.10	3.53	313	89	3.33	94	2.05	**153**	0.109	458	4,202
QG6.11	7.58	2,522	333	5.66	**446**	21.03	120	0.296	2,632	8,892
QG6.12	50.09	26,847	536	20.95	**1,281**	358.88	75	4.484	22,519	5,022
QG7.9	2.19	8	4	2.14	4	0.50	**14**	0.046	8	174
QG7.10	3.81	816	214	3.47	**235**	4.58	178	0.093	124	1,333
QG7.11	16.64	11,616	698	8.16	**1,424**	104.94	111	0.187	1,866	9,979
QG7.12	277.41	159,907	576	70.53	**2,267**	2370.37	68	0.593	5,731	9,664
QG7.13	786.55	312,108	397	180.66	**1,728**	*3602.19*	52	0.265	1,307	4,932

shown in Table 1. We ran experiments on a variety of machines for this paper, but any comparison between methods on an identical instance were run on a single machine for consistency to enable valid comparisons. Our SAT experiments were run under Windows (XP SP2), with a Pentium 4 3GHz and 2GB of RAM, compilation being done using g++ 3.4.4 under cygwin.

On random instances, watched literals are a slight overhead, presumably because all clauses have only 3 literals. Both versions of MINION outperform Solver on random instances. MiniSAT easily outperforms both versions of MINION. Although MINION does search more nodes per second than MiniSAT, its very small runtimes may make these figures unduly affected by setup times. For the structured QG benchmarks, we see that in all cases the watched literal propagator outperforms the classical version by up to 6 times in run time. ILOG Solver does outperform MINION on a number of instances, but only on easy instances: we suspect that this is because of MINION's larger initialisation costs. MINION searches faster on all instances where either solver takes even 10 seconds. On hard instances MINION can search many times faster than Solver, up to 82 times in the case of QG2.8. MINION is again outclassed easily by MiniSAT. However, on instances where it takes 10 seconds, MINION, is never as much as 6 times slower than MiniSAT in terms of decisions per second. Certainly MINION is not a competitive SAT solver. However, the efficiency with which it propagates clauses does suggest that it is ideally suited for solving hybrid instances containing significant numbers of SAT clauses together with more expressive constraints.

4 Element Constraint

The "Element" constraint is a great addition to the expressivity of a constraint language. It allows one to use an integer variable for the index of an array [8]. Suppose that A is an array of n integer variables and that Index and Result are two more integer variables. The Element constraint is that $A[\mathit{Index}] = \mathit{Result}$. Although the constraint is implemented in constraint toolkits such as ILOG Solver, Gecode [14], and Choco [11], we are not aware of a literature on how to propagate it. This may be because, using classical triggers, some aspects of Element are hard to propagate efficiently enough to repay the overheads.[5] In contrast, using watched literals, establishing and maintaining GAC is straightforward to implement and efficient. In particular, where little propagation occurs little overhead is incurred. We start by establishing the exact definition of GAC:

Theorem 1. *The domains of variables in an Element constraint are Generalised Arc Consistent if and only if all of the following hold:*

$$\mathit{Dom}(\mathit{Index}) = \{i\} \Longrightarrow \mathit{Dom}(A[i]) \subseteq \mathit{Dom}(\mathit{Result}) \tag{1}$$

$$i \in \mathit{Dom}(\mathit{Index}) \Longrightarrow \mathit{Dom}(A[i]) \cap \mathit{Dom}(\mathit{Result}) \neq \emptyset \tag{2}$$

$$\mathit{Dom}(\mathit{Result}) \subseteq \bigcup_{i \in \mathit{Dom}(\mathit{Index})} \mathit{Dom}(A[i]) \tag{3}$$

Proof: (If) Suppose all the conditions hold. By (2) there is a value to support each value of Index. If $\mathit{Dom}(\mathit{Index})$ is a singleton then (1) and (3) show that $\mathit{Dom}(A[\mathit{Index}]) = \mathit{Dom}(\mathit{Result})$, and any value of one variable is supported by the same value of the other. If $|\mathit{Dom}(\mathit{Index})| > 1$, every value for each $A[i]$ is unconstrained since it is supported if $\mathit{Index} \neq i$. Also, (3) ensures that each value j of Result is supported by a pair $\mathit{Index} = i, A[i] = j$. (Only If) Suppose the domains of Index, Result, and each $A[i]$, are GAC. If $\mathit{Dom}(\mathit{Index}) = \{i\}$ then certainly any value in $\mathit{Dom}(A[i])$ must be in $\mathit{Dom}(\mathit{Result})$. If $i \in \mathit{Dom}(\mathit{Index})$ then there must be at least one value in the domain of both $\mathit{Dom}(A[i])$ and $\mathit{Dom}(\mathit{Result})$. Finally, for any value $v \in \mathit{Dom}(\mathit{Result})$ there must be an index $i \in \mathit{Dom}(\mathit{Index})$ such that $v \in \mathit{Dom}(A[i])$. ***QED***

This proof shows the close link between the conditions and the *support* for each value of each variable in the constraint. We make this explicit, by describing support for each value of $A[i]$, Index and Result. Notice that each kind of support involves at most two other variables, instead of a tuple involving all variables in the constraint. There are three cases, corresponding exactly to (1), (2) and (3) above.

Support for $A[i] = j$**: either** $|\mathit{Dom}(\mathit{Index})| > 1$ **or** $j \in \mathit{Dom}(\mathit{Result})$.
Support for $\mathit{Index} = i$**:** any j such that $j \in \mathit{Dom}(A[i])$ and $j \in \mathit{Dom}(\mathit{Result})$.
Support for $\mathit{Result} = j$**:** any i such that $i \in \mathit{Dom}(\mathit{Index})$ and $j \in \mathit{Dom}(A[i])$.

We now describe how we proceed if the support is lost, i.e. when values are removed which were supporting a variable-value pair in one of the cases above.

[5] Choco and Gecode achieve GAC on Element, but we believe Solver 5.3 does not.

1. There are two cases where we might have lost support for $A[i] = j$. The support in either case is independent of i, depending only on the values of *Index* or *Result* respectively.
   (a) If *Index* is assigned to i (i.e. its domain is the singleton $\{i\}$) then we have to remove all values from $Dom(A[i])$ not in the domain of *Result*. This can be achieved by a classical trigger on the assignment of *Index*.
   (b) If a value j is removed from the domain of *Result*, and *Index* has already been assigned to i, then we remove j from the domain of $Dom(A[i])$. There are a number of ways to achieve this: in our pseudocode and implementation we use a classical domain reduction trigger on *Result*.
2. If the value j is removed from either $Dom(A[i])$ or $Dom(Result)$, we try to find another value j' in the domain of both. If we succeed, this is the new support for $Index = i$. If not, then we remove i from the domain of *Index*. We implement this by *watching* the literals $A[i] = j$ and $Result = j$. If we find j' then we move the watched literals to be on $A[i] = j'$ and $Result = j'$. If we do not find a new support, we leave the watched literals in place, so that they will be correct when j returns to the domain on backtracking.
3. If the value i is removed from $Dom(Index)$, or j is removed from $Dom(A[i])$, we try to find another pair of values i', j' with $i \in Dom(Index)$ and $j \in Dom(A[i])$. These will be the new support for $Result = j$, or else we insist that $Result \neq j$. We implement this by *watching* $Index = i$ and $A[i] = j$, moving these to $Index = i'$ $A[i'] = j'$ if possible. If not, we again leave the watches in place.

We find this development very natural. We started from the definition of domains being GAC, moved to what is needed to support each value of each domain, and finally showed how new supports could be sought when old supports are lost. This last stage can be implemented in Minion. Pseudocode for this implementation is in Fig. 2, where the four constituent functions correspond exactly with the four cases above. We have not discussed initialisation because it is a straightforward variant on the pseudocode of Fig. 2. Finding the initial watched values is essentially the same as finding new ones from old, and values are removed if we cannot find initial supports for them.

We have not compared our approach empirically with existing classical implementations, but we argue that watched literals have the potential to be much faster. For each i in $Dom(Index)$ we are watching two things, and for each j in $Dom(Result)$ we are watching two things. If the array is size 500 and each domain is of size 100, we therefore have 1,200 literals to watch in addition to the conventional triggers on *Index* and *Result*. Yet in total in $A[i]$, *Index*, and *Result*, there are 500,600 domain elements, so we watch less than 0.25% of all domain elements. This argument also shows why it is important that we do not need to restore trigger data on backtracking when using watched literals. If we did have to, we would have to restore the values of 1,200 triggers on backtracking, an overhead which would outweigh the benefits we have gained. The only genuine efficiency loss compared to conventional methods happens when we fail

GAC for Element Constraint: $A[Index] = Result$

```
SupportLostForArrayValue-a($i$)
        Triggered by AssignmentOf $Index = i$
                              // Remove from domain of $A[Index]$ any values not in domain of $Result$
A01     foreach $k$ in the current domain of $A[i]$
A01.1       if $k \notin Domain(Result)$ then
A01.2           RemoveFromDomain($A[i]$,$k$)

SupportLostForArrayValue-b()
        Triggered by AnyDomainRemovalOf $Result$
                              // If $Index$ is assigned to $i$ then $Dom(A[i]) \subseteq Dom(Result)$
B01     if VariableIsAssigned($Index$) then
B02         foreach $k$ in the current domain of $A[i]$
B02.1           if $k \notin Domain(Result)$ then
B02.2               RemoveFromDomain($A[i]$,$k$)

SupportLostForIndexValue($i, j$)
        Triggered by RemovalOf $A[i] = j$ or RemovalOf $Result = j$
                  // Previously we supported $Index = i$ because $j \in Domain(A[i]) \cap Domain(Result)$
                              // Now we must find a replacement for $j$ or insist that $Index \neq i$
C01     foreach $k$ in the current domain of $A[i]$
                              //See caption for details of how we step through domains
C02         if $k$ in the current domain of $Result$ then
C02.1           MoveWatchFrom $A[i] = j$ To $A[i] = k$;
C02.2           MoveWatchFrom $Result = j$ To $Result = k$;
C03.3           return                         // We are finished, leave function
C04     endforeach                             // We failed to find a new support so $Index \neq i$
C05     RemoveFromDomain($Index$,$i$)

SupportLostForResultValue($i, j$)
        Triggered by RemovalOf $Index = i$ or RemovalOf $A[i] = j$
        // Previously we supported $Result = j$ because $i \in Domain(Index)$ and $j \in Domain(A[i])$
                              // Now we must find a replacement pair for $(i, j)$ or insist that $Result \neq j$
D01     foreach $k$ in the current domain of $Index$
D02         if $j$ in the current domain of $A[k]$ then
D02.1           MoveWatchFrom $A[i] = j$ To $A[k] = j$;
D02.2           MoveWatchFrom $Index = i$ To $Index = k$;
D03.3           return
D04     endforeach                             // We failed to find a new support so $Result \neq j$
D05     RemoveFromDomain($Result$,$j$)
```

Fig. 2. Propagator for the Element constraint. Note that we mix types of triggers freely. The first function triggers when a variable is assigned, and the second triggers on a domain removal. The final two functions both trigger when one of two literals being watched is removed. At line C01 search the domain starting from j, and looping back to j if we reach the end of the domain. A similar approach is taken at D01.

to find new support, for example in line C01 in Fig. 2. We have to search the whole domain instead of just the remaining part of the domain.

To compare with our watched literal implementation, we wrote two other propagators using classical triggers. Both are triggered any time any literal is removed from a variable, and are told the literal removed. One of these accomplishes GAC, by looking at all literals which could have lost support by the removal of this literal, and checking if support for them still exists.[6] This has the advantage of being very space efficient, using no memory which must be backtracked at all, at the cost of extra computation at each node. The second, non-GAC implementation, checks only each time any variable is assigned, and performs only those checks which are possible in O(1) time.

Langford's Problem is problem 24 in CSPLib: $L(k, n)$ requires finding a list of length $k * n$, which contains k sets of the numbers 1 to n, such that for all $m \in \{1, 2, \ldots, n\}$ there is a gap of size m between adjacent occurrences of m. For example, `41312432` is a solution to $L(4, 2)$. We modelled $L(2, n)$ in Minion using two vectors of variables, V and P, each of size $2n$. Each variable in V has domain $\{1, 2, \ldots, n\}$, and V represents the result. For each $i \in \{1, 2, \ldots\}$ the $2i^{th}$ and $2i + 1^{st}$ variables in P are the first and second positions of i in V. Each variable in P has domain $\{0, 1, \ldots, 2n - 1\}$, indexing matrices from 0. We write $=_{\texttt{elem}}$ to distinguish the usage of the element constraint from indexing of a vector by a constant. Where i ranges from 1 to n, the constraints are:

$$\begin{aligned} V[P[2 * i]] &=_{\texttt{elem}} i \\ V[P[2 * i + 1]] &=_{\texttt{elem}} i \\ P[2 * i] &= i + P[2 * i + 1] \end{aligned}$$

We found all solutions to Langford's problem up to $n = 8$ using this model. Experiments for increasing values of n were performed for our three propagators for element presented in this paper. These were run under Mac OS X (10.4.6) running on a 1.2Ghz PowerPC G4 with 768MB RAM compiled with g++ 4.0.1. Results are presented in Table 2. Performing GAC on the element constraints improves solving time by an order of magnitude. The watched GAC propagator is over twice as fast as the non-watched GAC propagator, and in terms of nodes searched per second is only slightly slower than the non-GAC propagator. We also performed experiments on constraint encodings of Quasigroup construction problems, shown in Table 3. Experiments were performed under Windows as described earlier, and under Linux (Fedora Core 4) with a Pentium 4 3.4 GHz dual processor with 8GB RAM, with compilation by g++ 4.0.2. On QG3, GAC performs no additional propagation, but remarkably our GAC algorithm using watched literals is *faster* than the non-GAC algorithm. We suggest this is because of the benefit of watched literals avoiding unnecessary work. On QG7, we get significant search reductions, but here speed per node is greatly reduced by the extra work and the overheads are not repaid. In both cases the watched literal GAC propagator is much faster than the classical version.

[6] This approach is similar to the GAC algorithm in Choco, as seen in its source code.

Table 2. Comparison of propagators in Minion on Langford's problem

		Non-GAC			Classical-GAC		Watched-GAC		
	Solutions	Time(s)	Nodes	Nodes/s	Time(s)	Nodes	Time(s)	Nodes	Nodes/s
L(4,2)	2	<0.1	378	-	<0.1	46	<0.1	46	-
L(5,2)	0	0.1	6,956	-	<0.1	694	<0.1	694	-
L(6,2)	0	2.3	169,275	73,000	0.6	15,388	0.3	15,388	59,000
L(7,2)	52	76.8	4,912,580	64,000	16.7	413,573	7.5	413,573	55,000
L(8,2)	300	3,276.9	176,320,552	54,000	635.7	13,471,366	267.2	13,471,366	50,000

Table 3. Propagators in Minion on Quasigroup Existence Problems. (QG3.10 was not solved in an hour by any propagator.) QG3 ran under Windows, QG7 under Linux.

	Non-GAC			Classical-GAC		Watched-GAC		
Problem	Time(s)	Nodes	Nodes/s	Time(s)	Nodes	Time(s)	Nodes	Nodes/s
QG3.6	0.016	51	-	0.031	51	0.031	31	-
QG3.7	0.031	11	-	0.047	11	0.016	11	-
QG3.8	7.031	105,414	14,992	43.453	105414	6.453	105,414	16,335
QG3.9	0.047	26	-	0.141	26	0.031	26	-
QG3.11	0.078	132	-	0.375	132	0.078	132	-
QG7.7	<0.1	844	-	0.03	311	0.02	311	-
QG7.8	0.08	12,450	155,625	0.5	4,628	0.33	4,628	14,024
QG7.9	<0.1	233	-	0.02	83	0.01	83	-
QG7.10	205.56	31,383,717	152,674	840.33	3,408,114	329.8	3,408,114	10,333.9

5 Watched-GAC for the Table Constraint

The 'Table constraint' provides generalised arc consistency for any user-defined constraint, given by a list of acceptable tuples of the variables involved in the constraint. This can be very useful where critical parts of a problem have no natural expression in primitive constraints, but which need to be propagated effectively. The table constraint can be implemented using any GAC algorithm, provided it is suitably adapted to work correctly in a backtracking environment. We base our algorithm on GAC-2001/3.1 [4].

This section shows the third significant benefit of using watched literals. We have already seen it greatly speed up propagation of SAT-like constraints and enable the simple and efficient implementation of GAC for the element constraint. Here, we show that we can convert an easy-to-implement but coarse-grained GAC algorithm into a fine-grained algorithm where the required data structure maintenance is provided entirely by an already-implemented central infrastructure. GAC-2001/3.1 was presented as a *coarse-grained* algorithm, i.e. it is a constraint-oriented propagation algorithm [4]. By using watched literals, we convert it into a *fine-grained* algorithm, i.e. "the deletion of a value in the domain of a variable will be propagated only to the affected values in the domains of other variables" [4]. The watched literal infrastructure provides the services to make this happen automatically, so we need only minimal adaptations to a non-watched version of the algorithm. This is enormously much more straightforward than the complicated data structures which need to be implemented to enable classical fine-grained GAC algorithms such as AC-6, AC-7, or GAC-Schema [3,2,1] to work correctly and efficiently on backtracking. The penalty is that our version of GAC-2001/3.1 is no longer time-optimal in the worst case.

GAC Propagators for Table Constraint $\langle X_1, X_2, \ldots, X_n \rangle \in Table$

Global Variables: $tupleList, Last$

```
Setup($inputTupleList, Vars$)
A01         $tupleList = inputTupleList$
A02         foreach $v \in vars$
A02.1           foreach $i \in Dom(v)$
A02.1.1             $Last(v, i) = tupleList[0]$
A02.1.2             if SUPPORTED($Last(v, i)$)
A02.1.2.1               $Last(v, i) =$ FindNextSupportingTuple($v, i, \tau$)
A02.2               if $Last(v, i) \neq$ nil
A02.2.1                 REMOVEFROMDOMAIN($v, i$)
A02.3               else
A02.3.1                 foreach $v' \in vars$
A02.3.1.1                   ATTACHNEWTRIGGERTO($v', Last(v', i)$)
```

```
SupportingTupleLost($i, j$)
B00         Triggered by DOMAINREMOVALOF some $X_k = l$ in $Last(X_i, j)$
                                        // $X_i = j$ was supported by the tuple $Last(X_i, j)$
                                        // We must find new supporting tuple, or set $X_i \neq j$
B01         $\tau =$ FindNextSupportingTuple($i, j, Last(X_i, j)$);
B02         if $\tau \neq$ nil
B02.1       then
B02.2           for $k = 1$ to $n$
B02.2.1             MOVEWATCHFROM $Last(X_i, j)[k]$ TO $\tau[k]$
B02.3           $Last(X_i, j) = \tau$
B02.4       else                        // We failed to find a new support so $X_i \neq j$
B02.5           REMOVEFROMDOMAIN($X_i$,$j$)
```

```
FindNextSupportingTuple($i, j, \tau$)
C01         if($check = \tau + 1$; $check < sizeof(TupleList)$; $check = check + 1$)
C02.1           if(SUPPORTED($tupleList[check]$))
C02.1.1             return $tupleList[check]$
C03         if($check = 0$; $check < \tau$; $check = check + 1$)
C04.1           if(SUPPORTED($tupleList[check]$))
C04.1.1             return $tupleList[check]$
C05         return nil
```

Fig. 3. Propagator for the Table constraint. We write $\tau[k]$ for the variable-value pair at position k in the tuple.

Pseudocode for Watched-GAC is given in Figure 3. Each literal is associated with a set of triggers. These are initially attached to the literals which provide the first support for the literal that can be found. During search, if any of these literals are deleted, the trigger is activated, and either a new support is found, or if no support can be found the literal is deleted. As with earlier examples, there is no need to backtrack these supports. If a new support is found, that support

Table 4. Comparison of propagators for the table constraint in Minion on the Prime Queen Attacking problem. For size 7, the problem was not solved to optimality in reasonable time. The time given is to reach the same sub-optimal value.

	Dynamic			Watched	
Problem	Time(s)	Nodes	Nodes/s	Time(s)	Nodes/s
4	0.21	3,373	16,062	0.1	33,730
5	0.17	954	5,612	0.07	13,629
6	69.94	268,113	3,833	21.01	12,761
7	7,146.25	10,354,130	1,449	4637.11	2,233

will also be valid after backtracking. If a new support cannot be found, then we leave the triggers where they are. When search backtracks past this search node, the deleted literal will be restored, as will the literals in the old support. They must have all been present at the end of the previous node, else they would have been moved then.

This does introduce one problem when compared with GAC 2001. In GAC 2001, the tuple supporting each literal at a node is restored when backtracking to that node. This allows every tuple to be checked at most once down any branch. The watched implementation of GAC-2001 does not have this property, because the supporting tuple can change when search continues beneath a node, and it is not restored on backtracking. Therefore when checking for support, it is necessary to scan through all tuples. The current implementation always begin searching from the current support, which means that at any particular node each possible tuple will be checked at most twice, as after one pass through the tuples there must be no support, which may require one more pass through to prove. Also, this behaviour could be repeated at several nodes down a branch. We also implemented GAC for the table constraint using dynamic (i.e. non-watched) triggers. We no longer have to loop in searching for support, therefore we retain optimality down a branch, but the penalty is the overhead of storing dynamic triggers and support information in backtrackable memory.

The prime queen attacking problem (number 29 at www.csplib.org) is to put a queen and the numbers 1, ... n^2 on the cells of an an $n \times n$ chess board such that any number i is reachable via a knight's move from the cell containing $i-1$. The number of primes not attacked by the queen should be minimised (the queen does not attack its own cell). To model this problem, we use a vector V of n^2 variables, each with domain $0..(n^2-1)$ to indicate the cell to which each value is assigned. To constrain consecutive values to be placed a knight's move apart, we use a binary table constraint between each adjacent pair of elements of V. We also introduce a variable to represent the cell to which the queen is assigned, also with domain $0..(n^2-1)$. For each prime value between 2 and n^2, we introduce a 0/1 variable. A ternary table constraint ensures that this 0/1 variable is set to 1 iff the queen is attacking the corresponding value. We maximise the sum of these 0/1 variables. Experiments, shown in Table 4, for the table constraint were performed under Linux as described earlier. They show that the watched table constraint is faster than the dynamic one: i.e. the overhead of additional search are less than the overhead of restoring dynamic data structures. The limited range of experiments, and the lack of comparison against other techniques, means that we cannot draw extensive conclusions. We do conclude that watched literals provide a realistic and relatively straightforward way to implement GAC-table.

6 Implementing Watched Literals in Minion

We report briefly on our infrastructure for watched literals in Minion. This infrastructure is used by each of the propagators in this paper and is available for future propagators to be implemented. Since the intention is to use watched literals to make propagation and search faster in practical constraint solvers, it is important that implementation is done in a space and time-efficient manner. Our implementation respects two primary goals. These are that the maintenance of watched literals requires constant space after initialisation, and that key data access and update operations are fast. The key operations are finding the location of literals being watched when a value is removed, and changing which literals are being watched. We also provide infrastructure for non-stable dynamic triggers: it is almost identical except that triggers are stored in backtrackable memory.

In Minion, watched literals can be created and destroyed at any time, but each constraint has to declare at initialisation the maximum number of watched literals it will need at any one time. Each watched trigger is associated with a variable and constraint and also with a unique identifier within its constraint, while a watched literal also stores the value being watched. The constraint is responsible for maintaining any other information it needs for the trigger. For example the table constraint requires the current supporting tuple associated with a watched literal: this is stored in an array indexed by the trigger identifier.

A watched trigger consists of four values. These are: the identifier of the trigger; a pointer to the constraint associated with this trigger; and two pointers which are used to splice the trigger in and out of doubly linked lists. Once search begins, no trigger is ever moved or copied to another place in memory. Instead the pointers are used to change which list the trigger is in. Every literal in the CSP has a doubly linked list which contains the list of watches currently attached to it. Note that this requires $O(nd)$ space if we have n variables each of domain size d. Each variable also has a list for watched triggers for domain, bounds, and assignment triggers. When a literal is deleted, the solver moves through this list, triggering each constraint in turn. Thus, no work is done for these literals with no watches currently associated with them, unlike in a traditional solver where each constraint the literal is in would have to be notified. This is one of the key features behind watched literals. When a constraint moves a watched trigger `t`, in the general case we execute `t.next.prev = t.prev` and `t.prev.next = t.next`. Thus a watched trigger movement is achieved in $O(1)$, and the space used by the trigger is free to be reused for its new location, with its pointers updated accordingly. Constraints are free to move their watched literals from watching one literal to a different one. This needs to be a fast operation, so we need random access to their location in the list associated with a variable value pair. This again is achieved by a pointer.

There are further complications. (We plan to report more details at the Minion website, minion.sourceforge.net.) Constraints can leave triggers on variable-value pairs which have been removed, so we cannot just assume the trigger list will be emptied. Second, while a list of triggers is being processed, constraints may delete or move some of the triggers on it, or even move other triggers from other literals onto this list. This complicates the process of propagating all constraints attached to a literal, as the obvious methods have problems when triggers are removed from the list while the constraints are being triggered. While we raise the issue as an important implementation issue, we do not discuss our solution in detail as being of too low a level to be of general interest.

Our infrastructure for dynamic and watched triggers makes it possible to adapt MINION to add constraints dynamically, although this has not yet been implemented. A new constraint set up with dynamic triggers will automatically be retracted on backtracking past it, while a constraint using watched triggers will persist for the rest of search. This would enable techniques such as learning nogoods during search, a technique that has proved vital in SAT.

7 Further Work and Conclusions

We have demonstrated the utility of watched literals in constraint solving. In particular, we have shown how three propagators, Sum of Booleans, Element, and Table can benefit from their use. It is important to emphasise, however, that watched literals do not render classical propagation triggering mechanisms useless. Classical triggers have a lower overhead than watched literals and so are more efficient when their use is appropriate. Many are still used in Minion.

A natural and important piece of future work is to explore the integration of nogood learning into Minion. Learning is also a crucial component of a modern SAT solver, and there is every reason to believe that it can also be of great benefit to constraint solving. Minion's ability to manage large numbers of nogoods efficiently is clearly a substantial advantage in pursuing this goal.

References

1. C. Bessière and J.C. Régin. Arc consistency for general constraint networks: Preliminary results. *IJCAI*, 398–404, 1997.
2. C. Bessière. Arc-consistency and arc-consistency again. *AIJ*, 65(1):179–190, 1994.
3. C. Bessière, E. C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. *IJCAI*, 592–599, 1995.
4. C. Bessière, J.-C. Régin, R. H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *AIJ*, 165(2):165–185, 2005.
5. D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. *DAC*, 830–835. ACM, 2003.
6. N. Eén and N. Sörensson. An extensible SAT-solver. *SAT*, 502–518, 2003.
7. I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast, scalable, constraint solver. *ECAI*, 2006.

8. P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with ai and or techniques. *AAAI*, 660–664, 1988.
9. P. Van Hentenryck, V.A. Saraswat, Y. Deville: Design, implementation, and evaluation of the constraint language cc(FD). *J Log. Program.* 37(1-3): 139-164 (1998)
10. H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. *SAT*, 283–292, 2000.
11. F. Laburthe. CHOCO: implementing a CP kernel. In *Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, 2000.
12. M. Moskewicz, C. Madigan, Y. Zhao, S. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *DAC*, 2001.
13. J.-C. Régin: Maintaining Arc Consistency algorithms during the search without additional space cost. CP 2005: 520-533
14. C. Schulte and P. J. Stuckey. Speeding up constraint propagation. *CP*, 619–633, 2004.

Inner and Outer Approximations of Existentially Quantified Equality Constraints

Alexandre Goldsztejn[1] and Luc Jaulin[2]

[1] Computer Science Department, University of Central Arkansas,
Conway, Arkansas, USA
`Alexandre@Goldsztejn.com`
[2] Luc Jaulin: E3I2, ENSIETA, 2 rue F. Verny, 29806 Brest Cedex 09
`Luc.Jaulin@ensieta.fr`

Abstract. We propose a branch and prune algorithm that is able to compute inner and outer approximations of the solution set of an existentially quantified constraint where existential parameters are shared between several equations. While other techniques that handle such constraints need some preliminary formal simplification of the problem or only work on simpler special cases, our algorithm is the first pure numerical algorithm that can approximate the solution set of such constraints in the general case. Hence this new algorithm allows computing inner approximations that were out of reach until today.

1 Introduction

Many problems in computer science amount to characterizing an inner and an outer approximation of a set defined by nonlinear constraints where quantifiers may be involved. We address here the case where existential quantifiers are involved (which actually corresponds to the projection of a manifold defined by equalities). When these constraints are polynomial, symbolic methods have been shown to be able to solve the problem (see e.g., [1]). However, these techniques are restricted to very small systems. When the constraints are defined by inequalities, interval methods make it possible to characterize the solution set (see e.g., [2]). When equality constraints are involved, the problem is much more difficult and no general method seems to be available to compute an inner approximation of a set defined by nonlinear equalities. Some works were already proposed to deal with some specific subclasses of these problems: [3, 4] are restricted to linear systems, [5] is restricted to cases where the different constraints do not share any existentially quantified parameters and [6] is more general but still suffers from strong restrictions.

The paper is dedicated to the approximation of the graph of an existentially quantified constraint $c(x_1, \cdots, x_{n_x})$ defined by

$$\big(\exists y_1 \in \mathbf{y}_1\big) \cdots \big(\exists y_{n_y} \in \mathbf{y}_{n_y}\big)$$
$$\Big(f_1(x_1, \cdots, x_{n_x}, y_1, \cdots, y_{n_y}) = 0 \wedge \cdots \wedge f_m(x_1, \cdots, x_{n_x}, y_1, \cdots, y_{n_y}) = 0\Big),$$

where $\mathbf{y}_k$ for $k \in [1..n_y]$ are some bounded and nonempty intervals. Using vectorial notations, this constraint is written $c_{f,\mathbf{y}}(x)$ and defined by

$$c_{f,\mathbf{y}}(x) \iff \big(\exists y \in \mathbf{y}\big)\big(f(x,y) = 0\big), \tag{1}$$

where $x \in \mathrm{I\!R}^{n_x}$ and $y \in \mathrm{I\!R}^{n_y}$ and $\mathbf{y}$ is a bounded and nonempty box of dimension n_y and $f : \mathrm{I\!R}^{n_x} \times \mathrm{I\!R}^{n_y} \longrightarrow \mathrm{I\!R}^{m}$. The graph of $c_{f,\mathbf{y}}$ is denoted by $\Sigma(f,\mathbf{y}) := \{x \in \mathrm{I\!R}^{n_x} \mid c_{f,\mathbf{y}}(x)\}$. It is likely to have a non-zero volume if $n_y \geq m$, i.e. if there are at least as many existentially quantified variables as equations. Therefore, both inner and outer approximations are relevant. Many practical problems can be formulated as the characterization of such a set. Let us quote two of them:

- **Control.** Most dynamical systems can be described by the following state equation $\dot{x}(t) = f\big(x(t), u(t)\big) \wedge g\big(x(t),\ y(t), u(t)\big) = 0$. where the vector u is the input vector, x is the state vector and y is the output vector. The *feasible output set* $\mathbb{O}$ is the set of all $\bar{y}$ such that one can find a control u such that $g(t)$ converges to $\bar{y}$. As proved in [7], $\mathbb{O}$ satisfies

$$\mathbb{O} = \{\bar{y} \mid \exists \bar{u}, \exists \bar{x}, f(\bar{x}, \bar{u}) = 0 \wedge g(\bar{x}, \bar{y}, \bar{u}) = 0\}, \tag{2}$$

  and its characterization is therefore an instance of the problem we propose to solve.
- **Robotic.** The geometric model of a robot can often be described by the relation $f(u,x) = 0$, where u is the articulation vector and x is the coordinate vector of the tool (see e.g., [8]). For serial robots, the relation becomes $x = g(u)$ and for parallel robots, it is $u = g(x)$. However for more general robots, neither u nor x can be isolated in the relation. The workspace $\mathbb{W}$ of the robot is defined by

$$\mathbb{W} = \{x \mid \exists u \in \mathbf{u}, f(x,u) = 0\}, \tag{3}$$

  where $\mathbf{u}$ is the box of all feasible configuration vectors for the robot. Characterizing the workspace of a general robot can thus be cast into is a projection-equality problem.

2 Interval Analysis

The modern interval analysis was born in the 60's with [9]. Since, it has been widely developed and is today one central tool in the resolution of constraints acting over continuous domains (see [10] and extensive references). We now present the main concepts of interval analysis that will be used in the sequel.

Intervals are denoted by boldface symbols. The set of intervals is denoted by $\mathbb{IR}$ and contains, by convention, the empty set. The union between intervals is not an interval in general. The join between intervals (also called interval hull) is introduced to correct this bad behavior of the union. Let E be a set of intervals. The join of E, denoted by $\vee E$, is the smallest interval that contains each interval of E. When E contains only two elements, i.e. $E = \{\mathbf{x}, \mathbf{y}\}$, the join of E is also denoted by $\mathbf{x} \vee \mathbf{y}$.

The elementary functions are extended to intervals in the following way: let $\circ \in \{+, -, \times, /\}$ then $\mathbf{x} \circ \mathbf{y} = \{x \circ y \mid x \in \mathbf{x}, y \in \mathbf{y}\}$. Due to the monotony properties of these simple functions, formal expressions for the interval arithmetic are available. E.g. $[a, b] + [c, d] = [a + c, b + d]$. Also, continuous one variable functions $f(x)$ are extended to intervals using the same definition: $f(\mathbf{x}) = \{f(x) \mid x \in \mathbf{x}\}$, which is an interval because f is continuous. When one represents numbers using finite precision, the previous operations cannot be computed in general. The outer rounding is then used so as to keep valid the interpretations. For example, $[1, 2] + [2, 3]$ could be equal to $[2.999, 5.001]$ when rounded with a three decimal accuracy.

When one considers more complicated functions that are compounded of elementary functions, he will compute the interval evaluation of the function: this consists of replacing all real operations by their interval counterpart. A very basic result from interval analysis proves that the interval evaluation computes intervals that contain the range of the function. For example, $\mathbf{x} \times (\mathbf{y} - \mathbf{x}) \supseteq \{x(y - x) \mid x \in \mathbf{x}, y \in \mathbf{y}\}$.

This will be useful to use some vectorial notations. The variables $x_1, \ldots, x_n$ are denoted by the vector $x = (x_1, \ldots, x_n)$. The domains of the variables $x_1, \ldots, x_n$ are then denoted by the n-dimensional box $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, meaning that the domain of x_k is $\mathbf{x}_k$. It will also be useful to denote the vector $(x_1, \ldots, x_{n_x}, y_1, \ldots, y_{n_y})$ by (x, y) and therefore the box $(\mathbf{x}_1, \ldots, \mathbf{x}_{n_x}, \mathbf{y}_1, \ldots, \mathbf{y}_{n_y})$ by $(\mathbf{x}, \mathbf{y})$.

3 General Description of the Algorithm

In the sequel, we consider two initial boxes $\mathbf{x}^{Init} \in \mathbb{IR}^{n_x}$ and $\mathbf{y}^{Init} \in \mathbb{IR}^{n_y}$, both bounded and nonempty. The inner and outer approximations of $\Sigma(f, \mathbf{y}^{Init}) \cap \mathbf{x}^{Init}$ will be studied. The algorithm is decomposed into three phases:

1. We compute a set of boxes $\mathcal{F}$ (called *boundary Free boxes*) that are proved not to intersect the boundary of $\Sigma(f, \mathbf{y}^{Init})$ (Section 4). The remaining boxes (called *Weak boundary boxes*) are put in the list $\mathcal{W}$.
2. We classify the *boundary free boxes* into outer and inner boxes (Section 5). Unknown boxes can appear here but this is very unlikely.
3. We focus on the *weak boundary boxes* and classify them into inner boxes or unknown boxes (Section 6).

This is summarized in Algorithm 1 which is built from functions that are described in the next sections.

4 Computation of Boundary Free Boxes

The first phase consists of computing two finite sets of boxes $\mathcal{F}$ and $\mathcal{W}$ (all subsets of $\mathbf{x}^{Init}$) such that:

1. $\cup(\mathcal{F} \cup \mathcal{W}) = \mathbf{x}^{Init}$ and the boxes of $\mathcal{F} \cup \mathcal{W}$ do not overlap except perhaps over their boundaries;
2. Each box of $\mathcal{F}$ does not intersect $\mathbf{x}^{Init} \cap \partial\Sigma(f, \mathbf{y}^{Init})$ (these boxes are called *boundary free boxes*); therefore, $\mathbf{x}^{Init} \cap \partial\Sigma(f, \mathbf{y}^{Init})$ is included in $\cup\mathcal{W}$.

Algorithm 1. $\mathsf{Approximate}(f, \mathbf{x}^{Init}, \mathbf{y}^{Init}, \epsilon)$

```
   Input: $f$ (from $\mathbb{R}^{n_x} \times \mathbb{R}^{n_y}$ to $\mathbb{R}^m$), $\mathbf{x}^{Init} \in \mathbb{IR}^{n_x}$, $\mathbf{y}^{Init} \in \mathbb{IR}^{n_y}$, $\epsilon \in \mathbb{R}^+$
   Output: $(\mathcal{I}, \mathcal{O}, \mathcal{U})$ (triplet of finite sets of boxes in $\mathbb{IR}^{n_x}$)
1  $(\mathcal{F}, \mathcal{W}) = \mathsf{BoundaryFreeBoxes}(f, \mathbf{x}^{Init}, \mathbf{y}^{Init}, \epsilon)$;
2  $(\mathcal{I}', \mathcal{O}, \mathcal{W}') = \mathsf{ClassifyBoundaryFreeBoxes}(f, \mathcal{F}, \mathbf{y}^{Init}, \epsilon)$;
3  $(\mathcal{I}'', \mathcal{U}) = \mathsf{ClassifyWeakBoundaryBoxes}(f, \mathcal{W} \cup \mathcal{W}', \mathbf{y}^{Init}, \epsilon)$;
4  $\mathcal{I} = \mathcal{I}' \cup \mathcal{I}''$;
5  return $(\mathcal{I}, \mathcal{O}, \mathcal{U})$;
```

Algorithm 2. $\mathsf{BoundaryFreeBoxes}(f, \mathbf{x}^{Init}, \mathbf{y}^{Init}, \epsilon)$

```
   Input: $f$ (from $\mathbb{R}^{n_x} \times \mathbb{R}^{n_y}$ to $\mathbb{R}^m$), $\mathbf{x}^{Init} \in \mathbb{IR}^{n_x}$, $\mathbf{y}^{Init} \in \mathbb{IR}^{n_y}$, $\epsilon \in \mathbb{R}^+$
   Output: $(\mathcal{F}, \mathcal{W})$ (couple of finite sets of boxes in $\mathbb{IR}^{n_x}$)
1  $\mathcal{U} \leftarrow \mathsf{BranchAndPrune}((\mathbf{x}^{Init}, \mathbf{y}^{Init}), \mathcal{C}, \epsilon)$ where $\mathcal{C}$ is given by (6);
2  $(\mathcal{F}, \mathcal{W}) \leftarrow \mathsf{Projection}(\mathbf{x}^{Init}, \mathcal{U})$;
3  return $(\mathcal{F}, \mathcal{W})$;
```

The boxes from $\mathcal{F}$ will be classified into inner or outer boxes in Section 5 while the boxes from $\mathcal{W}$ will be classified into inner or unknown boxes in Section 6.

Algorithm 2 computes the wanted sets of boxes. The two functions that are used in Algorithm 2 are described in the rest of the section. First of all, Subsection 4.1 presents the basic test that will be used to characterize boundary free boxes. The computations performed at Line 1 are described in Subsection 4.2 while the computations performed at Line 2 are described in Subsection 4.3.

4.1 Basic Test

Our algorithm is based on the study of the relative position of boxes w.r.t. the boundary of $\Sigma(f, \mathbf{y}^{Init})$. The following theorem will play a key role in this approach.

Theorem 1. *Let $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_y} \longrightarrow \mathbb{R}^m$ be a continuously differentiable function and $\tilde{x} \in \mathbb{R}^{n_x}$ be an arbitrary vector and $\mathbf{y} \in \mathbb{IR}^{n_y}$ bounded and nonempty. For $x \in \mathbb{R}^{n_x}$ and $y \in \mathbb{R}^{n_y}$ define the matrix $M_{f,\mathbf{y}}(x, y)$ in the following way:*

$$\Big(M_{f,\mathbf{y}}(x,y)\Big)_{ij} := \begin{cases} \frac{\partial f_i}{\partial y_j}(x,y) & \text{if } \ y_j \in \operatorname{int} \mathbf{y}_j \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Then $\tilde{x} \in \partial\Sigma(f, \mathbf{y})$ implies

$$\big(\exists y \in \mathbf{y}\big)\big(f(\tilde{x}, y) = 0 \wedge \operatorname{rank} M_{f,\mathbf{y}}(\tilde{x}, y) < m\big). \tag{5}$$

Proof. Provided in Appendix A.1.

Theorem 1 is a generalization of Equation (6) in [11]. Theorem 1 is more efficient in a constraint framework: it can deal with bounded domains directly. The

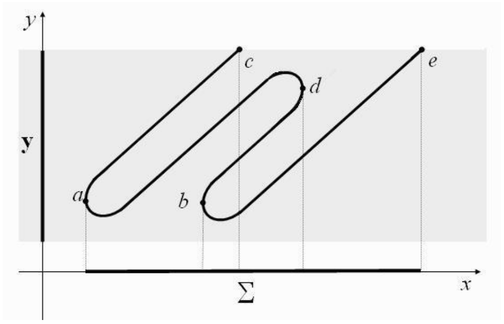


Fig. 1.

technique proposed in [11] needs a change of variables that introduces sin and cos functions, hence leading to less efficient computations.

Example 1. Consider a function $f : \mathbb{R}\times\mathbb{R} \longrightarrow \mathbb{R}$ whose implicit graph $f(x, y) = 0$ is plotted on Figure 1. The graph is restricted to $\mathbb{R} \times \mathbf{y}$ so its projection on the x-axis equals $\Sigma(f, \mathbf{y})$. We displayed the vectors a, b, c, d and e which satisfy the condition $\mathsf{rank}\, M_{f,\mathbf{y}}(x, y) < 1$: first, in the case of a vector $(x, y) \in \{a, b, d\}$, we have $\frac{\partial f}{\partial y}(x, y) = 0$ and therefore $\mathsf{rank}\, M_{f,\mathbf{y}}(x, y) = 0$. Second, in the case of a vector $(x, y) \in \{c, e\}$, the component y is on the boundary of $\mathbf{y}$ and therefore $M_{f,\mathbf{y}}(x, y)$ is set to zero by definition and finally $\mathsf{rank}\, M_{f,\mathbf{y}}(x, y) = 0$. As one can see on Figure 1, the boundary of $\Sigma(f, \mathbf{y})$ is included in the projection of $\{a, b, c, d, e\}$.

Definition 1. *With the notations introduced in Theorem 1, vectors* $(x, y) \in \mathbb{R}^{n_x} \times \mathbb{R}^{n_y}$ *that satisfy* $f(x, y) = 0$ *and* $\mathsf{rank}\, M_{f,\mathbf{y}}(x, y) < m$ *are called* singular vectors of f in $\mathbf{y}$ *(or simply singular vectors as no confusion is possible here). Then, the* weak boundary *of* $\Sigma(f, \mathbf{y})$ *is defined as the projection of the set of singular vectors into the* x*-space.*

With these definitions, Theorem 1 is simply stated saying that the boundary of $\Sigma(f, \mathbf{y})$ is included inside its weak boundary. In Example 1, the singular vectors are $\{a, b, c, d, e\}$. The projection of these vectors is the weak boundary of $\Sigma(f, \mathbf{y})$, and it actually contains $\partial\Sigma(f, \mathbf{y})$.

The computation of boundary free boxes $\mathcal{F}$ is done in two steps: a branch and prune algorithm is used to construct an outer approximation of the set of singular vectors (Subsection 4.2), and then this set is projected in order to provide a rigorous outer approximation of the weak boundary of $\Sigma(f, \mathbf{y})$ (Subsection 4.3).

4.2 Outer Approximation of the Set of Singular Vectors

An outer approximation of the set of singular vectors is computed using a basic branch and prune algorithm. This algorithm is described in Algorithm 3.

Algorithm 3. BranchAndPrune($\mathbf{u}, \mathcal{C}, \epsilon$)

```
Input: $\mathbf{u} \in \mathbb{IR}^n$ , $\mathcal{C}$ (finite set of n-ary constraints) , $\epsilon \in \mathrm{I\!R}^+$
Output: $\mathcal{U} \subseteq \mathbb{IR}^n$
1  $\mathcal{L} \leftarrow \{\mathbf{u}\}$;
2  while $\mathcal{L} \neq \emptyset$ do
3      $\tilde{\mathbf{u}} \leftarrow$ Extract($\mathcal{L}$);
4      if $||\mathrm{wid}\,\tilde{\mathbf{u}}|| \geq \epsilon$ then
5          foreach $c \in \mathcal{C}$ do
6              $\tilde{\mathbf{u}} \leftarrow$ Prune($\tilde{\mathbf{u}}, c(u)$);
7          end
8          if $\tilde{\mathbf{u}} \neq \emptyset$ then
9              $\mathcal{L} \leftarrow \mathcal{L} \cup$ Bisect($\tilde{\mathbf{u}}$);
10         end
11     else
12         $\mathcal{U} \leftarrow \mathcal{U} \cup \{\tilde{\mathbf{u}}\}$;
13     end
14 end
15 return $\mathcal{U}$;
```

The function $\mathsf{prune}\big(\tilde{\mathbf{u}}, c(u)\big)$ is often called a contractor and returns a new box $\tilde{\mathbf{u}}' \subseteq \tilde{\mathbf{u}}$ such that $\big(\forall u \in \mathbf{u}\big)\big(c(u) \Rightarrow u \in \mathbf{u}'\big)$. This algorithm is well known and $\cup\mathsf{BranchAndPrune}\big((\mathbf{x}^{Init}, \mathbf{y}^{Init}), \mathcal{C}, \epsilon\big)$, where

$$\mathcal{C} = \{\ f(x,y) = 0\ ,\ \mathsf{rank}\, M_{f,\mathbf{y}^{Init}}(x,y) < m\ \}, \tag{6}$$

is obviously an outer approximation of the set of singular vectors of f in $(\mathbf{x}^{Init}, \mathbf{y}^{Init})$.

Usually, the function $\mathsf{extract}(\mathcal{L})$ extracts the box that has the largest $||\mathrm{wid}\,\tilde{\mathbf{u}}||$. This presents the advantage that the search is performed uniformly in the search space. Extracting the box that has the smallest $||\mathrm{wid}\,\tilde{\mathbf{u}}||$ leads to a deep-first algorithm. This latter algorithm is well suited for a quick search of one approximate solution and will be used in Section 5. The function bisect must bisect fairly, meaning that each component is regularly bisected. A widely used bisection strategy is to bisect the largest component of the box, hence ensuring convergence.

Remaining is to describe the contractors that will be used for each of the two involved constraints. The contractor $\mathsf{prune}\big((\mathbf{x},\mathbf{y}), f(x,y) = 0\big)$ can be implemented using the usual constraint satisfaction techniques (cf. [10,12,13]). The contractor $\mathsf{prune}\big((\mathbf{x},\mathbf{y}), \mathsf{rank}\, M_{f,\mathbf{y}}(x,y) < m\big)$ is implemented using the interval Gauss elimination algorithm (cf. [14]). First, we need an interval evaluation of $M_{f,\mathbf{y}}(x,y)$. For $\mathbf{x}, \mathbf{y}, \tilde{\mathbf{y}} \in \mathbb{IR}^n$, $\mathbf{y} \subseteq \tilde{\mathbf{y}}$, let us define $\mathbf{M}_{f,\tilde{\mathbf{y}}}(\mathbf{x},\mathbf{y})$ in the following way:

$$\Big(\mathbf{M}_{f,\tilde{\mathbf{y}}}(\mathbf{x},\mathbf{y})\Big)_{ij} := \begin{cases} \frac{\partial \mathbf{f}_i}{\partial y_j}(\mathbf{x},\mathbf{y}) \text{ if } \ \mathbf{y}_j \subseteq \mathrm{int}\,\tilde{\mathbf{y}}_j \\ \quad 0 \qquad \text{otherwise,} \end{cases} \tag{7}$$

Algorithm 4. $\mathsf{Projection}(\mathbf{x},\mathcal{U},\epsilon)$

```
   Input: $\mathbf{x} \in \mathbb{IR}^{n_x}$, $\mathcal{U}$ (finite set of boxes in $\mathbb{IR}^{n_x} \times \mathbb{IR}^{n_y}$), $\epsilon \in \mathbb{R}^+$
   Output: $(\mathcal{F}, \mathcal{W})$ (couple of finite sets of boxes in $\mathbb{IR}^{n_x}$)
1  $\mathcal{L} \leftarrow \{\mathbf{x}\}$;
2  while $\mathcal{L} \neq \emptyset$ do
3      $\tilde{\mathbf{x}} \leftarrow \mathsf{Extract}(\mathcal{L})$;
4      if $||\mathsf{wid}\,\tilde{\mathbf{x}}|| \geq \epsilon$ then
5          if $\big(\forall(\tilde{\mathbf{x}}',\tilde{\mathbf{y}}') \in \mathcal{U}\big)\big(\tilde{\mathbf{x}} \cap \tilde{\mathbf{x}}' = \emptyset\big)$ then
6              $\mathcal{F} \leftarrow \mathcal{F} \cup \{\tilde{\mathbf{x}}\}$;
7          else
8              $\mathcal{L} \leftarrow \mathcal{L} \cup \mathsf{Bisect}(\tilde{\mathbf{x}})$;
9          end
10     else
11         $\mathcal{W} \leftarrow \mathcal{W} \cup \{\tilde{\mathbf{x}}\}$;
12     end
13 end
14 return $(\mathcal{F},\mathcal{W})$;
```

where $\frac{\partial \mathbf{f}_i}{\partial y_j}(\mathbf{x},\mathbf{y})$ is an interval evaluation of $\frac{\partial f_i}{\partial y_j}(x,y)$. With this definition, we obviously have $M_{f,\tilde{\mathbf{y}}}(x,y) \in \mathbf{M}_{f,\tilde{\mathbf{y}}}(\mathbf{x},\mathbf{y})$ for all $(x,y) \in (\mathbf{x},\mathbf{y})$. We can therefore define

$$\mathsf{prune}\big((\mathbf{x},\mathbf{y}),\mathsf{rank}\, M_{f,\tilde{\mathbf{y}}}(x,y) < m\big) := \begin{cases} \emptyset & \text{if } \mathsf{GaussElim}\big(\mathbf{M}_{f,\tilde{\mathbf{y}}}(\mathbf{x},\mathbf{y})\big) \\ (\mathbf{x},\mathbf{y}) & \text{otherwise,} \end{cases}$$

where the function $\mathsf{GaussElim}(\mathbf{M})$ returns true if and only if the interval Gauss elimination algorithm succeeds in proving that $\mathbf{M}$ has full rank.

4.3 Projection in the x-Space

Algorithm 4 computes the sets of boxes $\mathcal{F}$ (boundary free boxes) and $\mathcal{W}$ (weak boundary boxes) using the outer approximation $\mathcal{U}$ of the singular vectors computed in the previous subsection. We can display two points:

- Line 6: a box $\tilde{\mathbf{x}}$ is put in $\mathcal{F}$ only if $\big(\forall(\tilde{\mathbf{x}}',\tilde{\mathbf{y}}') \in \mathcal{U}\big)\big(\tilde{\mathbf{x}} \cap \tilde{\mathbf{x}}' = \emptyset\big)$. Because $\cup\mathcal{U}$ is an outer approximation of the set of singular vectors of $\Sigma(f,\mathbf{y}^{Init})$, this proves that $\tilde{\mathbf{x}}$ does not contain any projection of some singular vectors. Hence, $\tilde{\mathbf{x}}$ does not intersect the weak boundary of $\Sigma(f,\mathbf{y}^{Init})$ and finally does not intersect $\partial\Sigma(f,\mathbf{y}^{Init})$ neither.
- A box $\tilde{\mathbf{x}}$ is either put in $\mathcal{F}$ (Line 6) or in $\mathcal{W}$ (Line 8) or bisected (Line 11). Therefore, we have $\cup(\mathcal{F} \cup \mathcal{W}) = \tilde{\mathbf{x}}$, and hence $\partial\Sigma(f,\mathbf{y}^{Init}) \subseteq (\cup\mathcal{W})$.

The efficiency of Algorithm 4 can be drastically improved by keeping track of the tests performed at Line 5 in order to avoid useless comparisons, but the details are not presented here.

5 Classification of Boundary Free Boxes

In this section, we consider a finite set of boundary free boxes $\mathcal{F}$, i.e. boxes that are proved not to intersect $\partial\Sigma(f,\mathbf{y}^{Init})$. We aim to classify these boxes $\mathbf{x}$ into inner boxes, i.e. $\mathbf{x}\subseteq\Sigma(f,\mathbf{y}^{Init})$, and outer boxes, i.e. $\mathbf{x}\cap\Sigma(f,\mathbf{y}^{Init})=\emptyset$.

As $\mathbf{x}\in\mathcal{F}$ does not intersect the boundary of $\Sigma(f,\mathbf{y}^{Init})$, we can study a simpler problem focusing on one arbitrary vector inside the box $\mathbf{x}$. This is formalized by Proposition 1.

Proposition 1. *Let* $\mathbf{x}\in\mathbb{IR}^n$ *and* E *be a closed subset of* $\mathbb{R}^n$ *such that* $\mathbf{x}\cap\partial E=\emptyset$. *Then* $\mathbf{x}\cap E\neq\emptyset\Longrightarrow\mathbf{x}\subseteq E$, *or equivalently* $\mathbf{x}\not\subseteq E\Longrightarrow\mathbf{x}\cap E=\emptyset$.

Proof. Provided in Appendix A.2.

This proposition has two interesting consequences. First, given a box $\mathbf{x}\in\mathcal{F}$, we can now focus on one arbitrary vector inside $\mathbf{x}$. The problem will now be to decide if $\big(\exists y\in\mathbf{y}^{Init}\big)\big(f(\mathsf{mid}\,\mathbf{x},y)=0\big)$ is true or not, instead of having to decide if $\big(\forall x\in\mathbf{x}\big)\big(\exists y\in\mathbf{y}^{Init}\big)\big(f(x,y)=0\big)$ is true or not. Second, when a box $\mathbf{x}$ is proved to be inside or outside $\Sigma(f,\mathbf{y}^{Init})$, the same property holds for all boxes $\mathbf{x}'\in\mathcal{F}$ such that $\mathbf{x}\cap\mathbf{x}'\neq\emptyset$. This last remark is able to strongly accelerate the computations but it is not explicitly described in Algorithm 5.

The next proposition is an existence test that allows to check the existence of a solution to the system of equations $g(y)=0$. In our context it will be used with $g(y)=f(\mathsf{mid}\,\mathbf{x},y)$. This existence test is new, and should be compared to the usual existence tests (e.g. Moore-Kioustelidis). Proposition 2 presents two advantages over the usual existence tests: first it does not need any preconditioning. Second, it can be applied to under-constrained systems of equations. We use it here for these two reasons.

Proposition 2. *Let* $g:\mathbb{R}^n\longrightarrow\mathbb{R}^m$ *be a continuously differentiable function and* $\mathbf{y}\in\mathbb{IR}^n$ *be a bounded nonempty box. Consider a box* $\mathbf{z}$ *such that* $0\in\mathbf{z}$ *and* $g^{-1}(\mathbf{z})\cap\mathbf{y}\neq\emptyset$.[1] *Suppose that*

$$\{(y,z)\in(\mathbf{y},\mathbf{z})\mid g(y)=z\ \wedge\ \mathsf{rank}\,M_{g,\mathbf{y}}(y)<m\}=\emptyset,\tag{8}$$

where

$$\Big(M_{g,\mathbf{y}}(y)\Big)_{ij}:=\begin{cases}\frac{\partial g_i}{\partial y_j}(y) & \text{if }\ y_j\in\mathsf{int}\,\mathbf{y}_j\\ 0 & \text{otherwise.}\end{cases}$$

Then there exists $y\in\mathbf{y}$ *such that* $g(y)=0$.

Proof. Provided in Appendix A.2.

Being given a box $\mathbf{x}$ in addition to the initial box $\mathbf{y}^{Init}$, we define a set of constraints that correspond to the statement of Proposition 2:

$$\mathcal{C}=\{\ f(\mathsf{mid}\,\mathbf{x},y)=z\ ,\ \mathsf{rank}\,M_{f,\mathbf{y}^{Init}}(y)<m\ \}.\tag{9}$$

[1] The box $\mathbf{z}:=0\vee g(\tilde{y})$, where $\tilde{y}\in\mathbf{y}$ is an approximate solution of $g(y)=0$, is both efficient and easy to compute.

```
Algorithm 5. ClassifyBoundaryFreeBoxes($f, \mathcal{F}, \mathbf{y}, \epsilon$)
        Input: $f$ (from $\mathbb{R}^{n_x} \times \mathbb{R}^{n_y}$ to $\mathbb{R}^m$), $\mathcal{F}$ (finite sets of boxes in
               $\mathbb{IR}^{n_x}$), $\mathbf{y} \in \mathbb{IR}^{n_y}$, $\epsilon \in \mathbb{R}^+$
        Output: ($\mathcal{I}$, $\mathcal{O}$,$\mathcal{U}$) (triplet of finite sets of boxes in $\mathbb{IR}^{n_x}$)
      1 while $\mathcal{F} \neq \emptyset$ do
      2     $\mathbf{x} \leftarrow$ Extract($\mathcal{F}$);
 Com. 3     $\mathcal{L} \leftarrow$ BranchAndPrune( $\mathbf{y}$ , $\{f(\text{mid}\,\mathbf{x}, y) = 0\}$ , $\epsilon$);
 Com. 4     if $\mathcal{L} = \emptyset$ then
      5         $\mathcal{O} = \mathcal{O} \cup \{\mathbf{x}\}$;
      6     else
      7         $\mathbf{y}$ = Extract($\mathcal{L}$);
 Com. 8         $\mathbf{z} \leftarrow 0 \vee f(\text{mid}\,\mathbf{x}, \text{mid}\,\mathbf{y})$;
      9         $\mathcal{L}' \leftarrow$ BranchAndPrune( $(\mathbf{y}, \mathbf{z})$ , $\mathcal{C}$ , $\epsilon$);
     10         if $\mathcal{L}' = \emptyset$ then
     11             $\mathcal{I} = \mathcal{I} \cup \{\mathbf{x}\}$;
     12         else
     13             $\mathcal{U} = \mathcal{U} \cup \{\mathbf{x}\}$;
     14         end
     15     end
     16 end
     17 return ($\mathcal{I}, \mathcal{O}, \mathcal{U}$);
```

We can now propose Algorithm 5 that classifies boundary free boxes into inner boxes and outer boxes (and possibly unknown boxes). Lines preceded by "Com." are commented bellow:

- Line 3: the branch and prune algorithm must either prove the emptiness or provide one approximate solution; therefore, it is modified to a deep-first search algorithm.
- Line 4: if $\mathcal{L}$ is empty then the branch and prune algorithm has proved $\text{mid}\,\mathbf{x} \notin \Sigma(f, \mathbf{y}^{Init})$. Because $\mathbf{x}$ is supposed to be a boundary free box, Proposition 1 then proves that $\mathbf{x} \cap \Sigma(f, \mathbf{y}^{Init}) = \emptyset$.
- Line 8: $f(\text{mid}\,\mathbf{x}, \text{mid}\,\mathbf{y})$ is computed using interval arithmetic and therefore leads to an interval that rigorously contains the image of $(\text{mid}\,\mathbf{x}, \text{mid}\,\mathbf{y})$. The interval vector $\mathbf{z}$ is the join (or interval hull) of the latter interval vector and 0, and it therefore contains both 0 and the image of $(\text{mid}\,\mathbf{x}, \text{mid}\,\mathbf{y})$. Hence, $\mathbf{z}$ is a good interval vector to use in Proposition 2.

6 Classification of Weak Boundary Boxes

We consider a box $\mathbf{x}$ that was not proved to be a boundary free box. The idea is to consider a stronger problem $\Sigma(f, \mathbf{y}^{Init\prime})$ with $\mathbf{y}^{Init\prime} \subseteq \mathbf{y}^{Init}$. Then, the weak boundary of $\Sigma(f, \mathbf{y}^{Init\prime})$ is certainly different from the one of $\Sigma(f, \mathbf{y}^{Init})$, and the box $\mathbf{x}$ is hopefully a boundary free box for the new problem. So, we will use the algorithms presented in the previous sections to handle the new problem $\Sigma(f, \mathbf{y}^{Init\prime})$.

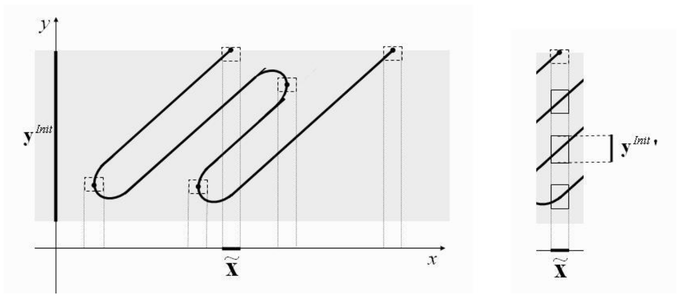


Fig. 2.

Let us illustrate this technique with an example. Consider the quantified constraint represented on Figure 1. The left-hand side graphic of Figure 2 displays the unknown boxes generated at Line 1 of Algorithm 2. These boxes contain the singular vectors of $\Sigma(f, \mathbf{y}^{Init})$. The projection of these boxes forms the weak boundary. The right-hand side graphic focuses on one box $\tilde{\mathbf{x}}$ among the weak boundary box. The four boxes of the right hand side graphic are obtained using the branch and prune algorithm to prune the constraint $f(x, y) = 0$ with $x \in \tilde{\mathbf{x}}$ and $y \in \mathbf{y}^{Init}$. We can now easily pick up a box $\mathbf{y}^{Init\prime}$ where no singularity occurs, and the algorithms presented in the previous section are likely to succeed in proving that it is an inner box.

In practice, however, the difference between singular boxes and nonsingular boxes is not as clearly identified as on Figure 2. Although finding an efficient heuristic to compute a new initial box $\mathbf{y}^{Init\prime}$ is one important forthcoming work, the experimentations presented in the next section show that this simple heuristic is already useful.

The process described in this section leads to a function

$$\mathsf{ClassifyWeakBoundaryBoxes}(f, \mathcal{W}, \mathbf{y}^{Init}, \epsilon)$$

that returns a couple $(\mathcal{I}, \mathcal{U})$. The set of boxes $\mathcal{I}$ contains the boxes that where proved to be inside $\Sigma(f, \mathbf{y}^{Init\prime}) \subseteq \Sigma(f, \mathbf{y})$. The set of boxes $\mathcal{U}$ contains the boxes we were not able to prove anything about.

7 Experimentations

The application of the algorithm to three examples is now presented. No comparison is provided as our algorithm is the first numerical algorithm that is able to compute the inner approximations proposed in this section. Formal quantifier elimination (cf. [1]) can be applied to the first two examples. The results obtained by our algorithm are similar to the one obtained by formal quantifier elimination.

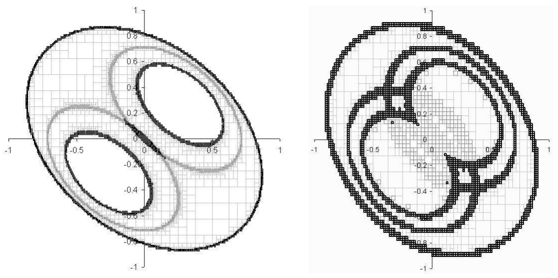


Fig. 3.

7.1 Well-Constrained Academic Problem

The first problem is defined by

$$f(x,y)=\begin{pmatrix} x_1^2+x_2^2+y_1^2+y_2^2-1 \\ x_1+x_2+y_1+y_2 \end{pmatrix} \; ; \; \mathbf{x}^{Init}=\begin{pmatrix}[-1,1]\\ [-1,1]\end{pmatrix} \; ; \; \mathbf{y}^{Init}=\begin{pmatrix}[-0.7,0.7]\\ [-0.8,0.8]\end{pmatrix}.$$

The pavings plotted in the left hand side graphic of Figure 3 are obtained after one minute using a precision $\epsilon = 0.01$. Inner boxes are in gray (light gray for boundary free boxes and dark gray for weak boundary boxes that have been proved to be inner boxes) and unknown boxes are in black. The algorithm behaves very well with this example and provides a good inner approximation.

7.2 Under-Constrained Academic Problem

The second problem is defined by

$$f(x,y)=\begin{pmatrix} x_1^2+x_2^2+y_1^2+y_2^2+y_3^2-1 \\ x_1+x_2+y_1+y_2+y_3 \end{pmatrix} \; ; \; \mathbf{x}^{Init}=\begin{pmatrix}[-1,1]\\ [-1,1]\end{pmatrix} \; ; \; \mathbf{y}^{Init}=\begin{pmatrix}[-0.7,0.7]\\ [-0.8,0.8]\\ [-2,2]\end{pmatrix}.$$

The pavings plotted in right hand side graphic of Figure 3 are obtained after 15 minutes using a precision $\epsilon = 0.02$. The heuristic presented in Section 6 for the classification of weak boundary boxes is clearly not efficient for this example: it is missing some parts of the weak boundary while 99% of the computations are spent working on the weak boundary. This will have to be investigated; however, the algorithm presents a good behavior for the computation and classification of boundary free boxes.

7.3 Speed Diagram of a Sailboat

The third problem was proposed in [5]. It is defined by

$$f(v,\theta,\delta_r,\delta_s)=\begin{pmatrix} \alpha_s(V\cos(\theta+\delta_s)-v\sin\delta_s)\sin\delta_s-\alpha_r v\sin^2\delta_r-\alpha_f v \\ (l-r_s\cos\delta_s)\alpha_s(V\cos(\theta+\delta_s)-v\sin\delta_s)-r_r\alpha_r v\sin\delta_r\cos\delta_r \end{pmatrix},$$

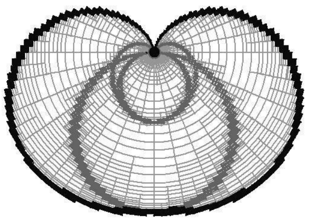


Fig. 4.

where the following values are chosen for parameters: $\alpha_s = 500$, $\alpha_r = 300$, $\alpha_f = 60$, $V = 10$, $r_s = 1$, $r_r = 2$, and $l = 1$. The initial domains are $\mathsf{Dom}(v) = [0, 20]$, $\mathsf{Dom}(\theta) = [-\pi, \pi]$, $\mathsf{Dom}(\delta_r) = [-\pi/2, \pi/2]$ and $\mathsf{Dom}(\delta_r) = [-\pi/2, \pi/2]$. The graph to be approximated is therefore

$$\Big\{(v, \theta) \in ([0, 20], [-\pi, \pi]) \mid (\exists \delta_r \in [-\frac{\pi}{2}, \frac{\pi}{2}])\,(\exists \delta_s \in [-\frac{\pi}{2}, \frac{\pi}{2}])\,(f(v, \theta, \delta_r, \delta_s) = 0)\Big\}.$$

This set corresponds to the speed v and angle θ w.r.t. the wind that can be reached for some command δ_r and δ_s in their domains. An inner approximation was computed in [5] after a specific formal simplification of the problem. The pavings plotted in Figure 4 are obtained after 10 minutes using a precision $\epsilon = 0.01$. We obtain the same results as in [5] but without any preliminary formal simplification. Our algorithm is slower, but it works in a 4 dimensional space (while the simplification used in [5] decreases the dimension by one), and it was not yet optimized.

8 Conclusion

We have presented the first numerical algorithm that is able to compute an inner approximation (and obviously an outer approximation) of the graph of an existentially quantified constraint with an arbitrary number of equality constraints. Although some previous works were dedicated to the inner approximation of such constraints in some special cases, the algorithm we proposed can be applied for arbitrary numbers of equalities and existentially quantified parameters.

The idea consisting of using a branch and prune algorithm to approximate the boundary of the constraint graph instead of the constraint graph itself seems to be new. Not only does it allow simplification of the problem to be solved, but it should also make the algorithm accumulate on this boundary and therefore lead to efficient computations. Timings on presented examples are reasonable but not yet good. However, no optimization has been done in order to present the concepts clearly. We expect some strong efficiency improvements in the next implementations of the algorithm. Finally, one advantage of the proposed method is that it relies only on a simple standard branch and prune algorithm. Therefore, any future improvement for pruning operators will be useful for our algorithm.

Convergence of the algorithm remains to be studied. This convergence strongly depends on the heuristic used to deal with weak boundary boxes. Actually, the simple heuristic proposed in Section 6 showed is usefulness but can be certainly improved. In particular, experimentations showed it was not very efficient for under-constrained problems. Therefore, a heuristic that makes the algorithm convergent will have to be found.

References

1. Collins, G.: Quantifier elimination by cylindrical algebraic decomposition–twenty years of progress. In Quantifier Elimination and Cylindrical Algebraic Decomposition (1998) 8–23
2. Ratschan, S.: Uncertainty propagation in heterogeneous algebras for approximate quantified constraint solving. Journal of Universal Computer Science **6**(9) (2000) 861–880
3. Shary, S.: A new technique in systems analysis under interval uncertainty and ambiguity. Reliable computing **8** (2002) 321–418
4. Goldsztejn, A.: A Right-Preconditioning Process for the Formal-Algebraic Approach to Inner and Outer Estimation of AE-solution Sets. Reliable Computing **11**(6) (2005) 443–478
5. Herrero, P., Jaulin, L., Vehi, J., Sainz, M.: Inner and outer approximation of the polar diagram of a sailboat. Interval analysis, constraint propagation, applications, Held in conjunction with the Eleventh International Conference on Principles and Practice of Constraint Programming (CP 2005), Sitges, Spain (2005)
6. Goldsztejn, A.: A branch and prune algorithm for the approximation of non-linear ae-solution sets. In: Proceedings of the 21st ACM Symposium on Applied Computing track Reliable Computations and their Applications, Dijon, France, April 2006 (SAC 2006). (2006)
7. Khalil, H.: Nonlinear Systems, Third Edition. Prentice Hall (2002)
8. Reboulet, C.: Modélisation des robots parallèles. In Boissonat, J.D., Faverjon, B., Merlet, J.P., eds.: Techniques de la robotique, architecture et commande. Herms, Paris, France (1988) 257–284
9. Moore, R.: Interval analysis. Prentice-Hall (1966)
10. Benhamou, F., Older, W.: Applying Interval Arithmetic to Real, Integer and Boolean Constraints. Journal of Logic Programming **32**(1) (1997) 1–24
11. Haug, E., Luh, C., Adkins, F., Wang, J.: Numerical algorithms for mapping boundaries of manipulator workspaces. ASME Journal of Mechanical Design **118** (1996) 228–234
12. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics. Springer-Verlag (2001)
13. Lebbah, Y., Michel, C., Rueher, M., Daney, D., Merlet, J.: Efficient and Safe Global Constraints for handling Numerical Constraint Systems. SIAM Journal on Numerical Analysis **42**(5) (2005) 2076–2097
14. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge Univ. Press, Cambridge (1990)
15. Goldsztejn, A., Jaulin, L.: Inner Approximation of the Range of Vector-Valued Functions. (Submitted to Reliable Computing)

A Proofs

A.1 Proof of Theorem 1

Lemma 1. *Let $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_y} \longrightarrow \mathbb{R}^m$ be a continuous function and $\mathbf{y} \in \mathbb{IR}^{n_y}$ be bounded and nonempty. Then $\Sigma(f, \mathbf{y})$ is closed in $\mathbb{R}^{n_x}$.*

Proof. Consider a sequence $x_n \in \Sigma(f, \mathbf{y})$ that converges to $\tilde{x}$. We have to prove $\tilde{x} \in \Sigma(f, \mathbf{y})$. As $x_n \in \Sigma(f, \mathbf{y})$, there exists $y_n \in \mathbf{y}$ such that $f(x_n, y_n) = 0$. As $\mathbf{y}$ is bounded, the Bolzano-Weierstrass theorem proves that the sequence y_n has at least one accumulation point in $\mathbf{y}$. Let us denote one of these accumulation points $\tilde{y}$. We can pick up a subsequence $y_{\pi(n)}$ that converges to $\tilde{y}$. The sequence $x_{\pi(n)}$ obviously converges to $\tilde{x}$. Therefore, $\lim_{n\to\infty} f(x_{\pi(n)}, y_{\pi(n)}) = f(\tilde{x}, \tilde{y}) = 0$. We proved that there exists $\tilde{y} \in \mathbf{y}$ such that $f(\tilde{x}, \tilde{y}) = 0$ and hence $\tilde{x} \in \Sigma(f, \mathbf{y})$. □

Lemma 2. *Let $\mathbf{y} \in \mathbb{IR}^n$. If $y \in \partial\mathbf{y}$ then there exists $i \in [1..n]$ such that $y_i \in \partial\mathbf{y}_i$.*

Proof. We prove the contrapose. Suppose for all $i \in [1..n]$ we have $y_i \in \mathsf{int}\,\mathbf{y}_i$. That is, for all $i \in [1..n]$ we have $\inf \mathbf{y}_i < y_i < \sup \mathbf{y}_i$. This proves $y \in \mathsf{int}\,\mathbf{y}$. □

Proof of Theorem 1. We prove the contrapositive of the statement. There are only two possible cases: $\big(\forall y \in \mathbf{y}\big)\big(f(\tilde{x}, y) \neq 0\big)$ or $\big(\exists y \in \mathbf{y}\big)\big(f(\tilde{x}, y) = 0\big)$. Let us consider both cases. In the first case, we have $\tilde{x} \notin \Sigma(f, \mathbf{y})$, and because $\Sigma(f, \mathbf{y})$ is closed in $\mathbb{R}^{n_x}$ by Lemma 1, we have $\tilde{x} \notin \partial\Sigma(f, \mathbf{y})$. The second case relies on the implicit function theorem. Consider $\tilde{y} \in \mathbf{y}$ such that $f(\tilde{x}, \tilde{y}) = 0$. By hypothesis, we have $\mathsf{rank}\, M_{f,\mathbf{y}}(\tilde{x}, \tilde{y}) = m$ (which implies $n_y \geq m$). Therefore, there exists a set of indices $\mathcal{E} := \{e_1, \cdots, e_m\}$ such that $\det M \neq 0$ where $M \in \mathbb{R}^{m\times m}$ is defined by $M_{ij} := \big(M_{f,\mathbf{y}}(\tilde{x}, \tilde{y})\big)_{ie_j}$.

Claim: $\tilde{y}_{\mathcal{E}} \in \mathsf{int}\,\mathbf{y}_{\mathcal{E}}$. The claim is proved by contradiction. Suppose that $\tilde{y}_{\mathcal{E}} \in \partial\mathbf{y}_{\mathcal{E}}$, therefore by Lemma 2 there exists $e_j \in \mathcal{E}$ such that $\tilde{y}_{e_j} \in \partial\mathbf{y}_{e_j}$. Then by definition of $M_{f,\mathbf{y}}(\tilde{x}, \tilde{y})$ we have $M_{ij} = 0$ for all $i \in [1..n]$. Therefore M is singular which is absurd because $\det M \neq 0$ by hypothesis.

Now define $g : \mathbb{R}^{n_x} \times \mathbb{R}^m \longrightarrow \mathbb{R}^m$ by $g(x, y_{\mathcal{E}}) = f(x, y)$ where $y_{[1..n]\setminus\mathcal{E}}$ is fixed to $\tilde{y}_{[1..n]\setminus\mathcal{E}}$. With this definition, we have $\frac{\partial g_i}{\partial y_{e_j}}(\tilde{x}, \tilde{y}_{\mathcal{E}}) = M_{ij}$. As M is nonsingular, we can apply the implicit function theorem that proves the existence of

(i) some open sets $\mathbb{X} \subseteq \mathbb{R}^{n_x}$ and $\mathbb{Y} \subseteq \mathbb{R}^m$ that contain respectively $\tilde{x}$ and $\tilde{y}_{\mathcal{E}}$;
(ii) a continuously differentiable function $\phi : \mathbb{X} \longrightarrow \mathbb{Y}$ such that $\phi(\tilde{x}) = \tilde{y}_{\mathcal{E}}$;
(iii) $y_{\mathcal{E}} = \phi(x)$ implies $g(x, y_{\mathcal{E}}) = 0$ for any $x \in \mathbb{X}$.

Now define $\mathbb{Y}' = \mathbb{Y} \cap (\mathsf{int}\,\mathbf{y}_{\mathcal{E}})$ which is open because it is the intersection of two open sets. As $\tilde{y}_{\mathcal{E}} \in \mathsf{int}\,\mathbf{y}_{\mathcal{E}}$ and $\tilde{y}_{\mathcal{E}} \in \mathbb{Y}$ we have $\tilde{y}_{\mathcal{E}} \in \mathbb{Y}'$. As ϕ is continuous the preimage $\mathbb{X}' := \phi^{-1}(\mathbb{Y}')$ is also open. Furthermore $\tilde{x} \in \mathbb{X}'$ because $\tilde{y}_{\mathcal{E}} \in \mathbb{Y}'$ and $\phi(\tilde{x}) = \tilde{y}_{\mathcal{E}}$.

For all $x \in \mathbb{X}'$ define $y \in \mathbf{y}$ by $y_{\mathcal{E}} := \phi(x)$ and $y_{[1..n]\setminus\mathcal{E}} = \tilde{y}_{[1..n]\setminus\mathcal{E}}$. Using (iii), we have $g(x, y_{\mathcal{E}}) = 0$ which implies $f(x, y) = 0$ by definition of g. We have therefore proved $\mathbb{X}' \subseteq \Sigma(f, \mathbf{y})$ which eventually proves that $\tilde{x} \in \mathsf{int}\,\Sigma(f, \mathbf{y})$. Therefore $\tilde{x} \notin \partial\Sigma(f, \mathbf{y})$.

A.2 Proofs of Proposition 1 and Proposition 2

Lemma 3. *Let E be closed in $\mathbb{R}^n$ and $x \in \operatorname{int} E$ and $x' \notin E$. Any continuous path connecting x to x' intersects ∂E.*

Proof. Cf. [15]. □

Proof of Proposition 1. It is sufficient to prove that the box $\mathbf{x}$ is either inside or outside of E. This is proved by contradiction: let us suppose that $\mathbf{x}$ is neither inside nor outside, i.e. there exist $x, x' \in \mathbf{x}$ such that $x \in E$ and $x' \notin E$. As $\mathbf{x}$ does not intersect ∂E, we have $x \in \operatorname{int} E$. Furthermore, $\mathbf{x}$ being path-connected, there exists a path w that is contained in $\mathbf{x}$ and which connects x and x'. Applying Lemma 3, we prove that w intersects ∂E, and therefore that $\mathbf{x}$ intersects ∂E, which is eventually absurd because we supposed $\mathbf{x} \cap \partial E = \emptyset$.

Proof of Proposition 2. We define $h(z, y) := g(y) - z$ and by definition we have $\Sigma(h, \mathbf{y}) = \big\{z \in \mathbb{R}^m \mid (\exists y \in \mathbf{y})\big(h(z, y) = 0\big)\big\}$. We will prove that $\mathbf{z} \subseteq \Sigma(h, \mathbf{y})$, which will conclude the proof because by hypothesis $0 \in \mathbf{z}$ and by definition of h, $h(0, y) = 0 \implies g(y) = 0$. As $g^{-1}(\mathbf{z}) \cap \mathbf{y} \neq \emptyset$, there exists $\tilde{y} \in \mathbf{y}$ such that $g(\tilde{y}) \in \mathbf{z}$. We have $g(\tilde{y}) \in \Sigma(h, \mathbf{y})$ because $h(g(\tilde{y}), \tilde{y}) = 0$ by definition of h. As $g(\tilde{y}) \in \mathbf{z}$ by hypothesis, we have $\mathbf{z} \cap \Sigma(h, \mathbf{y}) \neq \emptyset$. Therefore, thanks to Proposition 1, we just have to prove that $\mathbf{z} \cap \partial\Sigma(h, \mathbf{y}) = \emptyset$. By definition of h and $M_{g,\mathbf{y}}(y)$ and $M_{h,\mathbf{y}}(z, y)$, the condition (8) is equivalent to $\{(y, z) \in (\mathbf{y}, \mathbf{z}) \mid h(z, y) = 0 \ \wedge\ \operatorname{rank} M_{h,\mathbf{y}}(z, y) < m\} = \emptyset$. As a direct consequence we obtain

$$\big\{z \in \mathbf{z} \mid (\exists y \in \mathbf{y})\big(h(z, y) = 0 \ \wedge\ \operatorname{rank} M_{h,\mathbf{y}}(z, y) < m\big)\big\} = \emptyset.$$

Finally, this condition validates the hypothesis of Theorem 1 which proves that $\mathbf{z} \cap \partial\Sigma(h, \mathbf{y}) = \emptyset$, hence concluding the proof.

Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms

Frank Hutter[1], Youssef Hamadi[2], Holger H. Hoos[1], and Kevin Leyton-Brown[1]

[1] University of British Columbia, 2366 Main Mall, Vancouver BC, V6T1Z4, Canada
{hutter, kevinlb, hoos}@cs.ubc.ca
[2] Microsoft Research, 7 JJ Thomson Ave, Cambridge, UK
youssefh@microsoft.com

Abstract. Machine learning can be used to build models that predict the run-time of search algorithms for hard combinatorial problems. Such *empirical hardness models* have previously been studied for complete, deterministic search algorithms. In this work, we demonstrate that such models can also make surprisingly accurate predictions of the run-time distributions of incomplete and randomized search methods, such as stochastic local search algorithms. We also show for the first time how information about an algorithm's parameter settings can be incorporated into a model, and how such models can be used to automatically adjust the algorithm's parameters on a per-instance basis in order to optimize its performance. Empirical results for Novelty$^+$ and SAPS on structured and unstructured SAT instances show very good predictive performance and significant speedups of our automatically determined parameter settings when compared to the default and best fixed distribution-specific parameter settings.

1 Introduction

The last decade has seen a dramatic rise in our ability to solve combinatorial optimization problems in many practical applications. High-performance heuristic algorithms increasingly exploit problem instance structure. Thus, knowledge about the relationship between this structure and algorithm behavior forms an important basis for the development and successful application of such algorithms. This has inspired a large amount of research on methods for extracting and acting upon such information. These range from search space analysis to automated algorithm selection and tuning methods.

An increasing number of studies explore the use of machine learning techniques in this context [15,18,6,8]. One recent approach uses linear basis function regression to obtain models of the time an algorithm will require to solve a given problem instance [19,21]. These so-called *empirical hardness models* can be used to obtain insights into the factors responsible for an algorithm's performance, or to induce distributions of problem instances that are challenging for a given algorithm. They can also be leveraged to select among several different algorithms for solving a given problem instance.

In this paper, we extend on this work in three significant ways. First, past work on empirical hardness models has focused exclusively on complete, deterministic algorithms [19,21]. Our first goal is to show that the same methods can be used to predict sufficient statistics of the run-time distributions (RTDs) of incomplete, randomized algorithms, and in particular of stochastic local search (SLS) algorithms for SAT. This is important because SLS algorithms are among the best existing techniques for solving a wide range of hard combinatorial problems, including hard subclasses of SAT [14].

The behavior of many randomized heuristic algorithms is controlled by parameters with continuous or large discrete domains. This holds in particular for most state-of-the-art SLS algorithms. For example, the performance of WalkSAT algorithms such as Novelty [20] or Novelty$^+$ [12] depends critically on the setting of a noise parameter whose optimal value is known to depend on the given SAT instance [13]. Understanding the relationship between parameter settings and the run-time behavior of an algorithm is of substantial interest for both scientific and pragmatic reasons, as it can expose weaknesses of a given search algorithm and help to avoid the detrimental impact of poor parameter settings. Thus, our second goal is to extend empirical hardness models to include algorithm parameters in addition to features of the given problem instance.

Finally, hardness models could also be used to automatically determine good parameter settings. Thus, an algorithm's performance could be optimized for each problem instance without any human intervention or significant overhead. Our final goal is to explore the potential of such an approach for automatic per-instance parameter tuning.

In what follows, we show that we have achieved all three of our goals by reporting the results of experiments with SLS algorithms for SAT. (We note however, that our approach is by no means limited to SLS algorithms or SAT, though the features we use were created with some domain knowledge. In experimental work it is obviously necessary to choose *some* specific domain. We have chosen to study the SAT problem because it is the prototypical and best-studied $\mathcal{NP}$-complete problem and there exists a great variety of SAT benchmark instances and solvers.) Specifically, we considered two high-performance SLS algorithms for SAT, Novelty$^+$ [12] and SAPS [17], and several widely-studied structured and unstructured instance distributions. In Section 2, we show how to build models that predict the sufficient statistics of RTDs for randomized algorithms. Empirical results demonstrate that we can predict the median run-time for our test distributions with surprising accuracy (we achieve correlation coefficients between predicted and actual run-time of up to 0.995), and that based on this statistic we can also predict the complete exponential RTDs Novelty$^+$ and SAPS exhibit. Section 3 describes how empirical hardness models can be extended to incorporate algorithm parameters; empirical results still demonstrate good performance for this harder task (correlation coefficients reach up to 0.98). Section 4 shows that these models can be leveraged to perform automatic per-instance parameter tuning that results in significant reductions of the algorithm's run-time compared to using default settings (speedups of up to two orders of magnitude) or even the best fixed parameter values for the given instance distribution (speedups of up to an order of magnitude). Section 5 describes how Bayesian techniques can be leveraged when predicting run-time for test distributions that differ from the one used for training of the empirical hardness model. Finally, Section 6 concludes the paper and points out future work.

2 Run-Time Prediction: Randomized Algorithms

Previous work [19,21] has shown that it is possible to predict the run-time of deterministic tree-search algorithms for combinatorial problems using supervised machine learning techniques. In this section, we demonstrate that similar techniques are able to predict the run-time of algorithms which are both randomized and incomplete. We support our arguments by presenting the results of experiments involving two powerful local search algorithms for SAT.

2.1 Prediction of Sufficient Statistics for Run-Time Distributions

It has been shown in the literature that high-performance randomized local search algorithms tend to exhibit exponential run-time distributions [14], meaning that the run-times of two runs that differ only in their random seeds can easily vary by an order of magnitude. Even more extreme variability in run-time has been observed for randomized complete search algorithms [11]. Due to this inherent algorithm randomness, we have to predict a probability distribution over the amount of time an algorithm will take to solve the problem. For many randomized algorithms such *run-time distributions* closely resemble standard parametric distributions such as exponential or Weibull (see, e.g., [14]). These parametric distributions are completely specified by certain sufficient statistics. For example, an exponential distribution can be specified by its median. It follows that by predicting such sufficient statistics, a prediction for the entire run-time distribution for an unseen instance is obtained.

Note that for randomized algorithms, the error in a model's predictions can be divided into two components: the extent to which the model fails to describe the data, and the inherent noise in the employed summary statistics due to randomness of the algorithm. This latter component may be reduced by measuring the statistics over a larger number of runs per instance. As we will see in Figures 1(a) and 1(b), while empirical hardness models of SLS algorithms are able to predict the run-times of single runs reasonably well, their predictions of median run-times over a larger set of runs are much more accurate.

Our approach for run-time prediction of randomized incomplete algorithms largely follows the basis function regression approach of [19,21].[1] While an extension of our work to randomized tree search algorithms should be straight-forward, experiments in this paper are restricted to incomplete local search algorithms.

In order to predict the run-time of an algorithm $\mathcal{A}$ on a distribution $\mathcal{D}$ of instances, we draw an i.i.d. sample of N instances from $\mathcal{D}$. For each instance s_n in this training set, $\mathcal{A}$ is run some constant number of times and an empirical fit r_n of the sufficient statistics of interest is recorded. Note that r_n is a $1 \times M$ vector if there are M sufficient statistics of interest. We also compute a set of $k = 43$ instance features $\boldsymbol{x}_n = [x_{n,1}, \ldots, x_{n,k}]$ for each instance. This set is a subset of the features used in [21], including basic statistics, graph-based features, local search probes, and DPLL-based measures.[2] We restricted

[1] In previous preliminary and unpublished experiments for the winner determination problem, we examined other techniques such as support vector machine regression, multivariate adaptive regression splines and lasso regression; none improved predictive performance significantly. More recent experiments (see Section 5) suggest that Gaussian process regression *can* increase performance, especially when the amount of training data is small. However, this method has complexity cubic in the number of *data points*, complicating its practical use.

[2] Information on precisely which features we used, as well as the rest of our experimental data and Matlab code, is available online at `http://www.cs.ubc.ca/labs/beta/Projects/Empirical-Hardness-Models/`

the subset of features because some features from [21] timed out for large instances—the computation of all our 43 features took only about 2 seconds per instance.

Given this data for all the training instances, a function $f(\boldsymbol{x})$ is fitted that, from the features $\boldsymbol{x}_n$ of an instance s_n, approximates the sufficient statistics r_n of $\mathcal{A}$'s run-time distribution on this instance. Since linear functions of these raw features may not be expressive enough, we construct a richer set of basis functions which can include arbitrarily complex functions of *all* features $\boldsymbol{x}_n$ of an instance s_n, or simply the raw features themselves. These basis functions typically contain a number of elements which are either unpredictive or highly correlated. Predictive performance can thus be improved (especially in terms of robustness) by applying some form of feature selection that identifies a small subset of D important features; as explained later, here we use forward selection with a designated validation set to select up to $D = 40$ features. We denote the reduced set of D basis functions for instance s_n as $\boldsymbol{\phi}_n = \boldsymbol{\phi}(\boldsymbol{x}_n) = [\phi_1(\boldsymbol{x}_n), \ldots, \phi_D(\boldsymbol{x}_n)]$. We then use ridge regression to fit the $D \times M$ matrix of free parameters $\boldsymbol{w}$ of a linear function $f_{\boldsymbol{w}}(\boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_n)^\top \boldsymbol{w}$, that is, we compute $\boldsymbol{w} = (\delta I + \boldsymbol{\Phi}^\top \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top \boldsymbol{r}$, where δ is a small regularization constant (set to 10^{-2} in our experiments), $\boldsymbol{\Phi}$ is the $N \times D$ design matrix $\boldsymbol{\Phi} = [\boldsymbol{\phi}_1^\top, \ldots, \boldsymbol{\phi}_N^\top]^\top$, and $\boldsymbol{r} = [{r_1}^\top, \ldots, {r_N}^\top]^\top$. Given a new, unseen instance s_{N+1}, a prediction of the M sufficient statistics can be obtained by computing the instance features $\boldsymbol{x}_{N+1}$ and evaluating $f_{\boldsymbol{w}}(\boldsymbol{x}_{N+1}) = \boldsymbol{\phi}(\boldsymbol{x}_{N+1})^\top \boldsymbol{w}$. One advantage of the simplicity of ridge regression is a low computational complexity of $\Theta(\max\{D^3, D^2N, DNM\})$ for training and of $\Theta(DM)$ for prediction for an unseen test instance.

2.2 Experimental Setup and Empirical Results for Predicting Median Run-Time

We performed experiments for the prediction of run-time distributions for two SLS algorithms, SAPS and Novelty$^+$. Because previous studies [12,17,14] have shown that these algorithms tend to have approximately exponential run-time distributions, the sufficient statistics r_n for each instance s_n reduce to the empirical median run-time of a fixed number of runs. In this section we fix SAPS parameters to their default values $\langle \alpha, \rho, P_{smooth} \rangle = \langle 1.3, 0.8, 0.05 \rangle$. For Novelty$^+$, we use its default parameter setting $\langle noise, wp \rangle = \langle 0.5, 0.01 \rangle$ for unstructured instances. On structured instances Novelty$^+$ is known to perform better with lower noise settings, and indeed with noise=0.5 the majority of runs did not finish within an hour of CPU time. Thus, we chose $\langle noise, wp \rangle = \langle 0.1, 0.01 \rangle$ which solved all structured instances in 15 minutes of CPU time. We consider models that incorporate multiple parameter settings in the next section.

In our experiments, we used six widely-studied SAT benchmark distributions, half consisting of unstructured instances and half of structured instances. The first two distributions we studied each consisted of 20,000 uniform-random 3-SAT instances with 400 variables; the first (**CV-var**) varied the clauses-to-variables ratio between 3.26 and 5.26, while the second (**CV-fixed**) fixed $c/v = 4.26$. These distributions were previously studied in [21], facilitating a comparison of our results with past work. Our third unstructured distribution (**SAT04**) consisted of 3,000 random unstructured instances generated with the two generators used for the 2004 SAT solver competition (with identical parameters) and was employed to evaluate our automated parameter tuning procedure on a competition benchmark.

Table 1. Evaluation of learned models on test data. N is the number of instances for which the algorithm's median runtime is $\leq$ 900 CPU seconds (only those instances are used and split 50:25:25 into training, validation, and test sets). Columns for correlation coefficient and RMSE indicate values using only raw features as basis functions, and then using raw features and their pairwise products. SAPS was always run with its default parameter settings $\langle \alpha, \rho \rangle = \langle 1.3, 0.8 \rangle$. For Novelty$^+$, we used noise=0.5 for unstructured and noise=0.1 for structured instances.

Unstructured instances					
Dataset	N	Algorithm	Runs	Corrcoeff	RMSE
CV-var	9952	SAPS	1	0.903/0.911	0.37/0.35
CV-var	9952	SAPS	10	0.960/0.968	0.23/0.20
CV-var	9952	SAPS	100	0.967/0.977	0.21/0.17
CV-var	9952	SAPS	1000	0.968/0.978	0.20/0.17
CV-var	9952	Novelty$^+$	10	0.947/0.952	0.25/0.23
CV-fixed	10125	SAPS	10	0.765/0.781	0.46/0.44
CV-fixed	10125	Novelty$^+$	10	0.586/0.603	0.61/0.60
SAT04	1457	SAPS	10	0.933/0.938	0.52/0.50
SAT04	1426	Novelty$^+$	10	0.934/0.938	0.58/0.56

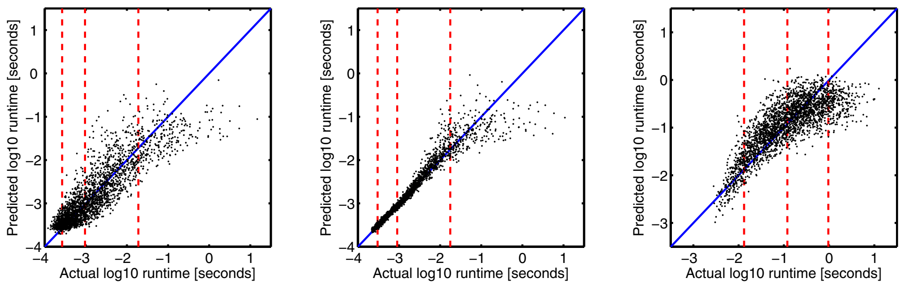
Structured instances					
Dataset	N	Algorithm	Runs	Corrcoeff	RMSE
QWH	7793	SAPS	10	0.988/0.995	0.33/0.21
QWH	8049	Novelty$^+$	10	0.988/0.992	0.22/0.18
QCP	14716	SAPS	10	0.995/0.997	0.17/0.15
QCP	15263	Novelty$^+$	10	0.993/0.994	0.12/0.11
SW-GCP	4287	SAPS	10	0.890/0.892	0.45/0.45
SW-GCP	5573	Novelty$^+$	10	0.690/0.691	0.23/0.23

Our first two structured distributions are different variants of quasigroup completion problems. The first one (**QCP**) consisted of 30,626 quasigroup completion instances, while the second one (**QWH**) contained 9,601 instances of the quasigroup completion problem for quasigroups with randomly punched holes) [10]. Both distributions were created with the generator `lsencode` by Carla Gomes. The ratio of unassigned cells varied from 25% to 75%. We chose quasigroup completion problems as a representative of structured problems because this domain allows the systematic study of a large instance set with a wide spread in hardness, and because the structure of the underlying Latin squares is similar to the one found in applications such as scheduling, time-tabling, experimental design, and error correcting codes [10]. Our last structured distribution (**SW-GCP**) contained 20,000 instances of graph coloring based on small world graphs that were created with the generator `sw.lsp` by Toby Walsh [9].

As is standard in the study of SLS algorithms, all distributions were filtered to contain only satisfiable instances, leading to 9,952, 10,125, 1,470, 17,989, 9,601, and 11,182 instances for CV-var, CV-fixed, SAT04, QCP, QWH, and SW-GCP, respectively. To limit computational time we only used instances that were solved in a single SAPS run of one hour. This further reduced the sets to 9,952, 10,125, 1,469, 15,263, 8,049, and 5,573 instances for CV-var, CV-fixed, SAT04, QCP, QWH, and SW-GCP, respectively.

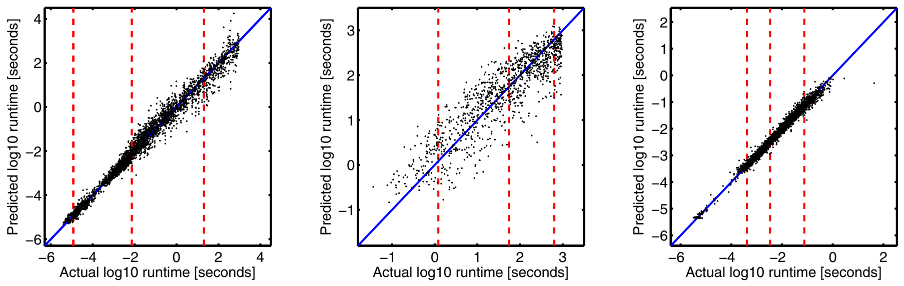
We then randomly split each instance set 50:25:25 into training, validation, and test sets; all experimental results are based on the test set and were stable with respect to reshuffling. We chose the 43 raw features and the constant 1 as our basis functions, and also included pairwise multiplicative combinations of all raw features. We then performed forward selection to select up to 40 features, stopping when the error on the validation set first began to grow. Experiments were run on a cluster of 50 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running SuSE Linux 9.1.

Overall, our experiments show that we can consistently predict median run-time with surprising accuracy. Results for all our benchmark distributions are summarized in Table 1. Note that a correlation coefficient (CC) of 1 indicates perfect prediction while 0 indicates random noise; a root mean squared error (RMSE) of 0 means perfect prediction



(a) 1 Novelty$^+$ run on CV-var. CC=0.878, RMSE=0.37 (b) 100 Novelty$^+$ runs on CV-var. CC=0.962, RMSE=0.21 (c) 100 SAPS runs on CV-fixed. CC=0.800, RMSE=0.42

Fig. 1. Correlation between observed and predicted run-times/medians of run-times of SAPS and Novelty$^+$ on unstructured instances. The basis functions were raw features and their pairwise products. The three red vertical dashed lines in these and all other scatter plots in this paper denote the 10%, 50%, and 90% quantiles of the data. For example, this means that 40% of the data points lie between the left and the middle vertical lines.



(a) 10 SAPS runs on QWH (b) 10 SAPS runs on SW-GCP (c) 10 Novelty$^+$ runs on QCP

Fig. 2. Correlation between observed and predicted run-times/medians of run-times of SAPS and Novelty$^+$ on SAT04 and QWH. The basis functions were raw features and their pairwise products. For RMSEs and correlations coefficients, see Table 1.

while 1 roughly means average misprediction by one order of magnitude. Also note that the predictive qualities for Novelty$^+$ and SAPS are qualitatively similar.

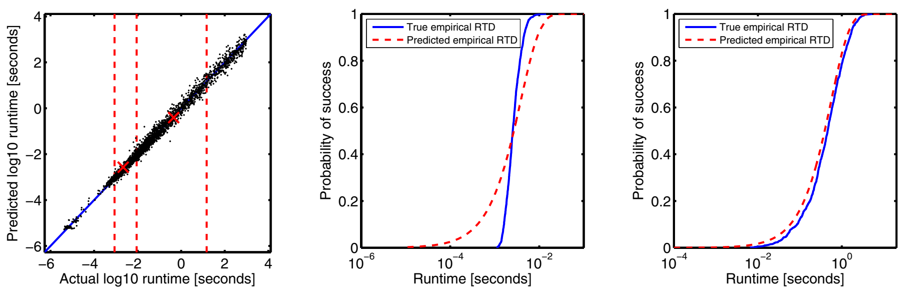
Figure 1(a) shows a scatterplot of predicted vs. actual run-time for Novelty$^+$ on CV-var, where the model is trained and evaluated on a single run per instance. Most of the data points are located in the very left of this plot, which we visualize by plotting the 10%, 50% and 90% quantiles of the data (the three red dashed lines). While a strong trend is evident in Figure 1(a), there is significant error in the predictions. Figure 1(b) shows the same algorithm on the same dataset, but now predicting the median of an empirical run-time distribution based on 100 runs. The error for the leftmost 90% of the data points is substantially reduced, leading to an almost halved RMSE when compared to predictions for a single run. It is also noteworthy that these run-time predictions are more accurate than the predictions for the deterministic algorithms kcnfs, satz, and ok-solver (compare against Figure 5(left) in [21]). While this is already true for predictions based on single runs it is much more pronounced when predicting median run-time. This same effect holds true for predicting median run-time of SAPS, and for different distributions. Figure 1(c) also shows much better predictions than we observed for deterministic tree search algorithms on CV-fix (compare this plot against Figure 7(left) in [21]). We believe that two factors contribute to this effect. First, we see deterministic algorithms as comparable to randomized algorithms with a fixed seed. Obviously, the single run-time of such an algorithm on a particular instance is less informative about its underlying run-time distribution (were it randomized) than the sufficient statistics of multiple runs. Second, one of the main reasons to introduce randomness in search is to achieve diversification. This allows the heuristic to recover from making a bad decision by exploring a new part of the search space, and hence reduces the variance of run-times across very similar instances. Because deterministic solvers do not include such diversification mechanisms, they can exhibit strikingly different run-times on very similar instances. (This observation is the basis of the literature on heavy-tailed run-time distributions in complete search, see e.g. [11].) For example, consider modifying a SAT instance by randomly shuffling the names of its variables. One would expect a properly randomized algorithm to have essentially the same run-time distributions for both instances; however, a deterministic solver could exhibit very different runtimes on the two instances [5]. Because empirical hardness models must give similar predictions for instances with similar feature values, the model for the deterministic solver could be expected to exhibit higher error in this case.

Figure 2 visualizes our predictive quality for structured data sets. Performance for both QWH and QCP, as shown in Figures 2(a) and 2(c), was very good with correlation coefficients between predicted and actual median run-time of up to 0.995. Note, however, that the hardest instance in Figure 2(c) was predicted to be much easier than it actually is. This is because the instance was exceptionally hard: over an order of magnitude harder than the hardest instance in the training set. The last structured data set, SW-GCP, is the hardest distribution for prediction we have encountered thus far (unpublished data shows RMSEs of around 1.0 when predicting the run-time of deterministic algorithms on SW-GCP). As shown in Figure 2(b), the predictions for SAPS are surprisingly good; predictive quality for Novelty$^+$ (see Table 1) is also much higher than what we have seen for deterministic algorithms.

We now look at which features are most important to our models; this is not straightforward since the features are highly correlated. Following [19,21], we build subset models of increasing size until the RMSE and correlation coefficient are comparable to the ones for the full model with 40 basis functions. Table 2 reports the results for SAPS on CV-fix and Novelty$^+$ on QCP and for each of these also gives the performance of the best model with a single basis function. Overall, we observe that the most important features for predicting run-time distributions of our SLS algorithms are the same ones that were observed to be important for predicting run-times of deterministic algorithms in [21]. Also similar to observations from [21], we found that very few features are needed to build run-time models of instances that are all satisfiable. While [21] studied only uniform-random data, we found in our experiments that this is true for both unstructured

Table 2. Feature importance in small subset models for predicting median run-time of 10 runs. The cost of omission for a feature specifies how much worse validation set predictions are without it, normalized to 100 for the top feature. The RMSE and Corrcoeff columns compare predictive quality on the test set to that of full 40-feature models.

#	Basis function	Cost of omission	Corrcoeff	RMSE
	SAPS on CV-fix			
1.	saps_BestSolution_CoeffVariance × saps_BestStep_CoeffVariance	100	0.744/0.785	0.47/0.44
1.	saps_BestSolution_CoeffVariance × saps_AvgImproveToBest_Mean	100	0.758/0.785	0.46/0.44
2.	saps_BestStep_CoeffVariance × saps_FirstLMRatio_Mean	45		
3.	gsat_BestSolution_CoeffVariance × lobjois_mean_depth_over_vars	37		
4.	saps_AvgImproveToBest_CoeffVariance	15		
5.	saps_BestCV_Mean × gsat_BestStep_Mean	11		
	Novelty$^+$ on QCP			
1.	VG_mean × gsat_BestStep_Mean	100	0.966/0.994	0.29/0.11
1.	saps_AvgImproveToBest_CoeffVariance × gsat_BestSolution_Mean	100	0.991/0.994	0.13/0.11
2.	vars_clauses_ratio × lobjois_mean_depth_over_vars	68		
3.	VG_mean × gsat_BestStep_Mean	12		
4.	TRINARY_PLUS × lobjois_log_num_nodes_over_vars	7		



(a) Predictions of median run-time, instances $q_{0.25}$ and $q_{0.75}$ (b) Easy QCP instance ($q_{0.25}$) (c) Hard QCP instance ($q_{0.75}$)

Fig. 3. Predicted versus actual empirical RTDs for SAPS on two QCP instances. 10 runs were used for learning median run-time and in (a), 1000 runs for the empirical RTDs in (b) and (c).

and structured instances and for both algorithms we studied. Small models for CV-var (both for SAPS and Novelty$^+$) almost exclusively use local search features (almost all of them based on short SAPS trajectories). The structured domain QCP employs a mix of local search probes (based on both SAPS and GSAT), constraint-graph-based features (e.g., VG_mean) and in the case of Novelty$^+$ also some DPLL-based features, such as the estimate of the search tree size (lobjois_mean_depth_over_vars). In some cases (e.g., models of SAPS on CV-var), and when we record relatively few runs per instance, a single feature can be sufficient for predicting single run-times with virtually the same accuracy as the full model.

To illustrate that based on the median we can fairly accurately predict entire run-time distributions for the SLS algorithms studied here, we show the predicted and empirically measured RTDs for SAPS on two QCP instances in Figure 3. The two instances correspond to the 0.25 and 0.75 quantiles of the distribution of actual median hardness for SAPS on the entire QCP instance set; they correspond to the red crosses in Figure 3(a),

Table 3. Parameter configurations employed in our experiments

Algorithm	Fixed parameters	Default parameters	Used parameter configurations
Novelty$^+$	$wp = 0.01$	$noise = 0.5\%$	$noise \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$
SAPS	$P_{smooth} = 0.05$, $wp = 0.01$	$\langle \alpha, \rho \rangle = \langle 1.3, 0.8 \rangle$	All combinations of $\alpha \in \{1.2, 1.3, 1.4\}$ and $\rho \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$

which shows the tight correlation between actual and predicted run-times. Consistent with previous results by Hoos et al. (see, e.g., Chapters 4 and 6 of [14]), the RTD for the $q_{0.75}$ instance is closely approximated by an exponential distribution, which our approach almost perfectly fits (see Figure 3(c)). The RTDs for easier instances are known to typically exhibit smaller variance; therefore, an approximation with an exponential distribution is less accurate (see Figure 3(b)). We plan to predict sufficient statistics for the more general distributions needed to characterize such RTDs, such as Weibull and generalized exponential distributions, in the future.

3 Run-Time Prediction: Parametric Algorithms

The behavior of most high-performance SLS algorithms is controlled by one or more parameters. It is well known that these parameters often have a substantial effect on the algorithm's performance (see, e.g., [14]). In the previous section, we showed that quite accurate empirical hardness models can be constructed when these parameters are held constant. In practice, however, we also want to be able to model an algorithm's behavior when these parameter values are changed. In this section, we demonstrate that it is possible to incorporate parameters into empirical hardness models for randomized, incomplete algorithms. Our techniques should also carry over to both deterministic and complete parametric algorithms (in the case of deterministic algorithms using single run-times instead of sufficient statistics of RTDs).

Our approach is to learn a function $g(\boldsymbol{x}, c)$ that takes as inputs both the features $\boldsymbol{x}_n$ of an instance s_n and the parameter configuration c of an algorithm $\mathcal{A}$, and that approximates sufficient statistics of $\mathcal{A}$'s RTD when run on instance s_n with parameter configuration c. In the training phase, for each training instance s_n we run $\mathcal{A}$ some constant number of times with a set of parameter configurations $\boldsymbol{c}_n = \{c_{n,1}, \ldots, c_{n,k_n}\}$, and collect fits of the sufficient statistics $\boldsymbol{r}_n = [r_{n,1}^\top, \ldots, r_{n,k_n}^\top]^\top$ of the corresponding empirical run-time distributions. We also compute s_n's features $\boldsymbol{x}_n$. The key change from the approach in Section 2.1 is that now the parameters that were used to generate an $\langle$instance,run-time$\rangle$ pair are effectively treated as additional features of that training example. We define a new set of basis functions $\boldsymbol{\phi}(\boldsymbol{x}_n, c_{n,j}) = [\phi_1(\boldsymbol{x}_n, c_{n,j}), \ldots, \phi_D(\boldsymbol{x}_n, c_{n,j})]$ whose domain now consists of the cross product of features and parameter configurations. For each instance s_n and parameter configuration $c_{n,j}$, we will have a row in the design matrix $\boldsymbol{\Phi}$ that contains $\boldsymbol{\phi}(\boldsymbol{x}_n, c_{n,j})^\top$—that is, the design matrix now contains k_n rows for training instance s_n. The target vector $\boldsymbol{r} = [\boldsymbol{r}_1^\top, \ldots, \boldsymbol{r}_N^\top]^\top$ just stacks all the sufficient statistics on top of each other. We learn the function $g_{\boldsymbol{w}}(\boldsymbol{x}, c) = \boldsymbol{\phi}(\boldsymbol{x}, c)^\top \boldsymbol{w}$ by applying ridge regression as in Section 2.1.

Our experiments in this section concentrate on predicting median run-time of SAPS since that is the more challenging problem. SAPS has three interdependent, continuous

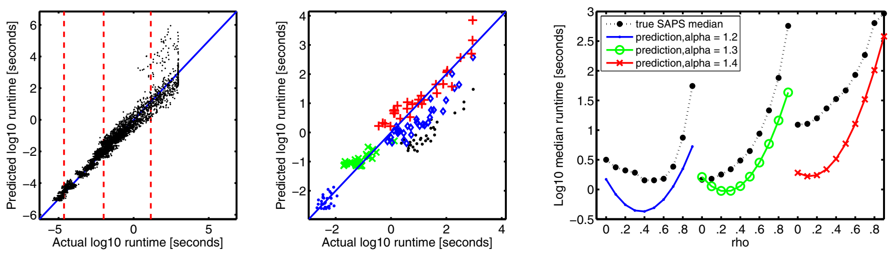


Fig. 4. Left: Predictions for SAPS on QWH with 30 parameter settings. Middle: Data points for 5 instances from SAPS on SAT04, different symbol for each instance. Right: Predicted run-time vs. median SAPS run-time over 1000 runs for 30 parameter settings on the median SAT04 instance, the one marked with blue diamonds in the middle figure.

parameters, as compared to Novelty$^+$ which has only one interesting parameter. (Both algorithms have an additional parameter, wp, which is typically set to a default value that results in uniformly good performance.) This difference notwithstanding, we observed qualitatively similar results with Novelty$^+$. Note that the approach outlined above allows one to use different parameter settings for each training instance. How to pick these settings for training in the most informative way is an interesting experimental design question which invites the use of active learning techniques; we plan to tackle it in future work. In this study, we used the parameter combinations defined in Table 3. We fixed $P_{smooth} = 0.05$ for SAPS since its effect is highly correlated with that of ρ.

As basis functions, we used multiplicative combinations of the raw instance features $\boldsymbol{x}_n$ and a 2nd-order expansion of all non-fixed (continuous) parameter settings. For K raw features ($K = 43$ in our experiments), this meant $3K$ basis functions for Novelty$^+$, and $6K$ for SAPS, respectively. As before we applied forward selection to select up to 40 features, stopping when the error on the validation set first began to grow. For each data set reported here, we randomly picked 1000 instances to be split 50:50 for training and validation. We ran one run per instance and parameter configuration yielding 30,000 data points for SAPS and 6,000 for Novelty$^+$. (Training on the median of more runs would likely have improved the results.) For the test set, we used an additional 100 distinct instances and computed the median of 10 runs for each parameter setting.

In Figure 4(left), we show predicted vs. actual SAPS run-time for the QWH dataset and the 30 $\langle \alpha, \rho \rangle$ combinations in Table 3. This may be compared to Figure 2(a), which shows the same algorithm on the same dataset for fixed parameter values. (Note, however, that Figure 2(a) was trained on more runs and using more powerful basis functions for the instance features.) We observe that our model still achieves good performance, yielding correlation coefficient/RMSE of 0.98/0.41, as compared to 0.988/0.33 for the fixed-parameter setting (using raw features as basis functions).

Figure 4(middle) shows predicted vs. actual SAPS median run-time for five instances from SAT04, namely the easiest and hardest instance, and the 0.25, 0.5, and 0.75 quantiles. Runs corresponding to the same instance are plotted using the same symbol. Note that run-time variation due to the instance is often greater than variation due to parame-

Table 4. Results for automated parameter tuning. For each instance set and algorithm, we give the correlation between actual and predicted run-time for all instances, RMSE, the correlation for all data points of an instance (mean ± stddev), and the best fixed a posteriori parameter setting on the test set. We also give the average speedup over the best possible parameter setting per instance (s_{bpi}, always ≤ 1), over the worst possible setting per instance (s_{wpi}, always ≥ 1), the default (s_{def}), and the best fixed setting. For example, on Mixed, Novelty$^+$ is on average 9.52 times faster than its best data-set specific fixed parameter setting (s_{fixed}). All experiments use second order expansions of the parameters (combined with the instance features). Bold face indicates speedups of the automated parameter setting over the default and best fixed parameter settings.

Set	Algo	Gross corr	RMSE	Corr per inst.	best fixed a posteriori	s_{bpi}	s_{wpi}	s_{def}	s_{fixed}
SAT04	Nov	0.90	0.76	0.84 ± 0.29	0.5	0.65	193.19	0.88	0.88
QWH	Nov	0.98	0.52	0.76 ± 0.43	0.1	0.85	683.04	**257.96**	0.94
Mixed	Nov	0.95	0.77	0.80 ± 0.35	0.2	0.71	350.12	**14.49**	**9.52**
SAT04	SAPS	0.91	0.60	0.63 ± 0.29	$\langle 1.3, 0 \rangle$	0.43	15.95	**2.66**	0.96
QWH	SAPS	0.98	0.41	0.43 ± 0.39	$\langle 1.2, 0 \rangle$	0.67	5.88	**2.39**	**1.02**
Mixed	SAPS	0.95	0.61	0.47 ± 0.38	$\langle 1.3, 0 \rangle$	0.48	8.53	**2.22**	0.97

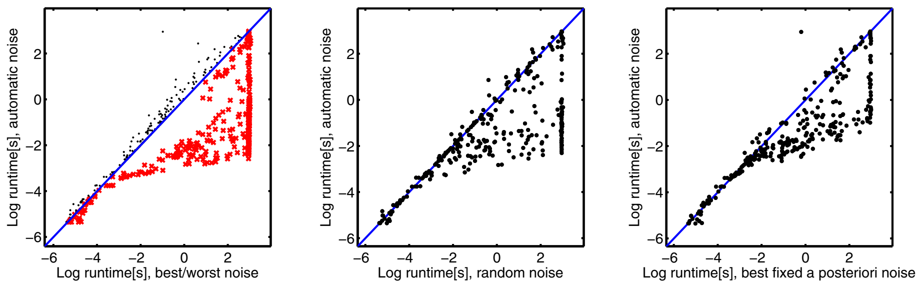
ter settings. However, harder instances tend to be more sensitive to variation in the algorithm's parameters than easier ones – this indicates the importance of parameter tuning, especially for hard instances. The average correlation coefficient for the 30 points per instance is 0.63; for the 6 points per instance in Novelty$^+$ it is 0.84, much higher.

Figure 4(right) shows SAPS run-time predictions for the median instance of our SAT04 test set at each of its 30 $\langle \alpha, \rho \rangle$ combinations; these are compared to the actual median SAPS run-times on this instance. We observe that the learned model predicts the actual run-times fairly well, despite the fact that the relationship between run-time and the two parameters is complex. In the experiment the figure is based on feature selection chose 40 features; thus, the model learned a 40-dimensional surface and the figure shows that its projection onto the 2-dimensional parameter space at the current instance features qualitatively captures the shape of the actual parameter-dependent run-time for this instance.

4 Automated Parameter Tuning

Our results, as suggested by Figure 4 indicate that our methods are able to predict per-instance and per-parameter run-times with reasonable accuracy. We can thus hope that they would also be able to predict which parameter settings result in the lowest run-time for a given instance. This would allow us to use a learned model to automatically tune the parameter values of an SLS algorithm on a per-instance basis by simply picking the parameter configuration out of the ones we consider (see Table 3) that is predicted to yield the lowest run-time. Note that our approach for parameter tuning is orthogonal to that of reactive search approaches such as Adaptive Novelty$^+$ [13] and RSAPS [17].

In this section we investigate this approach. We now focus on the Novelty$^+$ algorithm, because we observed SAPS's performance around $\langle \alpha, \rho \rangle = \langle 1.3, 0.1 \rangle$ to be very



(a) Automatic vs. best & worst (b) Automatic vs. random (c) Automatic vs. best fixed

Fig. 5. (a) Performance of automated parameter setting for Novelty$^+$ on data set Mixed, compared to the best (dots) and worst (crosses) per-instance parameter setting (out of the 6 parameter settings we employed). (b) Compared to independent random noise values for each instance. (c) Speedup of Novelty$^+$ over the best fixed a posteriori parameter setting for Mixed.

close to optimal across many different instance distributions.[3] SAPS thus offers little *possibility* for performance improvement through per-instance parameter tuning (Table 4 quantifies this). Novelty$^+$, on the other hand, exhibits substantial variation in the best parameter setting from one instance distribution to another, making it a good algorithm for the evaluation of our approach.[4] We used the same test and training data as in the previous section; thus, Table 4 summarizes the experiments both from the previous section and from this section. However, in this section we also created a new instance distribution "Mixed", which is the union of the QWH and SAT04 distributions. This mix enables a large gain for automated parameter tuning (when compared to the best fixed parameter setting) since Novelty$^+$ performs best with high noise settings on unstructured instances and low settings on structured instances.

Figure 5(a) shows the performance of our automatic parameter-tuning algorithm on test data from Mixed, as compared to upper and lower bounds on its possible performance. We observe that the run-time with automatic parameter setting is close to the optimal setting and far better than the worst one, with an increasing margin for harder instances. Figure 5(b) provides a comparison of our method against a uniform random picking of parameter combinations from the six considered Novelty$^+$ configurations (see Table 3). Figure 5(c) compares our automatic tuning against the best fixed parameter setting (this was determined in an a posteriori fashion as the setting with the best performance on the test set out of the ones we considered, see Table 3). This setting is often the best that can be hoped for in practice. (A common approach for tuning parameters is to perform a set of experiments, to identify the parameter setting which achieves the lowest overall run-time,

[3] This is true even though it has been demonstrated that for each SAPS parameter there exist instances for which a statistically significant improvement can be obtained over the default setting $\langle\alpha, \rho\rangle = \langle 1.3, 0.8\rangle$ (defined in [17]) by tuning that parameter [25]. We note that the setting $\langle\alpha, \rho\rangle = \langle 1.3, 0.1\rangle$ differs from the default studied by [25], raising the question of whether the cited result would also hold for this setting.

[4] Indeed, the large potential gains for tuning WalkSAT's noise parameter on a per-instance basis have been exploited before [22].

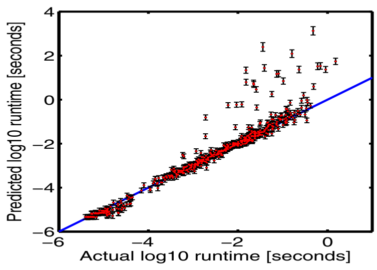
and then to fix the parameters to this setting.) Figure 5(c), in conjunction with Table 4, shows that our techniques can dramatically outperform this form of parameter tuning: Novelty$^+$ almost achieves an average speedup of an order of magnitude on Mixed as compared to the best fixed parameter setting on that set. SAPS improves upon its default setting by more than a factor of two for all three distributions. Considering that our method is fully automatic and very general, these are very promising results.

Related Work on Automated Parameter Tuning

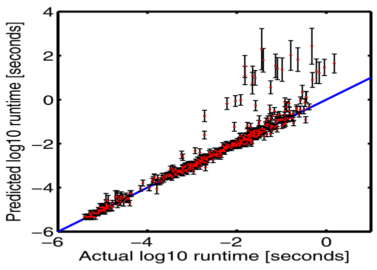
The task of configuring an algorithm's parameters for high and robust performance has been widely recognized as a tedious and time-consuming task that requires well-developed engineering skills. Automating this task is a very promising and active area of research. There exists a large number of approaches to find the best configuration for a given problem distribution [3,24,1]. All these techniques aim to find a parameter setting that optimizes some scoring function which averages over all instances from the given input distribution. If the instances are sufficiently homogeneous, this approach can perform quite well. However, if the problem instances to be solved come from heterogeneous distributions or even from completely unrelated application areas, the best parameter configuration may differ vastly from instance to instance. In such cases it is advisable to apply an approach like ours that can choose the best parameter setting for each run contingent on the characteristics of the current instance to be solved. This per-instance parameter tuning is more powerful but less general than tuning on a per-distribution basis in that it requires the existence of a set of discriminative instance features. However, we believe it to be not too difficult to engineer a good set of instance features if one is familiar with the general problem domain.

The only other approach for parameter tuning on a per-instance basis we are aware of is the Auto-WalkSAT framework [22]. This approach is based on empirical findings showing that the optimal parameter setting of WalkSAT algorithms tends to be about 0.1 above the one that minimizes the invariance ratio [20]. Auto-WalkSAT chooses remarkably good noise settings on a variety of instances, but for domains where the above relationship between invariance ratio and optimal noise setting does not hold (such as logistics problems), it performs poorly [22]. Furthermore, its approach is limited to SAT and in particular to tuning the (single) noise parameter of the WalkSAT framework. In contrast, our automated parameter tuning approach applies to arbitrary parametric algorithms and all domains for which good features can be engineered.

Finally, reactive search algorithms [2], such as Adaptive Novelty$^+$[13] or RSAPS [17] adaptively modify their search strategy *during* a search. (Complete reactive search algorithms include [18,6].) Many reactive approaches still have one or more parameters whose settings remain fixed throughout the search; in these cases the automated configuration techniques we presented here should be applicable to tune these parameters. While a reactive approach is in principle more powerful than ours (it can utilize different search strategies in different parts of the space), it is also less general since the implementation is typically tightly coupled to a specific algorithm. Ultimately, we aim to generalize our approach to allow for modifying parameters during the search—this requires that the features evaluated during search are very cheap to compute. We also see reinforcement learning as very promising in this context [18].



(a) Bayesian linear regression



(b) GP with squared exponential kernel

Fig. 6. Predictions and their uncertainty of Novelty^{+} median run-time of 10 runs: Bayesian linear regression and Gaussian Process with squared exponential kernel, trained on QWH and tested on QCP. The run-time predictions of these approaches are Gaussian probability distributions for every instance. The red dots specify the predictive mean and the black bars the standard deviation.

5 Uncertainty Estimates Through Bayesian Regression

So far, research in empirical hardness models has focused on the case where the targeted application domain is known *a priori* and training instances from this domain are available. In practice, however, an algorithm may have to solve problem instances that are significantly different from the ones encountered during training. Empirical hardness models may perform poorly in such cases. This is because the statistical foundations upon which their machine learning approach is built rely upon the test set being drawn from the same distribution as the training set. Bayesian approaches may be more appropriate in such scenarios since they explicitly model the *uncertainty* associated with their predictions. Roughly, they provide an automatic measure of how similar the basis functions for a particular test instance are to those for the training instances, and associate higher uncertainty with relatively dissimilar instances. We implemented two Bayesian methods: (a) sequential Bayesian linear regression (BLR) [4], a technique which yields mean predictions equivalent to ridge regression but also offers estimates of uncertainty; and (b) Gaussian Process Regression (GPR) [23] with a squared exponential kernel. We detail BLR and the potential applications of a Bayesian approach to run-time prediction in an accompanying technical report [16]. Since GPR scales cubically in the number of data points, we trained it on a subset of 1000 data points (but used all 9601 data points for BLR). Even so, GPR took roughly 1000 times longer to train.

We evaluated both our methods on two different problems. The first problem is to train and validate on our QWH data-set and test on our QCP data-set. While these distributions are not identical, our intuition was that they share enough structure to allow models trained on one to make good predictions on the other. The second problem was much harder: we trained on data-set SAT04 and tested on a very diverse test set containing instances from ten qualitatively different distributions from SATLIB. Figure 6 shows predictions and their uncertainty ($\pm$ one stddev) for both methods on the first problem. The two distributions are similar enough to yield very good predictions for both approaches. While BLR was overconfident on this data set, the uncertainty estimates of GPR make more sense: they are very small for accurately predicted data points and large for mispredicted ones. Both models achieved similar predictive accuracy (CC/RMSE 0.953/0.50 for BLR; 0.953/0.51 for GPR). For the second problem (space restrictions prevent a figure), BLR showed massive mispredictions (several tens of orders of magnitude) but associated very high uncertainty with the mispredicted instances, reflecting their dissimilarity with the training set. GPR showed more reasonable predictions, and also did a good job in indicating high uncertainty about instances for which predictive quality was low. Based on these preliminary results, we view Gaussian process regression as particularly promising and plan to study its application to run-time prediction in more detail. However, we note that its scaling behavior somewhat limits its usefulness in practice.

6 Conclusion and Future Work

In this work, we have demonstrated that empirical hardness models obtained from linear basis function regression can be extended to make surprisingly accurate predictions of the run-time of randomized, incomplete algorithms such as Novelty$^+$ and SAPS. Based on a prediction of sufficient statistics for run-time distributions (RTDs), we showed very good predictions of the entire empirical RTDs for unseen test instances. We have also demonstrated for the first time that empirical hardness models can model the effect of algorithm parameter settings on run-time, and that these models can be used as a basis for automated per-instance parameter tuning. In our experiments, this tuning never hurt and sometimes resulted in substantial and completely automatic performance improvements, as compared to default or optimized fixed parameter settings.

There are several natural ways in which this work can be extended. First, we are currently studying Bayesian methods for run-time prediction in more detail. Further, it should be straight-forward to apply our approach to randomized systematic search methods and we plan to do this in future work. We also plan to study the extent to which our results generalize to problems other than SAT and in particular to optimization problems. Finally, we would like to apply active learning approaches [7] in order to probe the parameter space in the most informative way in order to reduce training time.

References

1. B. Adenso-Daz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1), 2006. To appear.
2. R. Battiti and M. Brunato. Reactive search: machine learning for memory-based heuristics. Technical Report DIT-05-058, Università Degli Studi Di Trento, Dept. of information and communication technology, Trento, Italy, September 2005.
3. M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proc. of GECCO-02*, pages 11–18, 2002.
4. C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
5. F. Brglez, X. Y. Li, and M. F. Stallmann. On SAT instance classes and a method for reliable performance experiments with SAT solvers. *Annals of Mathematics and Artificial Intelligence*, 0:1–34, 2004.
6. T. Carchrae and J. C. Beck. Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):372–387, 2005.

7. D. A. Cohn, Z. Ghahramani, and M. I. Jordan. Active learning with statistical models. *JAIR*, 4:129–145, 1996.
8. C. Gebruers, B. Hnich, D. Bridge, and E. Freuder. Using CBR to select solution strategies in constraint programming. In *Proc. of ICCBR-05*, pages 222–236, 2005.
9. I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proc. of AAAI-99*, pages 654–660, Orlando, Florida, 1999.
10. C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proc. of AAAI-97*, 1997.
11. C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. of Automated Reasoning*, 24(1), 2000.
12. H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proc. of AAAI-99*, pages 661–666, 1999.
13. H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proc. of AAAI-02*, pages 655–660, 2002.
14. H. H. Hoos and T. Stützle. *Stochastic Local Search - Foundations & Applications*. Morgan Kaufmann, SF, CA, USA, 2004.
15. E. Horvitz, Y. Ruan, C. P. Gomes, H. Kautz, B. Selman, and D. M. Chickering. A Bayesian approach to tackling hard computational problems. In *Proc. of UAI-01*, 2001.
16. F. Hutter and Y. Hamadi. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. Technical Report MSR-TR-2005-125, Microsoft Research, Cambridge, UK, December 2005.
17. F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proc. of CP-02*, volume 2470, pages 233–248, 2002.
18. M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. In *Electronic Notes in Discrete Mathematics (ENDM)*, 2001.
19. K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proc. of CP-02*, 2002.
20. D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proc. of AAAI-97*, pages 321–326, 1997.
21. E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proc. of CP-04*, 2004.
22. D. J. Patterson and H. Kautz. Auto-WalkSAT: a self-tuning implementation of WalkSAT. In *Electronic Notes in Discrete Mathematics (ENDM), 9*, 2001.
23. C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.
24. B. Srivastava and A. Mediratta. Domain-dependent parameter selection of search-based algorithms compatible with user performance criteria. In *Proc. of AAAI-05*, 2005.
25. J. R. Thornton. Clause weighting local search for SAT. *J. of Automated Reasoning*, 2005.

Adaptive Clause Weight Redistribution

Abdelraouf Ishtaiwi[1,2], John Thornton[1,2], Anbulagan[3], Abdul Sattar[1,2], and Duc Nghia Pham[1,2]

[1] IIIS, Griffith University, QLD, Australia
[2] DisPRR, National ICT Australia Ltd, QLD, Australia
[3] Logic and Computation Program, National ICT Australia Ltd, Canberra, Australia
{a.ishtaiwi, j.thornton, anbulagan, abdul.sattar, duc-nghia.pham}@nicta.com.au

Abstract. In recent years, dynamic local search (DLS) clause weighting algorithms have emerged as the local search state-of-the-art for solving propositional satisfiability problems. However, most DLS algorithms require the tuning of domain dependent parameters before their performance becomes competitive. If manual parameter tuning is impractical then various mechanisms have been developed that can automatically adjust a parameter value during the search. To date, the most effective adaptive clause weighting algorithm is RSAPS. However, RSAPS is unable to convincingly outperform the best non-weighting adaptive algorithm AdaptNovelty$^+$, even though manually tuned clause weighting algorithms can routinely outperform the Novelty$^+$ heuristic on which AdaptNovelty$^+$ is based.

In this study we introduce R+DDFW$^+$, an enhanced version of the DDFW clause weighting algorithm developed in 2005, that not only adapts the total amount of weight according to the degree of stagnation in the search, but also incorporates the latest resolution-based preprocessing approach used by the winner of the 2005 SAT competition (R+AdaptNovelty$^+$). In an empirical study we show R+DDFW$^+$ improves on DDFW and outperforms the other leading adaptive (R+Adapt-Novelty$^+$, R+RSAPS) and non-adaptive (R+G^{2}WSAT) local search solv-ers over a range of random and structured benchmark problems.

1 Introduction

Since the development of the Breakout heuristic [1], clause weighting dynamic local search (DLS) algorithms for SAT have been intensively investigated, and continually improved [2,3]. However, the performance of these algorithms remained inferior to their non-weighting counterparts (e.g. [4]), until the more recent development of weight smoothing heuristics [5,6,7,8]). Such algorithms now represent the state-of-the-art for stochastic local search (SLS) methods on SAT problems. Interestingly, the most successful DLS algorithms (i.e. DLM [5], SAPS [7] and PAWS [8]) have converged on the same underlying weighting strategy: increasing weights on false clauses in a local minimum, then periodically reducing weights according to a problem specific parameter setting. DLM mainly differs from

PAWS by incorporating a plateau searching heuristic and PAWS mainly differs from SAPS by performing additive rather than multiplicative weight updates.

However, a key weakness of these approaches is that their performance depends on problem specific parameter tuning. This issue was partly addressed in the development of a reactive version of SAPS (RSAPS [7]) which used a similar adaptive noise mechanism to that used in AdaptNovelty$^+$ [9]. Nevertheless, as the 2005 International SAT competition (SAT2005) has shown, DLS algorithms, including RSAPS, have not proved competitive with the best SLS techniques when they are constrained to use fixed parameter values. This is explained by the sensitivity of the control parameters and by the lack of a sufficiently effective adaptive mechanism to adjust these parameters to specific problem instances.

In 2005, a new approach to clause weighting was developed, known as Divide and Distribute Fixed Weight (DDFW) [10]. DDFW's approach is to redistribute weight from satisfied to unsatisfied clauses in each local minimum, unifying the increase and decrease phases of weight control into a single action. This means there is no requirement for a problem specific parameter to decide when weight is to be reduced. In addition, DDFW only alters weights on those clauses that are false in a local minimum and an equal number of satisfied clauses. This makes it more efficient than earlier weight smoothing algorithms that also performed smoothing at each local minimum, but did so by adjusting weight on all the clauses in the problem (e.g. SDF [11]). However, DDFW still has a parameter (W_{init}) whose setting can effect performance by varying the amount of weight that is initially given to each clause. In the earlier empirical evaluation of DDFW this initial weight was fixed. However, the existence of such a parameter implies that DDFW could benefit from an adaptive mechanism to vary the amount of weight that is distributed according to the dynamic search conditions.

Also in 2005, it was shown that the performance of various SLS techniques can be significantly improved by the addition of a resolution-based preprocessing phase [12]. This work initially produced the winning algorithm in the SAT2005 satisfiable random problem category, R+AdaptNovelty$^+$. However, in the subsequent paper [12], the largest performance gains were obtained for clause weighting algorithms solving *structured* problem instances. Here R+AdaptNovelty$^+$ was convincingly outperformed by a R+RSAPS and a *tuned* version of R+PAWS on a range of quasigroup existence problems and standard structured SAT benchmarks.

The question we address in the current paper is which SLS SAT algorithm should be preferred in situations where parameter tuning is impractical and we have no other information that could guide us in choosing a particular approach. As this is exactly the situation we would expect to find in many real world applications, we take the relevance and importance of this question to be self evident. While the initial work on DDFW [10] showed that a fixed parameter version was able to outperform AdaptNovelty$^+$ and RSAPS on a range of random and structured SAT benchmarks, the question still remains whether the performance of DDFW can be further improved by incorporating a similar adaptive mechanism to that used by AdaptNovelty$^+$ and RSAPS to control the W_{init} parameter.

It also remains unanswered whether such an adaptive version of DDFW could derive enough benefit from resolution-based preprocessing to outperform the existing resolution-based versions of R+AdaptNovelty$^+$ or R+RSAPS. In addition, in SAT2005 a new SLS algorithm was introduced, G^{2}WSAT [13], which went on to win the silver medal in the random category of the competition. This algorithm has subsequently been improved and it too has yet incorporate a resolution-based preprocessor.

As a result of these considerations, our specific aim in the remainder of the paper is to introduce an adaptive resolution-incorporating version of DDFW (called R+DDFW$^+$) and to compare it with the three other most promising general purpose SLS SAT solvers, namely R+AdaptNovelty$^+$, R+RSAPS and an enhanced R+G^{2}WSAT. On the basis of an empirical study that covers a range of problems from SAT2005, the quasigroup existence domain and the SATLIB benchmark library, we conclude that R+DDFW$^+$ has the best overall performance of these methods, and that it derives significant benefits from its new adaptive mechanism.

2 Clause Weighting for SAT

Clause weighting local search algorithms for SAT follow the basic procedure of repeatedly flipping single literals that produce the greatest reduction in the sum of false clause weights. Typically, all literals are randomly initialized, and all clauses are given a fixed initial weight. The search then continues until no further cost reduction is possible, at which point the weight on all unsatisfied clauses is increased, and the search is resumed, punctuated with periodic weight reductions.

Existing clause weighting algorithms differ primarily in the schemes used to control the clause weights, and in the definition of the points where weight should be adjusted. Multiplicative methods, such as SAPS, generally adjust weights when no further improving moves are available in the local neighbourhood. This can be when all possible flips lead to a worse cost, or when no flip will improve cost, but some flips will lead to equal cost solutions. As multiplicative real-valued weights have much finer granularity, the presence of equal cost flips is much more unlikely than for an additive approach (such as DLM or PAWS), where weight is adjusted in integer units. This means that additive approaches frequently have the choice between adjusting weight when no improving move is available, or taking an equal cost (flat) move.

Despite these differences, the three most well-known clause weighting algorithms (DLM [5], SAPS [7] and PAWS [8]) share a similar structure in the way that weights are updated:[1] Firstly, a point is reached where no further improvement in cost appears likely. The precise definition of this point depends

[1] Additionally, a fourth clause weighting algorithm, GLSSAT [14], uses a similar weight update scheme, additively increasing weights on the least weighted unsatisfied clauses and multiplicatively reducing weights whenever the weight on any one clause exceeds a predefined threshold.

on the algorithm, with DLM expending the greatest effort in searching plateau areas of equal cost moves, and SAPS expending the least by only accepting cost improving moves. Then all three methods converge on increasing weights on the currently false clauses (DLM and PAWS by adding one to each clause and SAPS by multiplying the clause weight by a problem specific parameter $\alpha > 1$). Each method continues this cycle of searching and increasing weight, until, after a certain number of weight increases, clause weights are reduced (DLM and PAWS by subtracting one from all clauses with weight > 1 and SAPS by multiplying all clause weights by a problem specific parameter $\rho < 1$). SAPS is further distinguished by reducing weights probabilistically (according to a parameter P_{smooth}), whereas DLM and PAWS reduce weights after a fixed number of increases (again controlled by parameter). PAWS is mainly distinguished from DLM in being less likely to take equal cost or flat moves. DLM will take up to θ_1 consecutive flat moves, unless all available flat moves have already been used in the last θ_2 moves. PAWS does away with these parameters, taking flat moves with a fixed probability of 15%, otherwise it will increase weight.

However, as we have stressed in the introduction, the performance of these clause weighting algorithms remains very sensitive to the settings of their problem specific parameters (this has been shown in detail in [15]). While this sensitivity is also a problem for the non-weighting algorithms of the WalkSAT family, it has been somewhat counteracted by the use of heuristics that adapt parameter settings during the course of the search. The most successful of these algorithms, AdaptNovelty$^+$, works by adapting a noise parameter that controls whether a move is selected randomly or deterministically [9]. In simplified terms, the likelihood of making a random choice is increased the longer the search continues without achieving an improvement in the objective function. A similar scheme was added to SAPS, producing reactive SAPS or RSAPS [7]. However, adapting SAPS was not as successful as adapting Novelty, for, while a tuned SAPS generally produces better performance than a tuned Novelty+, RSAPS has not been able to reach the consistent performance AdaptNovelty$^+$ in the recent SAT competitions. One reason for this may be that SAPS requires the setting of *three* parameters to achieve its best performance, while RSAPS only adapts one of these parameters. Similarly, DLM requires the setting of at least three parameters before producing its best performance. In contrast, PAWS (like Novelty) only requires the tuning of a single parameter, but to date no successful heuristic has been discovered that can automatically adapt this value.

More recently, work has concentrated on *learning* empirical hardness models in order to predict the best parameter settings for SAPS [16]. This approach requires a set of training instances that are repeatedly solved by SAPS using different parameter settings. After this training phase, parameter settings can be generated for previously unseen instances taken from the same problem class. Results from this work are encouraging and could be generally applied to other local search algorithms. However, the weakness is that training is required on a representative test set before good predictions can be produced. It remains to be seen whether a general model can be devised that can predict good parameter settings for the SAT domain as a whole. In the meantime, if we are limited to solving problems from an undisclosed problem distribution and if manual parameter tuning is ruled out of court, then the best available clause weighting algorithm is probably RSAPS (discounting DDFW for the moment).

3 Divide and Distribute Fixed Weights

DDFW introduces two ideas into the area of clause weighting algorithms for SAT. Firstly, it evenly distributes a fixed quantity of weight across all clauses at the start of the search, and then escapes local minima by *transferring weight from satisfied to unsatisfied clauses*. The other existing state-of-the-art clause weighting algorithms have all divided the weighting process into two distinct steps: i) increasing weights on false clauses in local minima and ii) decreasing or normalising weights on all clauses after a series of increases, so that weight growth does not spiral out of control. DDFW combines this process into a single step of weight transfer, thereby dispensing with the need to decide when to reduce or normalise weight. In this respect, DDFW is similar to the predecessors of SAPS (SDF [6] and ESG [11]), which both adjust *and* normalise the weight distribution in each local minimum. Because these methods adjust weight across all clauses, they are considerably less efficient than SAPS, which normalises weight after visiting a series of local minima.[2] DDFW escapes the inefficiencies of SDF and ESG by only transferring weights between pairs of clauses, rather than normalising weight on all clauses. This transfer involves selecting a single satisfied clause for each currently unsatisfied clause in a local minimum, reducing the weight on the satisfied clause by an integer amount and adding that amount to the weight on the unsatisfied clause. Hence DDFW retains the additive (integer) weighting approach of DLM and PAWS, and combines this with an efficient method of weight redistribution, i.e. one that keeps all weight reasonably normalised without repeatedly adjusting weights on all clauses.

DDFW's weight transfer approach also bears similarities to the operations research subgradient optimisation techniques discussed in [11]. In these approaches, Lagrangian multipliers, analogous to the clause weights used in SAT, are associated with problem constraints, and are adjusted in local minima so that multipliers on unsatisfied constraints are increased and multipliers on satisfied constraints are reduced. This *symmetrical* treatment of satisfied and unsatisfied constraints is mirrored in DDFW, but not in the other SAT clause weighting approaches (which increase weights and then adjust). However, DDFW differs from subgradient optimisation in that weight is only transferred between pairs of clauses and not across the board, meaning less computation is required.

[2] Increasing weight on *false* clauses in a local minimum is efficient because only a small proportion of the total clauses will be false at any one time.

Algorithm 1. DDFW$^+$($\mathcal{F}$)

```
1: randomly instantiate each literal in $\mathcal{F}$;
2: set the weight $w_a$ of each clause $c_a \in \mathcal{F}$ to two;
3: set the minimum $m$ to the number of false clauses $c_f \in F$;
4: set counter $i$ to zero and boolean $b$ to false;
5: while solution is not found and not timeout do
6:     calculate the list $\mathcal{L}$ of literals causing the greatest reduction in weighted cost $\Delta w$ when flipped;
7:     if ($\Delta w < 0$) or ($\Delta w = 0$ and probability $\leq 15\%$) then
8:         randomly flip a literal in $\mathcal{L}$;
9:         if number of false clauses $< m$ then
10:            set counter $i$ to zero and minimum $m$ to the number of false clauses;
11:        else
12:            increment counter $i$ by one;
13:            if $i \geq$ number of literals in $\mathcal{F}$ then
14:                set counter $i$ to zero;
15:                if $b$ is false then
16:                    increase the weight $w_a$ of each clause $c_a \in \mathcal{F}$ by one;
17:                    set boolean $b$ to true;
18:                else
19:                    set the weight $w_s$ of each satisfied clause $c_s \in \mathcal{F}$ to two;
20:                    set the weight $w_f$ of each false clause $c_f \in \mathcal{F}$ to three;
21:                    set boolean $b$ to false;
22:                end if
23:            end if
24:        end if
25:    else
26:        for each false clause $c_f \in \mathcal{F}$ do
27:            select a satisfied same sign neighbouring clause $c_n$ with maximum weight $w_n$;
28:            if $w_n < 2$ then
29:                randomly select a clause $c_n$ with weight $w_n \geq 2$;
30:            end if
31:            if $w_n > 2$ then
32:                transfer a weight of two from $c_n$ to $c_f$;
33:            else
34:                transfer a weight of one from $c_n$ to $c_f$;
35:            end if
36:        end for
37:    end if
38: end while
```

3.1 Exploiting Neighbourhood Structure

The second and more original idea developed in DDFW, is the exploitation of neighbourhood relationships between clauses when deciding which pairs of clauses will exchange weight.

We term clause c_i to be a neighbour of clause c_j, if there exists at least one literal $l_{im} \in c_i$ and a second literal $l_{jn} \in c_j$ such that $l_{im} = l_{jn}$. Furthermore, we term c_i to be a *same sign* neighbour of c_j if the sign of any $l_{im} \in c_i$ is equal to the sign of any $l_{jn} \in c_j$ where $l_{im} = l_{jn}$. From this it follows that each literal $l_{im} \in c_i$ will have a set of same sign neighbouring clauses $C_{l_{im}}$. Now, if c_i is false, this implies all literals $l_{im} \in c_i$ evaluate to false. Hence flipping any l_{im} will cause it to become true in c_i, and also to become true in all the same sign neighbouring clauses of l_{im}, i.e. $C_{l_{im}}$. Therefore, flipping l_{im} will *help* all the clauses in $C_{l_{im}}$, i.e. it will increase the number of true literals, thereby increasing the overall level of satisfaction for those clauses. Conversely, l_{im} has a corresponding set of opposite sign clauses that would be *damaged* when l_{im} is flipped.

The reasoning behind the DDFW neighbourhood weighting heuristic proceeds as follows: if a clause c_i is false in a local minimum, it needs extra weight in order to encourage the search to satisfy it. If we are to pick a neighbouring clause c_j that will donate weight to c_i, we should pick the clause that is most able to pay. Hence, the clause should firstly already be satisfied. Secondly, it should be a same sign neighbour of c_i, as when c_i is eventually satisfied by flipping l_{im}, this will also raise the level of satisfaction of l_{im}'s same sign neighbours. However, taking weight from c_j only increases the chance that c_j will be helped when c_i is satisfied, i.e. not all literals in c_i are necessarily shared as same sign literals in c_j, and a non-shared literal may be subsequently flipped to satisfy c_i. The third criteria is that the donating clause should also have the largest store of weight within the set of satisfied same sign neighbours of c_i

The intuition behind the DDFW heuristic is that clauses that share same sign literals should form alliances, because a flip that benefits one of these clauses will always benefit some other member(s) of the group. Hence, clauses that are connected in this way will form groups that tend towards keeping each other satisfied. However, these groups are not closed, as each clause will have clauses within its own group that are connected by other literals to other groups. Weight is therefore able to move between groups as necessary, rather than being uniformly smoothed (as in existing methods).

3.2 Adapting DDFW

The new feature introduced in this study is the development of an adaptive mechanism that alters the total amount of weight that DDFW distributes according to the degree of stagnation in the search. This DDFW$^+$ heuristic is detailed in lines 9-24 of Algorithm 1. Previously DDFW would have initialised the weight of each clause to W_{init} (which was fixed at 8 in [10]). Now this initialisation value is set at two in line 2 of Algorithm 1, but can be altered during the search as follows: if the search executes a consecutive series of i flips without reducing the total number of false clauses, where i is equal to the number of literals in the problem, then the amount of weight on each clause is increased by one in the first instance. However, if after increasing weights, the search enters another consecutive series of i flips without improvement, then it will reset the weight on each satisfied clause back to two and on each false clause back to three. The search then continues to follow each increase with a reset and each reset with an increase. In this way a long period of stagnation will produce oscillating phases of weight increase and reduction, such that the total weight can never exceed 3 times the total number of clauses $c_a \in \mathcal{F}$ plus the total number of false clauses $c_f \in \mathcal{F}$.

The reasoning behind this adaptive heuristic is based on our observation that manually adjusting DDFW's original parameter W_{init} has a noticeable effect runtime performance, and that on several problems the default value of eight was not optimal. This is illustrated in Figure 1, which shows that on problem (a) $W_{init} = 8$ is near optimal whereas on problem (b) $W_{init} = 2$ is the better choice (if we consider the underlying trend). We conjectured that we could circumvent the need to

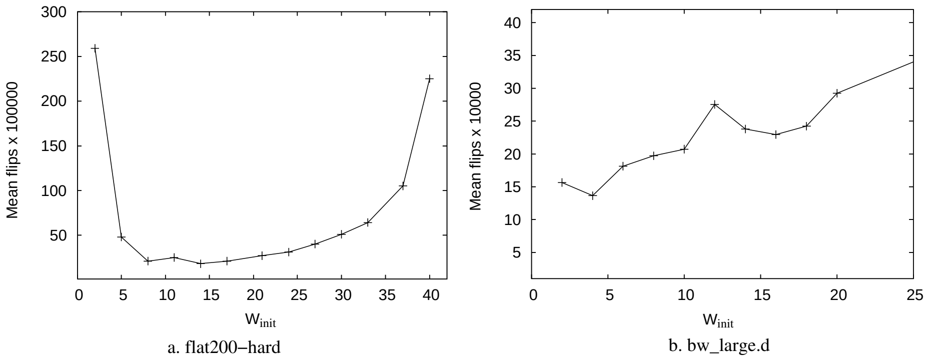


Fig. 1. Flip performance of DDFW for various settings of the W_{init} parameter

initialise the clauses with more weight at the start of the search by allowing context sensitive weight increases during the search. Hence we developed a stagnation measure, much like the measures used in AdaptNovelty and RSAPS, that injects extra weight when no cost improvement occurs and made the frequency of this injection depend on the size of the problem. The unusual feature of the DDFW$^+$ heuristic is that the search will only effect one increase after which, if stagnation is observed again, the weights are reset. This reset mechanism was adopted after a series of empirical trials that tested various combinations of weight increase and decrease phases. Our main difficulty was to keep the weight growth within bounds and we could find no decrease scheme that worked well across a wide range of problems without requiring a further problem dependent parameter (which would obviously defeat the purpose of the study). We therefore settled on a simple reset strategy that places a strict limit on weight growth and avoids adding an additional parameter.

4 Resolution Based Preprocessing

As discussed in the introduction, significant performance benefits have been gained by preprocessing a problem using resolution before starting a search. This result is already well-known in the complete search community, where Satz [17] uses a restricted resolution procedure, adding resolvents of length ≤ 3, as a preprocessor before running the complete backtrack search. The same procedure has now been added to AdaptNovelty$^+$, PAWS, RSAPS and WalkSAT [12], and there is empirical evidence to suggest that clause weighting algorithms in particular benefit from this approach when solving structured real-world problems.

Resolution itself is a rule of inference widely used in automated deduction [18,19,20]. In the present study, as in [12], we implement the Satz resolution process (see Algorithm 2) as follows: when two clauses of a CNF formula have the property that some variable x_i occurs positively in one and negatively in the other, the resolvent of the clauses is a disjunction of all the literals occurring in the clauses except x_i and $\overline{x_i}$. For example, the clause $(x_2 \vee x_3 \vee \overline{x_4})$ is the

Algorithm 2. ComputeResolvents($\mathcal{F}$)

```
1: for each clause $c_1$ of length $\leq 3$ in $\mathcal{F}$ do
2:     for each literal $l$ of $c_1$ do
3:         for each clause $c_2$ of length $\leq 3$ in $\mathcal{F}$ s.t. $\bar{l} \in c_2$ do
4:             Compute resolvent $r = (c_1 \backslash \{l\}) \cup (c_2 \backslash \{\bar{l}\})$;
5:             if $r$ is empty then
6:                 return "unsatisfiable";
7:             else
8:                 if $r$ is of length $\leq 3$ then
9:                     $\mathcal{F} := \mathcal{F} \cup \{r\}$;
10:                end if
11:            end if
12:        end for
13:    end for
14: end for
```

resolvent for the clauses $(\overline{x_1} \vee x_2 \vee x_3)$ and $(x_1 \vee x_2 \vee \overline{x_4})$ and is added to the clause set. The new clauses, provided they are of length ≤ 3, can in turn be used to produce other resolvents. The process is repeated until saturation. Duplicate and subsumed clauses are deleted, as are tautologies and any duplicate literals in a clause. It is worth noting that this resolution phase takes polynomial time.

5 Experimental Evaluation

As the resolution process is encapsulated in a preprocessing phase, it can be added to an existing SAT solver as a separate module, leaving the original solver unaltered. In our experimental study we added this preprocessing phase (as defined in Algorithm 2) to DDFW, DDFW$^+$, RSAPS, AdaptNovelty$^+$ and G^{2}WSAT, producing R+DDFW, R+DDFW$^+$, R+RSAPS, R+AdaptNovelty$^+$ and R+G^2-WSAT. Of these algorithms, R+RSAPS and R+AdaptNovelty$^+$ have already been entered into SAT2005 and reported in [12].[3] However, R+DDFW, R+DDFW$^+$ and R+G^{2}WSAT are new algorithms whose performance has yet to be reported.[4] We chose to compare DDFW with R+AdaptNovelty$^+$ and R+G^{2}WSAT because these two algorithms were the gold and silver medal winners in the SAT2005 satisfiable random category competition and achieved the best overall local search results in terms of the number of problems solved. We chose R+RSAPS because it was the best performing clause weighting algorithm in the competition. Together, therefore, these three algorithms can lay claim to being the state-of-the-art for general purpose local search SAT solving when manual parameter tuning is disallowed.

To evaluate the relative performance of these algorithms we divided our empirical study into four areas: firstly, we attempted to reproduce a reduced problem set similar to that used in the random category of the SAT competition

[3] AdaptNovelty$^+$ and RSAPS are available as part of the UBCSAT solver from http://www.satlib.org/ubcsat/

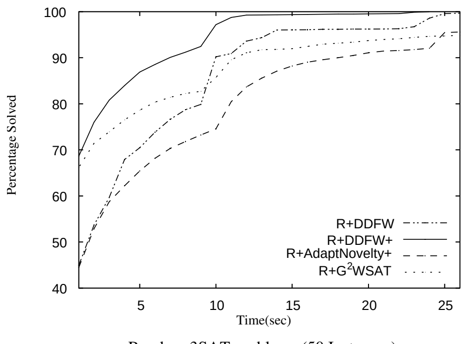
[4] G^{2}WSAT is available at http://www.laria.u-picardie.fr/%7Ecli/g2wsat2005.c. This latest version is described by the authors as generally more than 50% faster than the version entered in SAT2005.

(as this is the domain where local search techniques have dominated). To do this we selected the 50 satisfiable $k3$ problems from the SAT2004 competition benchmark. Secondly, we obtained the 10 SATLIB quasigroup existence problems used in [12]. These problems are relevant because they exhibit a balance between randomness and structure, while also producing clause sets to which resolution can be applied effectively. Thirdly, we obtained the structured problem set used to originally evaluate SAPS [7]. These problems have been widely used to evaluate clause weighting algorithms (e.g. in [8]) and contain a representative cross-section taken from the DIMACS and SATLIB libraries. In this set we also included 4 of the well-known DIMACS 16-bit parity learning problems. Finally, we used the 16 ferry planning problems from the SAT2005 competition that our local search techniques were able to solve. This was to give an indication of relative performance on the SAT2005 industrial problems.

Overall, the problem set is designed to show how R+DDFW$^+$ compares in absolute terms to the other algorithms and to examine the relative effect of the adaptive mechanism on differing problem classes. For this reason we also include the results for R+DDFW (i.e. *without* the adaptive mechanism). All experiments were performed on a Dell machine with 3.1GHz CPU and 1GB memory, except for the quasigroup problems which were run on a Sun supercomputer with 8 $\times$ Sun Fire V880 servers, each with 8 $\times$ UltraSPARC-III 900MHz CPU and 8GB memory per node. Cut-offs for the various algorithms were set as follows: first R+DDFW was given 10 trials on each problem with a flip cut-off of 1,000,000. If it was unable to solve any trial then the cut-off was raised to 10,000,000, and then in steps of 10,000,000 until at least one solution was found. R+DDFW was then allowed 100 trials at the given flip cut-off for all instances except the ferry problems, where it was limited to 10 trials. The total time allowed for R+DDFW on each set of 10 or 100 trials was then recorded and all other algorithms were given this as a time cut-off on each problem. The following results detail the mean time in seconds (including the resolution preprocessing step), mean flips and the success rate for these cut-offs (results in bold indicate the best performance for a particular problem).

5.1 SAT Competition Problem Results

The results in Figure 2a graph the performance of R+DDFW$^+$, R+DDFW, R+AdaptNovelty$^+$ and R+G^{2}WSAT after applying resolution on the 50 $k3$ problems from the SAT2004 competition (as R+RSAPS had very poor performance on the random instances it has been omitted from the figure and the following discussion). The graph shows the cumulative percentage of problems solved against runtime, assuming that each instance is solved in parallel (for example, in Figure 2a after 5 seconds approximately 71% of the 50 $\times$ 100 trials for R+DDFW will have terminated). Here R+DDFW$^+$ and R+DDFW were the only solvers that could reach a 100% success rate over all trials. Although R+G^{2}WSAT was competitive and could solve the easier problems faster than R+DDFW, it was unable to match R+DDFW as problem difficulty increased. Overall the graph shows that R+DDFW$^+$ has the superior perfor-



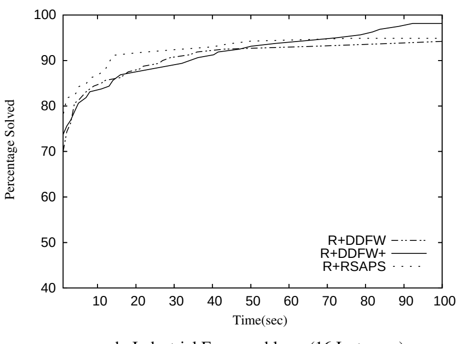


Fig. 2. Results for the SAT2004 random problems and SAT2005 industrial problems

mance across the range of problem sizes, clearly dominating R+DDFW and thereby demonstrating that the new adaptive heuristic can positively affect run-time performance. Figure 2a also shows that R+G^{2}WSAT generally dominates R+AdaptNovelty$^+$, although R+AdaptNovelty$^+$ does match R+G^{2}WSAT's success rate over the whole problem set.

The results for the SAT2005 industrial ferry problems are shown in Figure 2b and in Table 1 (as R+G^{2}WSAT and R+AdaptNovelty$^+$ were only able to solve 29% and 9% of the ferry instances respectively, they have been removed from the graphical analysis). Looking at Figure 2b we can see that R+RSAPS, after performing poorly on the random problems, is now able to dominate R+DDFW across the range of the ferry problems, but cannot quite reach R+DDFW$^+$'s 97.5% success rate. However, Table 1 shows that R+RSAPS is able to solve 10 of the 16 ferry problems faster than either DDFW variant, and that R+DDFW$^+$'s superior success rate is largely based on instance ferry4001. We must therefore conclude that there is little to choose between R+RSAPS and R+DDFW$^+$ on these problems. Nevertheless, R+DDFW$^+$ does more clearly outperform R+DDFW and again demonstrates that the adaptive heuristic can make noticeable improvements.

5.2 Quasigroup Problem Results

Table 2 shows the performance of the solvers on the quasigroup problems. Here we can see that R+DDFW and R+DDFW$^+$ clearly emerge as the two best solvers, sharing the best results for each instance and both achieving an overall success rate of 100%. Comparing between the two DDFW methods, for the first time it becomes unclear whether the adaptive heuristic has made any difference, as, for most instances the results are comparable. However R+DDFW$^+$ does exhibit noticeably better performance on instance qg1-08, whereas R+DDFW shows equally strong performance on qg7-13. We should therefore conclude that the adaptive mechanism does not change the overall performance of DDFW on this problem set, although it can make a difference, either positively or negatively, on individual instances.

Table 1. Results for the SAT2005 industrial ferry planning problems

	R+DDFW$^+$			R+DDFW			R+AdaptNovelty$^+$			G^{2}WSAT			R+RSAPS		
Problems	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
ferry3994	3.48	2,073,195	100	1.1	786,967	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**0.6**	**530,501**	**100**
ferry3995	1.54	933,726	100	0.6	458,302	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**0.1**	**89,730**	**100**
ferry3996	0.0	7,903	100	0.0	13,942	100	3.9	8,204,511	20	0.1	275,547	100	**0.0**	**7,741**	**100**
ferry3997	10.3	8,238,690	60	**10.3**	**5,055,539**	**90**	*n/a*	*n/a*	0	*n/a*	*n/a*	0	9.2	6,742,006	50
ferry3998	0.0	6,526	100	0.0	8,586	100	2.1	3,344,936	100	0.1	180,334	100	**0.0**	**5,070**	**100**
ferry3999	9.81	5,312,170	100	3.2	1,908,547	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**0.6**	**304,680**	**100**
ferry4000	0.0	31,774	100	0.0	19,280	100	*n/a*	*n/a*	0	1.8	2,442,300	80	**0.0**	**12,771**	**100**
ferry4001	**63.1**	**24,392,288**	**100**	99.4	40,117,368	90	*n/a*	*n/a*	0	*n/a*	*n/a*	0	90.0	54,061,467	80
ferry4002	0.0	9,637	100	0.0	20,336	100	4.8	7,535,284	30	2.1	1,958,552	90	**0.0**	**3,852**	**100**
ferry4003	21.2	10,395,968	100	21.2	7,773,439	50	*n/a*	*n/a*	0	*n/a*	*n/a*	0	**7.2**	**2,884,301**	**100**
ferry4004	0.0	30,348	100	0.1	40,547	100	*n/a*	*n/a*	0	2.4	2,437,826	50	**0.0**	**20,394**	**100**
ferry4006	0.0	14,640	100	0.0	17,697	100	*n/a*	*n/a*	0	4.9	2,616,491	20	**0.0**	**9,160**	**100**
ferry4008	**0.0**	**33,192**	**100**	0.1	51,796	100	*n/a*	*n/a*	0	3.2	2,655,066	20	0.1	42,938	100
ferry4009	**0.0**	**23,163**	**100**	0.1	24,015	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	0.1	17,612	100
ferry3992	**0.1**	**60,525**	**100**	0.2	102,413	100	*n/a*	*n/a*	0	*n/a*	*n/a*	0	0.2	92,346	100
ferry3993	**0.0**	**26,878**	**100**	0.1	43,595	100	*n/a*	*n/a*	0	7.2	3,399,169	10	0.2	54,742	100

Table 2. Results for Quasigroup SATLIB problems

	R+DDFW$^+$			R+DDFW			R+AdaptNovelty$^+$			R+G^{2}WSAT			R+RSAPS		
Problems	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
qg1-07	**0.0**	**4,388**	**100**	0.1	11,375	100	0.2	14,840	100	0.1	9,600	100	0.1	4,901	100
qg1-08	**10.2**	**352,276**	**100**	21.8	601,271	100	33.8	1,076,689	100	28.8	2,818,904	100	64.6	2,153,008	99
qg2-07	0.0	2,361	100	**0.0**	**2,035**	**100**	0.1	9,094	100	0.1	5,073	100	0.1	2,478	100
qg2-08	**57.5**	**1,556,545**	**100**	60.0	1,346,438	100	77.1	1,906,196	20	79.8	4,569,088	50	71.5	1,879,019	70
qg3-08	**0.1**	**16,867**	**100**	0.1	21,986	100	0.6	78,849	100	0.1	24,534	100	0.2	11,049	100
qg4-09	**0.2**	**25,311**	**100**	0.2	26,123	100	1.5	169,169	100	0.7	142,619	100	1.2	54,920	100
qg5-11	0.2	7,303	100	**0.2**	**6,797**	**100**	2.3	131,924	100	0.4	29,992	100	0.6	11,014	100
qg6-09	0.0	478	100	**0.0**	**466**	**100**	0.0	3,644	100	0.0	686	100	0.6	11,753	100
qg7-09	**0.0**	**292**	**100**	0.0	299	100	0.0	698	100	0.0	412	100	0.0	295	100
qg7-13	9.3	229,258	100	**3.2**	**122,091**	**100**	16.3	5,351,459	56	*n/a*	*n/a*	0	24.9	373,456	10

5.3 Structured Problem Results

Table 3 shows the results for the structured problems taken from the original SAPS problem set [7] and the parity learning problems taken from the original PAWS study [8]. This set comprises of two blocks world planning (bw) problems, two logistics planning instances, two flat graph coloring problems (flat), two all-interval-series problems (ais) and four 16-bit parity learning problems (par16*). The results confirm our earlier observation from the random problem results that G^{2}WSAT does not scale as well as DDFW. In this case R+G^{2}WSAT is the best algorithm on the smaller ais, logistics and flat problems, but is outperformed by R+DDFW on each of the larger instances of these problems. In addition, R+RSAPS has stronger performance than R+DDFW on the ais and par16 problems.

However, the situation changes if we consider the performance of R+DDFW$^+$. In comparison to R+DDFW, R+DDFW$^+$ is better on the ais10, both logistics and all par16 problems, whereas R+DDFW is only better on the ais12 and flat200 problems (the two methods perform identically on the bw problems be-

Table 3. Results for structured problems from the SAPS and PAWS original studies, (the = symbol means that R+DDFW$^+$ behaves identically to R+DDFW on these problems)

	R+DDFW$^+$			R+DDFW			R+AdaptNovelty$^+$			R+G^{2}WSAT			R+RSAPS		
Problems	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%	Time	Flips	%
ais10	0.0	298,650	100	0.5	498,911	100	1.4	1,214,321	100	**0.0**	**112,044**	**100**	0.0	25,459	100
ais12	5.0	4,036,866	100	2.3	1,934,170	100	10.1	7,328,426	51	2.4	1,854,652	100	**0.2**	**187,743**	**100**
logistics-c	0.0	242,540	100	0.3	414,645	100	0.0	26,696	100	**0.0**	**23,623**	**100**	0.0	5,364	100
logistics-d	**0.1**	**16,708**	**100**	0.1	25,869	100	0.1	109,650	100	0.5	350,711	100	0.1	20,918	100
flat200-m	0.3	262,905	100	0.2	161,902	100	0.2	351,563	100	**0.1**	**150,588**	**100**	0.4	362,786	100
flat200-h	3.2	2,814,221	100	**1.0**	**1,014,878**	**100**	3.6	8,166,964	36	2.4	5,535,185	100	3.5	3,517,562	94
bw_large.c	=	=	**100**	**0.6**	**145,607**	**100**	6.7	5,660,460	67	*n/a*	*n/a*	0	21.3	4,258,483	91
bw_large.d	=	=	**100**	**1.4**	**184,874**	**100**	13.4	7,974,818	38	*n/a*	*n/a*	0	*n/a*	*n/a*	0
par16-1	**4.3**	**3,828,086**	**100**	7.1	5,229,852	50	7.4	15,608,349	15	*n/a*	*n/a*	0	7.4	1,164,862	80
par16-2	23.2	21,670,517	100	27.9	20,542,514	60	36.8	54,634,563	10	*n/a*	*n/a*	0	**16.0**	**17,581,843**	**100**
par16-3	**7.7**	**7,146,517**	**100**	24.4	17,959,087	70	32.7	50,828,991	40	31.8	26,133,070	30	16.0	18,890,265	100
par16-4	**2.9**	**2,699,444**	**100**	11.4	12,800,152	100	26.8	41,099,634	50	26.5	51,205,540	60	8.1	9,445,556	100

cause the large number of literals mean the adaptive mechanism is not used). These results show that the R+DDFW$^+$ adaptive mechanism has again produced noticeable performance benefits, and has improved the overall behaviour of R+DDFW on this problem set. In addition, if we take a simple count of the number of problems on which R+DDFW$^+$ dominates we can see that it is also the best of the five algorithms considered.

6 Analysis and Conclusions

Overall we can conclude that the addition of an adaptive mechanism has improved the performance of DDFW over the entire range of the problem sets we have considered. The strongest dominance was observed on the random 3-SAT and parity problems (shown in Figure 2a and Table 3 respectively). On the other problems R+DDFW$^+$ improved over R+DDFW on 10 of the 16 ferry problems (in Table 1), 6 of the 10 quasigroup problems (in Table 2) and stays neutral on the remaining real-world problems (in Table 3).

We can further conclude that R+DDFW (i.e. even without the adaptive mechanism) has the better overall performance in comparison to AdaptNovelty$^+$, G^{2}WSAT and R+RSAPS. If we first look at R+G^{2}WSAT, while it performed well on the smaller random problems, it could not match R+DDFW on the larger more difficult random problems. In the other categories R+G^{2}WSAT was less competitive, again showing promise on the smaller structured problems in Table 3, but failing to scale up as well as R+DDFW on the more difficult problems. Interestingly, G^{2}WSAT performed strongly on the quasigroup problems when no resolution was performed, but was uncompetitive after resolution (these results are not reported in the current paper). This confirms the findings in [12] that suggest clause weighting algorithms can gain more advantage from resolution than non-weighting algorithms. In addition, R+G^{2}WSAT was uniformly worse than R+DDFW on the ferry problems.

Turning our attention to R+RSAPS, this algorithm showed slightly better performance than R+DDFW on the structured and ferry problems, dominating on 10 of the 16 ferry problems and on all the parity problems, with R+DDFW showing the better performance on the remaining 6 ferry problems and on the other larger structured problems. However, R+RSAPS was outperformed by R+DDFW$^+$ on the parity problems, was uniformly worse on the random problems and was uncompetitive with R+DDFW on the quasigroup problems, thereby failing to show the same robust performance as R+DDFW and R+DDFW$^+$ across the whole range of problem sets. Our third comparison algorithm, R+AdaptNovelty$^+$, also had the worst overall performance, being unable to achieve outright dominance on any of the problems considered.

In a further unpublished study (not reported here) we investigated the effect of the preprocessing resolution step on the performance of each algorithm. This showed that resolution has little effect on the random problem instances but has a positive effect on the quasigroup instances, with the effect being more pronounced for R+DDFW and less pronounced for R+G^{2}WSAT. For the real world instances, resolution was also generally helpful for the ferry, ais, logistics and parity problems but had little or no effect on the bw and flat problems.

In conclusion, we have introduced and integrated a new adaptive mechanism into the DDFW algorithm. This mechanism is unusual in that it oscillates between increasing and resetting clause weights, timing these changes according to a stagnation measure defined by the number of problem literals. While the increase mechanism increments the existing weight profile, the reset mechanism eliminates the profile entirely, returning the weights to their initial state. We conjecture that this dramatic and discontinuous change in the weighted cost surface increases diversity by allowing the search to explore new trajectories. The reset mechanism also ensures that the amount of weight added to a problem is strictly controlled without requiring an additional weight decrease parameter.

In order to evaluate the new adaptive algorithm, R+DDFW$^+$, we also incorporated the latest resolution-based preprocessing technique used by the winning algorithm in the SAT2005 competition. In a broad ranging empirical study we have shown that integrating our new adaptive mechanism into DDFW can significantly enhance its overall performance. We have also shown that R+DDFW$^+$ has the best overall performance across a range of representative structured and random problem instances in comparison to three of the best SLS solvers currently available. The results suggest that R+DDFW$^+$ should be the SLS algorithm of choice in situations where the characteristics of a problem domain are not known in advance and manual parameter tuning is not practical. In future work it would be worthwhile to experiment with other resolution techniques to see if further performance benefits can be obtained.

Acknowledgement. The authors would like to acknowledge the financial support of National ICT Australia (NICTA) and the Queensland government. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative and also through the Australian Research Council.

References

1. Morris, P.: The Breakout method for escaping from local minima. In: Proceedings of 11th AAAI. (1993) 40–45
2. Cha, B., Iwama, K.: Adding new clauses for faster local search. In: Proceedings of 13th AAAI. (1996) 332–337
3. Frank, J.: Learning short-term clause weights for GSAT. In: Proceedings of 15th IJCAI. (1997) 384–389
4. McAllester, D., Selman, B., Kautz, H.: Evidence for invariants in local search. In: Proceedings of 14th AAAI. (1997) 321–326
5. Wu, Z., Wah, B.: An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In: Proceedings of 17th AAAI. (2000) 310–315
6. Schuurmans, D., Southey, F.: Local search characteristics of incomplete SAT procedures. In: Proceedings of 10th AAAI. (2000) 297–302
7. Hutter, F., Tompkins, D., Hoos, H.: Scaling and Probabilistic Smoothing: Efficient dynamic local search for SAT. In: Proceedings of 8th CP. (2002) 233–248
8. Thornton, J., Pham, D.N., Bain, S., Ferreira Jr., V.: Additive versus multiplicative clause weighting for SAT. In: Proceedings of 19th AAAI. (2004) 191–196
9. Hoos, H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of 19th AAAI. (2002) 655–660
10. Ishtaiwi, A., Thornton, J., Sattar, A., Pham, D.N.: Neighbourhood clause weight redistribution in local search for SAT. In: Proceedings of 11th CP. (2005) 772 – 776
11. Schuurmans, D., Southey, F., Holte, R.: The exponentiated subgradient algorithm for heuristic boolean programming. In: Proceedings of 17th IJCAI. (2001) 334–341
12. Anbulagan, Pham, D., Slaney, J., Sattar, A.: Old resolution meets modern SLS. In: Proceedings of 20th AAAI. (2005) 354–359
13. Li, C.M., Huang, W.: Diversification and determinism in local search for satisfiability. In: Proceedings of 8th SAT. (2005) 158–172
14. Mills, P., Tsang, E.: Guided local search applied to the satisfiability (SAT) problem. In: Proceedings of 15th ASOR. (1999) 872–883
15. Thornton, J.: Clause weighting local search for SAT. Journal of Automated Reasoning (2006) (to appear)
16. Hutter, F., Hamadi, Y.: Parameter adjustment based on performance prediction: Towards an instance aware problem solver. In: Technical Report: MSR-TR-2005-125, Microsoft Research, WA. (2005)
17. Li, C.M., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: Proceedings of 3rd CP. (1997) 341–355
18. Quine, W.V.: A way to simplify truth functions. American Mathematical Monthly **62** (1955) 627–631
19. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 201–215
20. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM **12** (1965) 23–41

Localization of an Underwater Robot Using Interval Constraint Propagation

Luc Jaulin[1,2,3]

[1] E3I2, ENSIETA, 2 rue François Verny, 29806 Brest Cédex 09
[2] E3I2, ENSIETA, France
luc.jaulin@ensieta.fr
http://www.ensieta.fr/e3i2/Jaulin/
[3] GESMA (Groupe d'Etude Sous-Marine de l'Atlantique), Brest, France

Abstract. Since electromagnetic waves are strongly attenuated inside the water, the satellite based global positioning system (GPS) cannot be used by submarine robots except at the surface of the water. This paper shows that the localization problem in deep water can often be cast into a continuous constraints satisfaction problem where interval constraints propagation algorithms are particularly efficient. The efficiency of the resulting propagation methods is illustrated on the localization of a submarine robot, named *Redermor*. The experiments have been collected by the GESMA (Groupe d'Etude Sous-Marine de l'Atlantique) in the Douarnenez bay, in Brittany.

1 Introduction

This paper deals with the *simultaneous localization and map building* problem (SLAM) in a submarine context (see [1] for the general SLAM problem). The SLAM problem asks if it is possible for an autonomous robot to move in an unknown environment and build a map of this environment while simultaneously using this map to compute its location.

In this paper, we will show that the SLAM problem can be seen as a *continuous constraints satisfaction problem (CCSP)* (see e.g., [2], [3], [4], [5] for notions related CCSP and applications). Then, we will propose to use a basic constraints propagation algorithm (2B-consistency) to solve the CCSP. The efficiency of the approach will be illustrated on an experiment where an actual underwater vehicle is involved. In this problem, we will try to find an envelope for the trajectory of the robot and to compute sets which contain some detected objects.

Many ideas presented here can be found in [6] and [7] where interval analysis has already been used in the context of SLAM for wheeled robots. But the approach is here made more efficient by the addition of constraints propagation techniques, that have never been used in this context. Note that there exist many other robotics applications where interval constraints propagation methods have been successful (see e.g., [8] for the calibration of robots, [9], [10] for state estimation, [11], [12] for control of robots, [13] for topology analysis of configuration spaces, ...).

The paper is organized as follows. The robot to be considered will first be presented in Section 2. Then, in Section 3, a brief description of the available sensors will be given. By taking into account the state equations of the robot and the interpretation of the sensors, Section 4 will provide the constraints that will make it possible to cast our SLAM problem into a CCSP. The efficiency of our approach will be illustrated on an actual expriment in Section 5.1. Section 6 will then conclude the paper.

2 Robot

The robot to be considered in our application (see Figure 2.1) is an autonomous underwater vehicle (AUV), named *Redermor* (means *greyhound of the sea*, in the Breton language). This robot, developed by the GESMA (Groupe d'Etude Sous-Marine de l'Atlantique), has a length of 6 m, a diameter of 1 m and a weight of 3800 Kg. It has powerful propulsion and control system able to provide hovering capabilities. The main purpose of the *Redermor* is to evaluate improved navigation by the use of sonar information. It is equipped with a KLEIN 5400 side scan sonar which makes it possible to localize objects such as rocks or mines. It also encloses other sophisticated sensors such as a Lock-Doppler to estimate its speed and a gyrocompass to get its three Euler angles (i.e., its orientation).

3 Measurements

3.1 Sensors

The robot is equipped with the following sensors

- **A GPS** (Global Positioning System). A constellation of 24 satellites broadcasts precise timing signals by radio to GPS receivers, allowing them to accurately determine their location (longitude, latitude and altitude) in any weather, day or night, anywhere on the surface of the Earth. However, since electromagnetic waves (here around 1.2 MHz), do not propagate through the water, this sensor is operational only when the robot is at the surface of the ocean, but not when it is inside the water. During our two-hours experiment, using the GPS, the robot is only able to measure the location where it is dropped and the location where it comes back to the surface. Thus, we know that at time $t_0 = 6000$ s, the robot has been dropped approximately around the position

$$\ell^0 = (\ell^0_x, \ell^0_y) = (-4.458227931°, 48.212920614°)\,, \tag{1}$$

where ℓ^0_x is the west/east longitude and ℓ^0_y is the south/north latitude. The error related to this position is less than 2.5 meters. When the robot returns to the surface, at time $t_f = 11999.4$ sec, its position is approximately (*i.e.*, again with an error less than 2.5 meters) given by

$$\ell^f = (\ell^f_x, \ell^f_y) = (-4.454660760°, 48.219129760°)\,. \tag{2}$$

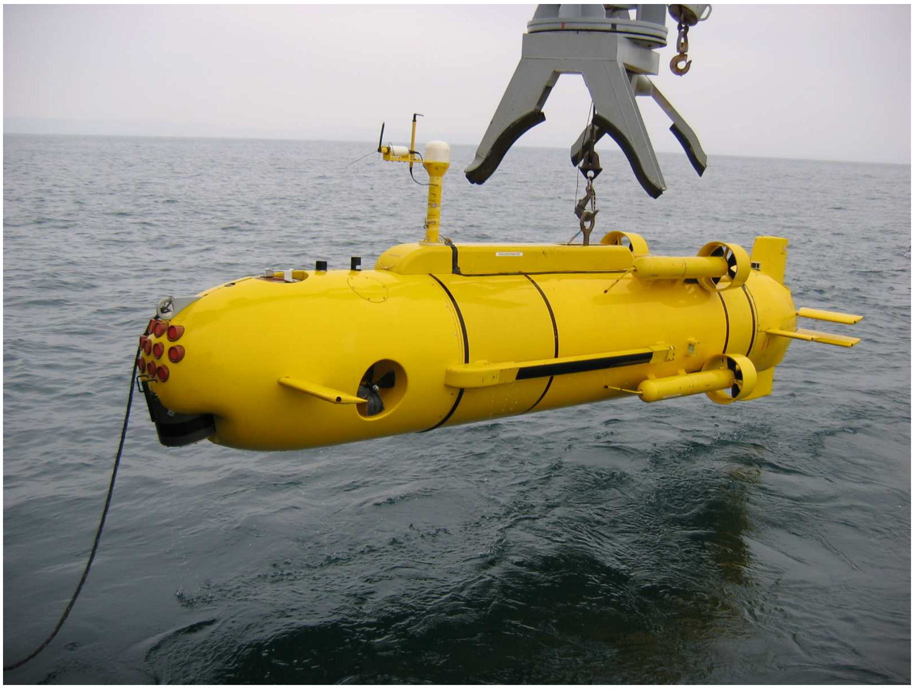


Fig. 2.1. The autonomous underwater vehicle, *Redermor*, built by the GESMA (Groupe d'Etude Sous-Marine de l'Atlantique)

– **A sonar** (KLEIN 5400 side scan sonar). During its mission, the robot detects objects using a sonar located starboard (*i.e.* on its right-hand side). This sonar emits ultrasonic waves to build images such as that represented on Figure 3.1. This image, also called a *waterfall,* is about 75m large for more than 10 km high (corresponding to the length covered by the robot during its mission). After the mission, a scrolling of the waterfall is performed by a human operator which is then able to perform an estimation $\tilde{r}(t)$ of the distance $r(t)$ from the robot to an object detected at time t. Recall that the positions of the objects are assumed to be unknown. From the width of the black vertical band on the left of the picture (called the *water column*), we are also able to compute an estimation $\tilde{a}(t)$ of the altitude $a(t)$ of the robot (distance between the robot and the bottom). Figure 3.1 is related to the detection of the 5th object in the case of the mission made by the robot. The associated ping is represented by the thin white rectangle. Up to now, the detection of an object and the matching between objects are performed manually, from a scrolling of the waterfall, once the robot has accomplished its mission. But we are planning to develop an automatic and reliable procedure for this task.

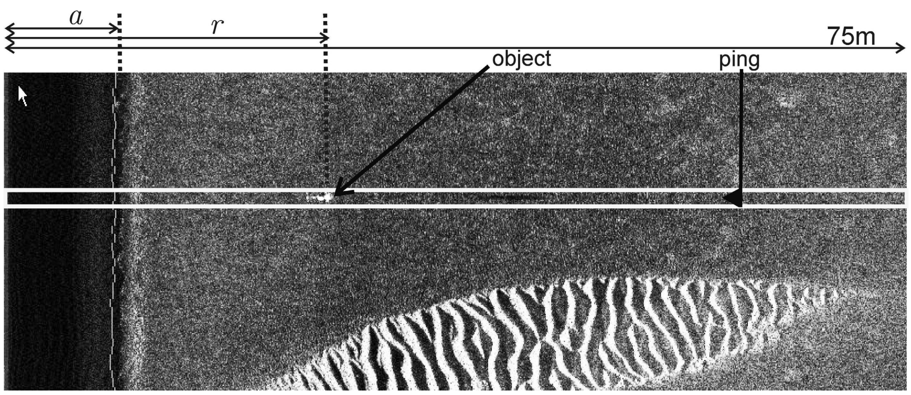


Fig. 3.1. The sonar image makes it possible to detect an object, to compute the distance r between the object and the robot, and the altitude a of the robot

Table 3.1. Measurements related to the objects detected by the sonar

i	0	1	2	3	4	5	6	7	8	9	10	11
$\tau(i)$	7054	7092	7374	7748	9038	9688	10024	10817	11172	11232	11279	11688
$\sigma(i)$	1	2	1	0	1	5	4	3	3	4	5	1
$\tilde{r}(i)$	52.42	12.47	54.40	52.68	27.73	26.98	37.90	36.71	37.37	31.03	33.51	15.05

– **A Loch-Doppler.** This sensor makes it possible to compute the speed of the robot $\mathbf{v}_r$ and returns it in the robot frame. The Lock-Doppler emits ultrasonic waves which are reflected on the bottom of the ocean. Since the bottom is immobile, this sensor is able to compute an estimation of its speed using the Doppler effect. When the frequency of the waves is around 300 kHz, then the actual speed is known to satisfy

$$\mathbf{v}_r \in \tilde{\mathbf{v}}_r + 0.004 * [-1,1]\,.\tilde{\mathbf{v}}_r + 0.004 * [-1,1]. \tag{3}$$

where $\tilde{\mathbf{v}}_r$ denotes the three dimensional speed vector returned by the sensor. The Loch-Doppler is also able to provide the altitude a of the robot with an error less than 10cm.

– **A Gyrocompass** (Octans III from IXSEA). This sensor uses the Sagnac effect and the rotation of the earth to compute the three Euler angles (the roll ϕ, the pitch θ, and the head ψ) of the robot with a high accuracy. If we denote by $\tilde{\phi}, \tilde{\theta}, \tilde{\psi}$, the angles returned by our gyrocompass, then the actual Euler angles for our robot should satisfy

$$\begin{pmatrix} \phi \\ \theta \\ \psi \end{pmatrix} \in \begin{pmatrix} \tilde{\phi} \\ \tilde{\theta} \\ \tilde{\psi} \end{pmatrix} + \begin{pmatrix} 1.75 \times 10^{-4}.\,[-1,1] \\ 1.75 \times 10^{-4}.\,[-1,1] \\ 5.27 \times 10^{-3}.\,[-1,1] \end{pmatrix}. \tag{4}$$

– **A barometer** is used to compute the depth of the robot (i.e., the distance between the robot and the surface of the ocean). If $\tilde{d}$ is the depth collected by the sensor, then the actual depth $p_z(t)$ of the robot satisfies $p_z(t) \in [-1.5, 1.5] + \tilde{d}.[0.98, 1.02]$. The interval $[-1.5, 1.5]$ may change depending on the strength of waves and tides.

3.2 Measurements

For each time $t \in \mathcal{T} \overset{\text{def}}{=} \{6000.0, 6000.1, 6000.2, \dots, 11999.4\}$, the vector of measurements

$$\tilde{\mathbf{u}}(t) = \left(\tilde{\phi}(t), \tilde{\theta}(t), \tilde{\psi}(t), \tilde{v}_r^x(t), \tilde{v}_r^y(t), \tilde{v}_r^z(t), \tilde{a}(t), \tilde{d}(t)\right), \tag{5}$$

is collected. Using the characteristics of the sensors, it is possible to get a box $[\mathbf{u}(t)]$ which contains the actual value for the vector

$$\mathbf{u}(t) = (\phi(t), \theta(t), \psi(t), v_r^x(t), v_r^y(t), v_r^z(t), a(t), p_z(t)), \tag{6}$$

for each $t \in \mathcal{T}$.

Moreover, six objects have been detected manually from the sonar waterfall (i.e. the sonar image) collected by the robot. Table 3.1, provides (i) the number i of the ping where an object has been detected starboard, (ii) the corresponding time $\tau(i)$, (iii) the number $\sigma(i)$ of the detected object, and (iv) a measure $\tilde{r}(i)$ of the distance between the robot and the object. The actual distance $r(i)$ between the robot and the object for the ith ping is supposed to satisfy the relation

$$r(i) \in [\tilde{r}(i) - 1, \tilde{r}(i) + 1]. \tag{7}$$

4 Constraints

Around the zone covered by the robot, let us build the frame $(\mathbf{O}, \overrightarrow{\mathbf{i}}, \overrightarrow{\mathbf{j}}, \overrightarrow{\mathbf{k}})$ where $\mathbf{O}$ is the location of the robot at time $t_0 = 6000$s, the vector $\overrightarrow{\mathbf{i}}$ indicates the north, $\overrightarrow{\mathbf{j}}$ indicates the east and $\overrightarrow{\mathbf{k}}$ is oriented toward the center of the earth. Denote by $\mathbf{p} = (p_x, p_y, p_z)$ the coordinates of the robot expressed in the frame $(\mathbf{O}, \overrightarrow{\mathbf{i}}, \overrightarrow{\mathbf{j}}, \overrightarrow{\mathbf{k}})$. From the latitude and the longitude, given by the GPS, we can deduce the two first coordinates of the robot using the following relation:

$$\begin{pmatrix} p_x \\ p_y \end{pmatrix} = 111120 * \begin{pmatrix} 0 & 1 \\ \cos\left(\ell_y * \frac{\pi}{180}\right) & 0 \end{pmatrix} \begin{pmatrix} \ell_x - \ell_x^0 \\ \ell_y - \ell_y^0 \end{pmatrix}. \tag{8}$$

Moreover, the robot motion can be described by the following differential equation (also called state equation)

$$\dot{\mathbf{p}}(t) = \mathbf{R}(\phi(t), \theta(t), \psi(t)).\mathbf{v}_r(t), \tag{9}$$

where

$$\mathbf{R}(\phi, \theta, \psi) = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix}$$

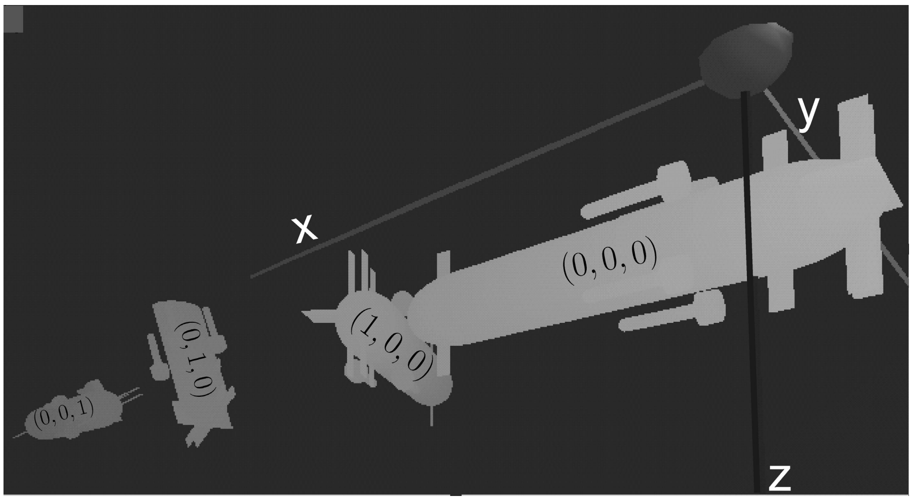


Fig. 4.1. The *Redermor* for different configurations (ψ, θ, ϕ)

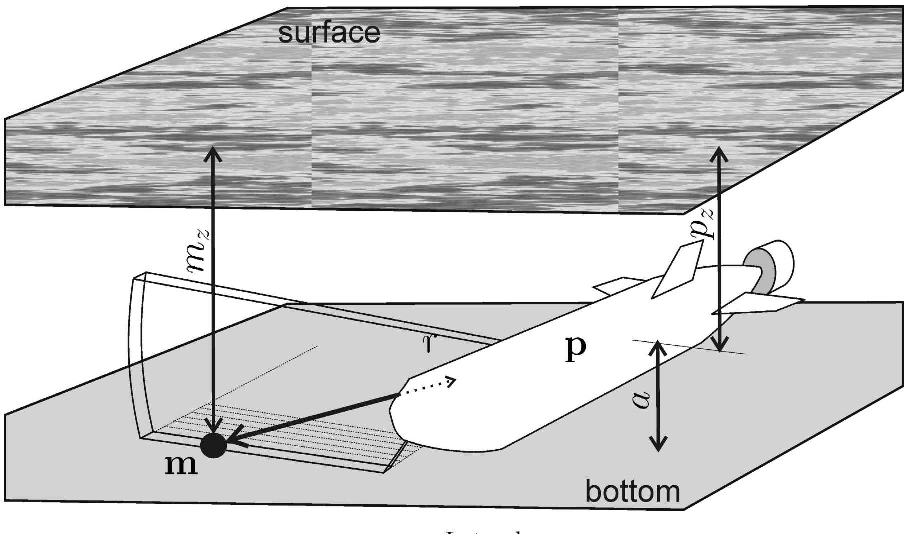


Fig. 4.2. The distance from the robot to the detected object can be obtained using a lateral sonar

and $\mathbf{v}_r$ represents the speed of the robot measured by the Loch-Doppler sensor (see Equation (3)). Figure 4.1 gives an illustration of the meaning of the angles

ψ, θ, ϕ. From the right to the left, we have (ψ, θ, ϕ) equal to $(0,0,0)$, $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$.

The state equation (9) can be interpreted as a constraint between the five functions $\dot{\mathbf{p}}(.), \phi(.), \theta(.), \psi(.)$ and $\mathbf{v}_r(.)$. Although this type of constraints could be handled inside a constraint propagation formalism [14,15], for simplicity, we shall approximate this constraint between functions by a constraint between variables by resorting to a discretrization. This operation makes it possible to cast our problem into a classical CSP over continuous domains, but it removes the compleness of our approach. Since the sampling time is given by $\delta = 0.1$s, an Euler discretization of the state equation (9) yields

$$\mathbf{p}(t+0.1) = \mathbf{p}(t) + 0.1 * \mathbf{R}(\phi(t), \theta(t), \psi(t)).\mathbf{v}_r(t). \tag{10}$$

When the ith object is detected at time $t = \tau(i)$ (see Table 3.1), it is located starboard of the robot and on a plane which is perpendicular to the robot axis (see Figure 4.2). The associated constraints are

$$\begin{cases} \text{(i)} & \|\mathbf{m}(\sigma(i)) - \mathbf{p}(t)\| = r(i) \\ \text{(ii)} & \mathbf{R}^{\mathrm{T}}(\phi(t), \theta(t), \psi(t)).\,(\mathbf{m}(\sigma(i)t) - \mathbf{p}(t)) \in [0,0] \times [0,\infty] \times [0,\infty] \\ \text{(iii)} & m_z(\sigma(i)) - p_z(t) - a(t) \in [-0.5, 0.5] \end{cases} \tag{11}$$

where, $\mathbf{m}(\sigma(i))$ represents the location of the ith object and $\mathbf{R}^{\mathrm{T}}(\phi, \theta, \psi).\,(\mathbf{m} - \mathbf{p})$ represents the vector $\mathbf{m} - \mathbf{p}$ expressed in the robot frame. In the constraint (ii), the first interval $[0,0]$ means that the vector $\mathbf{m} - \mathbf{p}$ is perpendicular to the main axis of the robot, the second interval $[0,\infty]$ indicates that the object is starboard and the third interval $[0,\infty]$ indicated that the object is deeper than the robot itself. If we assume that the bottom of the ocean is flat, then we should have $m_z(\sigma(i)) = p_z(t) + a(t)$ (i.e., the depth m_z of the object lying on the bottom is equal to the altitude a of the robot plus the depth p_z of the robot). The constraint (iii) translates this relation with a small uncertainty represented by the interval $[-0.5, 0.5]$. This assumption is true if the slope of the (almost flat) bottom is limited to $\frac{0.5}{75} = 0.7\%$, which is true in the bottom the Douarnenez bay.

5 Results

5.1 Constraints Satisfaction Problem

Our SLAM problem can be cast into the following constraints satisfaction problem.

$$\begin{cases}
t \in \{6000.0, 6000.1, 6000.2, \ldots, 11999.4\}, \quad i \in \{0, 1, \ldots, 11\}, \\
\begin{pmatrix} p_x(t) \\ p_y(t) \end{pmatrix} = 111120. \begin{pmatrix} 0 & 1 \\ \cos\left(\ell_y(t) * \frac{\pi}{180}\right) & 0 \end{pmatrix} \begin{pmatrix} \ell_x(t) - \ell_x^0 \\ \ell_y(t) - \ell_y^0 \end{pmatrix}, \\
\mathbf{p}(t) = (p_x(t), p_y(t), p_z(t)), \\
\mathbf{R}_\psi(t) = \begin{pmatrix} \cos\psi(t) & -\sin\psi(t) & 0 \\ \sin\psi(t) & \cos\psi(t) & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{R}_\theta(t) = \begin{pmatrix} \cos\theta(t) & 0 & \sin\theta(t) \\ 0 & 1 & 0 \\ -\sin\theta(t) & 0 & \cos\theta(t) \end{pmatrix}, \\
\mathbf{R}_\varphi(t) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi(t) & -\sin\varphi(t) \\ 0 & \sin\varphi(t) & \cos\varphi(t) \end{pmatrix}, \quad \mathbf{R}(t) = \mathbf{R}_\psi(t)\mathbf{R}_\theta(t)\mathbf{R}_\varphi(t), \\
\mathbf{p}(t + 0.1) = \mathbf{p}(t) + 0.1 * \mathbf{R}(t).\mathbf{v}_r(t), \\
||\mathbf{m}(\sigma(i)) - \mathbf{p}(\tau(i))|| \ = r(i), \\
\mathbf{R}^{\mathrm{T}}(\tau(i))\,(\mathbf{m}(\sigma(i)) - \mathbf{p}(\tau(i))) \in [0, 0] \times [0, \infty] \times [0, \infty], \\
m_z(\sigma(i)) - p_z(\tau(i)) - a(\tau(i)) \in [-0.5, 0.5]
\end{cases}$$

These constraints involve more than 300 000 variables (if a scalar decomposition of the vectors in performed). The sensors (GPS, sonar, ...) make it possible to get some accurate domains for the variables $\phi(t)$, $\theta(t)$, $\psi(t)$, $\mathbf{v}_r(t)$, $a(t)$, $p_z(t)$, $\ell_x(6000)$, $\ell_y(6000)$, $\ell_x(11999.4)$, $\ell_y(11999.4)$. The other variables $\ell_x(6000.1), \ldots,$ $\ell_x(11999.3), \ell_y(6000.1), \ldots, \ell_y(11999.3), p_x(6000), \ldots, p_x(11999.4), p_y(6000), \ldots,$ $p_y(11999.4), \mathbf{m}(0), \ldots, \mathbf{m}(5)$ are unknown and the domains for their components should initially be instantiated to $[-\infty, \infty]$.

A constraints propagation procedure could thus be thought to contract all domains of our CCSP. Since we want to get accurate results, a scalar decomposition of the matrix constraints involved in our CSP is not recommended. Instead, we have developed efficient contraction algorithms associated to all our matrix constraints, such as $\mathbf{A} = \mathbf{B} * \mathbf{C}$, $||\mathbf{v}|| = r, \ldots$. An illustration is given by the following example.

Example: To contract the constraint $\mathbf{R}(t) = \mathbf{R}_\psi(t)\mathbf{R}_\theta(t)\mathbf{R}_\varphi(t)$ involved in our CSP, we can take into account the fact that the matrices are all rotation matrices (i.e., their inverse is equal to their transpose). From this constraint, we can built other matrix constraints, as follows

$$\begin{aligned}
\mathbf{R}(t) = \mathbf{R}_\psi(t)\mathbf{R}_\theta(t)\mathbf{R}_\varphi(t) &\Leftrightarrow \mathbf{R}(t)\mathbf{R}_\varphi^{\mathrm{T}}(t) = \mathbf{R}_\psi(t)\mathbf{R}_\theta(t) \\
&\Leftrightarrow \mathbf{R}(t)\mathbf{R}_\varphi^{\mathrm{T}}(t)\mathbf{R}_\theta^{\mathrm{T}}(t) = \mathbf{R}_\psi(t) \Leftrightarrow \ldots.
\end{aligned}$$

From all these generated redundant constraints; one can built contractors by decomposing them into scalar constraints and by using a hull consistency procedure. The resulting procedure constitutes an efficient contractor for the constraint $\mathbf{R}(t) = \mathbf{R}_{\psi}(t)\mathbf{R}_{\theta}(t)\mathbf{R}_{\varphi}(t)$.

5.2 Propagation

The results obtained by an elementary constraints propagation algorithm (similar to hull consistency) are illustrated by Figure 5.1. Subfigure (a) represents a punctual estimation of the trajectory of the robot. This estimation has been obtained by integrating the state equations (9) from the initial point (represented on lower part). We have also represented the 6 objects that have been dropped manually at the bottom of the ocean during the experiments. Note that we are not supposed to know the location of these six object. When we dropped them, we measured their location, but we used this information only to check the consistency of results obtained by the propagation. Subfigure (b) represents an envelope of the trajectory obtained using an interval integration, from a small initial box, obtained by the GPS at the beginning of the mission. In Subfigure (c) a final GPS point has also been considered and a forward-backward propagation has been performed up to equilibrium. In Figure (d) the constraints involving the object have been considered for the propagation. The envelope is now thinner and enveloping boxes containing the objects have also been obtained (see Subfigure (e)). We have checked that the actual positions for the objects (that have been measured independently during the experiments) all belong to the associated box, painted black. In Subfigure (f), a zooming perspective of the trajectory and the enveloping boxes for the detected objects have been represented. The computing time to get all these envelopes is less than one minute with a Pentium III. About ten forward-backward interval propagations have been performed to get the steady box of the CCSP. The C++ code associated with this example as well as a windows executable program can be downloaded at

`http://www.ensieta.fr/e3i2/Jaulin/redermorcp06.zip`

In the case where the position of the objects is approximately known, the SLAM problem translates into a state estimation problem. The structure of the CSP becomes a tree [9] and it is possible to get the global consistency with only one forward and one backward propagation. The envelope for the trajectory becomes very thin and a short computation time is needed. The capabilities of interval propagation methods for state estimation in a bounded error context have already been demonstrated in several applications (see e.g., [16], [17], [8] [9]). For the SLAM problem, the graph of the CSP is not a tree anymore. Of course, the number of cycles of the graph is rather limited, and a large part of the graph is made with one huge tree resulting from the state space equations. Because of these cycles, the global consistency cannot be reached without any bissection. Now, the number of variables of our CSP is huge and we should give up the idea of reaching the global consistency via bissections. On the other hand, redundant constraints can easily be obtained by adding other sensors

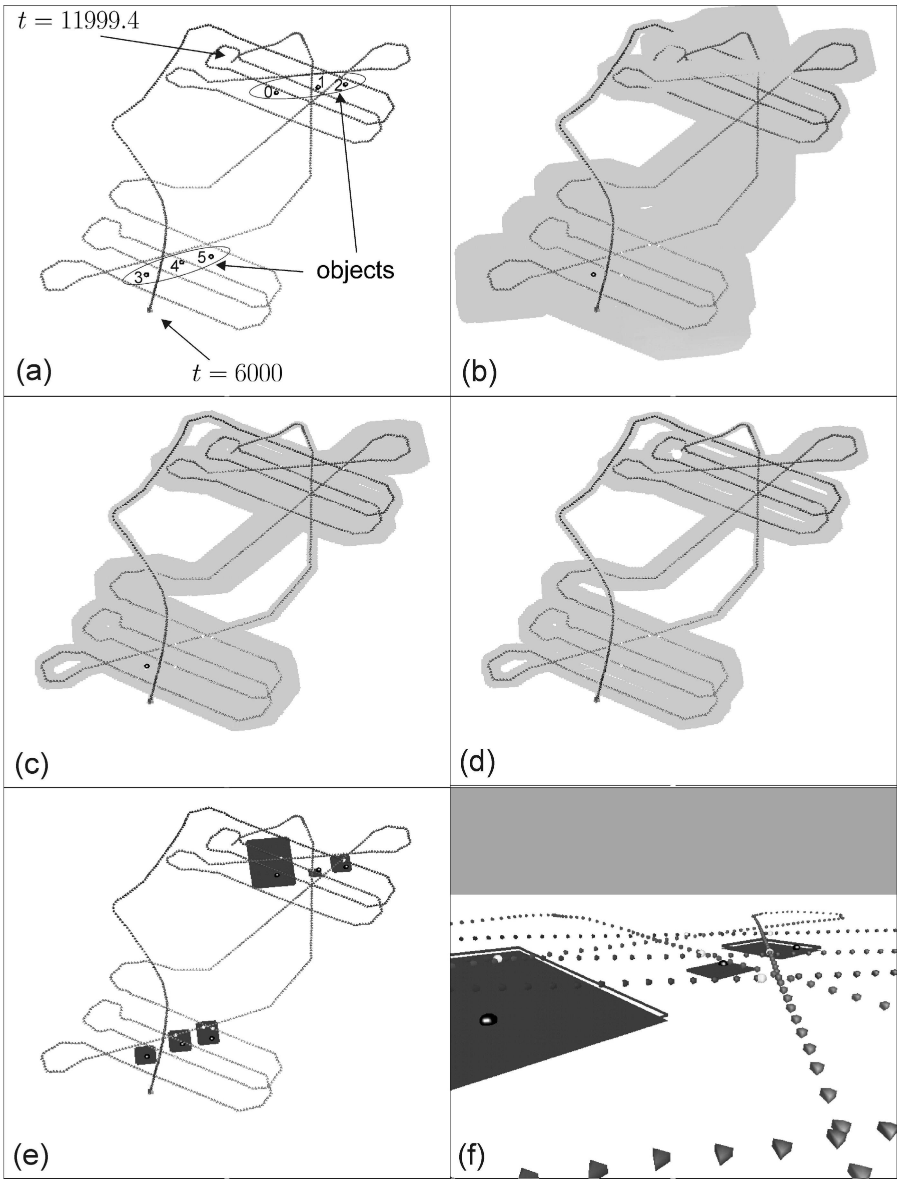


Fig. 5.1. Results obtained by our constraints propagation method

or by handling the existing constraints, in a symbolic way. Adding redundant constraints could thus be a realistic way to control the accuracy of an interval contraints propagation method for the SLAM problem.

6 Conclusion

In this paper, we have shown that interval constraints propagation could be applied to solve SLAM problems. The efficiency of the approach has been demonstrated on an experiment made with an actual underwater robot (the *Redermor*). The experiment lasted two hours and involved thousands of data. if all assumptions on the bounds of the sensors, on the flat bottom, on the model of the robot, ... are satisfied, then their exists always at least one solution of the CSP: that corresponding to the actual trajectory of the robot. Thus their is no need to prove the existence of a solution. Since the CSP has more variables than equations, the solution set contains generally a continuum of points.

When outliers occur during the experiment, our approach is not reliable anymore and one should take care about any false interpretation of the results. Consider now three different situation that should be known by any user of our approach for SLAM.

Situation 1. The solution set is empty and an empty set is returned by the propagation procedure. Our approach detects that their exists at least one outlier but it is not able to return any estimation of the trajectory and the positions of the objects. It is also not able to detect which sensor is responsible for the failure.

Situation 2. The solution set is empty but nonempty thin intervals for the variables are returned by the propagation. Our approach is not efficient enough to detect that outliers exist and we can wronly interpret that an accurate and guaranteed estimation of the trajectory of the robot has been done. Other more efficient algorithms could be able to prove that no solution exists which would lead us to the situation 1.

Situation 3. The solution set is not empty but it does not contain the actual trajectory of the robot. No method could be able to prove that outliers occur. Again, our approach could lead us to the false conclusion that a guaranteed estimation of the trajectory of the robot has been done, whereas, the robot might be somewhere else.

Now, for our experiment made on the Redermor, it is clear that outliers might be present. We have observed that when we corrupt some data volontarily (to create ouliers), the propagation method usually returns rapidly that no solution exists for our set of constraints. For our experiment with the data collected, we did not obtain an empty set. The only thing that we can conclude is that no outlier has been detected. The constraints propagation method can thus be seen a tool to validate (or unvalidate) reliability on models and sensor bounds.

References

1. Leonard, J.J., Durrant-Whyte, H.F.: Dynamic map building for an autonomous mobile robot. International Journal of Robotics Research **11**(4) (1992)
2. van Hentenryck, P., Deville, Y., Michel, L.: Numerica: A Modeling Language for Global Optimization. MIT Press, Boston, MA (1997)

3. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: Proceedings of the International Conference on Logic Programming, Las Cruces, NM (1999) 230–244
4. Sam-Haroud, D.: Constraint consistency techniques for continuous domains. PhD dissertation 1423, Swiss Federal Institute of Technology in Lausanne, Switzerland (1995)
5. van Emden, M.: Algorithmic power from declarative use of redundant constraints. Constraints **4**(4) (1999) 363–381
6. Drocourt, C., Delahoche, L., E. Brassart, B.M., Clerentin, A.: Incremental construction of the robot's environmental map using interval analysis. Global Optimization and Constraint Satisfaction: Second International Workshop, COCOS 2003 **3478** (2005) 127–141
7. Porta, J.: Cuikslam: A kinematics-based approach to slam. In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona (Spain) (2005) 2436–2442
8. Baguenard, X.: Propagation de contraintes sur les intervalles. Application l'talonnage des robots. PhD dissertation, Université d'Angers, Angers, France (2005) Available at: www.istia.univ-angers.fr/˜baguenar/.
9. L. Jaulin, M. Kieffer, O. Didrit, E. Walter: Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics. Springer-Verlag, London (2001)
10. Raissi, R., Ramdani, N., Candau, Y.: Set membership state and parameter estimation for systems described by nonlinear differential equations. Automatica **40** (2004) 1771–1777
11. Lydoire, F., Poignet, P.: Nonlinear predictive control using constraint satisfaction. In: In 2nd International Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS). (2003) 179–188
12. Vinas, P.H., Sainz, M.A., Vehi, J., Jaulin, L.: Quantified set inversion algorithm with applications to control. Reliable computing **11**(5) (2006) 369–382
13. Delanoue, N., Jaulin, L., Cottenceau, B.: Using interval arithmetic to prove that a set is path-connected. Theoretical Computer Science, Special issue: Real Numbers and Computers **351**(1) (2006) 119–128
14. Janssen, M., Hentenryck, P.V., Deville, Y.: A constraint satisfaction approach for enclosing solutions to parametric ordinary differential equations. SIAM Journal on Numerical Analysis **40**(5) (2002) 1896–1939
15. Jaulin, L.: Nonlinear bounded-error state estimation of continuous-time systems. Automatica **38** (2002) 1079–1082
16. Gning, A.: Localisation garantie d'automobiles. Contribution aux techniques de satisfaction de contraintes sur les intervalles. PhD dissertation, Universit de Technologie de Compigne, Compigne, France (2006)
17. Bouron, P.: Méthodes ensemblistes pour le diagnostic, l'estimation d'état et la fusion de données temporelles. PhD dissertation, Université de Compiègne, Compiègne, France (2002)

Approximability of Integer Programming with Generalised Constraints

Peter Jonsson*, Fredrik Kuivinen**, and Gustav Nordh***

Department of Computer and Information Science
Linköpings Universitet
S-581 83 Linköping, Sweden
{petej, freku, gusno}@ida.liu.se

Abstract. We study a family of problems, called MAXIMUM SOLUTION, where the objective is to maximise a linear goal function over the feasible integer assignments to a set of variables subject to a set of constraints. This problem is closely related to INTEGER LINEAR PROGRAMMING. When the domain is Boolean (i.e. restricted to $\{0, 1\}$), the maximum solution problem is identical to the well-studied MAX ONES problem, and the approximability is completely understood for all restrictions on the underlying constraints. We continue this line of research by considering domains containing more than two elements. We present two main results: a complete classification for the approximability of all maximal constraint languages, and a complete classification of the approximability of the problem when the set of allowed constraints contains all permutation constraints. Our results are proved by using algebraic results from clone theory and the results indicates that this approach is very useful for classifying the approximability of certain optimisation problems.

1 Introduction

Combinatorial optimisation problems can often be formulated as integer linear programs (here after abbreviated as ILP). In its most general form the aim in a ILP is to assign integers to a set of variables such that a set of linear inequalities are satisfied and a linear goal function is maximised or minimised. In this general form of the problem it is **NP**-hard to find feasible solutions [12]. It is well-known that restricted versions of the general ILP problem still has the ability to express many real-world optimisation problems. One such restriction is to only consider solutions consisting of 0 and 1, i.e., the domain is $\{0, 1\}$. This problem, commonly called MAXIMUM 0-1 PROGRAMMING, is still very hard, in fact it is **NPO**-complete [17]. It is also well-known that if certain restrictions are

* Partially supported by the *Center for Industrial Information Technology* (CENIIT) under grant 04.01, and by the *Swedish Research Council* (VR) under grant 621-2003-3421.

** Supported by the *Swedish Research Council* (VR) under grant 621-2002-4126.

*** Supported by the *National Graduate School in Computer Science* (CUGS), Sweden.

imposed on the constraint matrix, then the corresponding ILP becomes computationally easier. Two examples are totally unimodular constraint matrices [23] and matrices containing at most two non-zero entries per row [13].

In this paper, we study a variant of the ILP problem where we allow arbitrary finite sets of relations as our constraints. Since the set of allowed relations is finite, it follows that the arity of constraints is bounded (which is not the case for ILP). Our goal is to classify the complexity of obtaining approximate solutions to this problem for all different sets of allowed constraints. We approach this problem from the constraint satisfaction angle. A wide range of combinatorial problems can be viewed as 'constraint satisfaction problems' (CSPs), in which the aim is to find an assignment of values to a set of variables subject to certain constraints. Typical examples include the satisfiability problem, graph colourability problems and many others.

Let us now formally define the problem that we will study: Let $D \subset \mathbb{N}$ (*the domain*) be a finite set. The set of all n-tuples of elements from D is denoted by D^n. Any subset of D^n is called an n-ary relation on D. The set of all finitary relations over D is denoted by R_D. A constraint language over a finite set, D, is a finite set $\Gamma \subseteq R_D$. Constraint languages are the way in which we specify restrictions on our problems. The constraint satisfaction problem over the constraint language Γ, denoted CSP(Γ), is defined to be the decision problem with instance (V, D, C), where

- V is a set of variables,
- D is a finite set of values (sometimes called a domain), and
- C is a set of constraints $\{C_1, \ldots, C_q\}$, in which each constraint C_i is a pair (s_i, ϱ_i) where s_i is a list of variables of length m_i, called the constraint scope, and ϱ_i is an m_i-ary relation over the set D, belonging to Γ, called the constraint relation.

The question is whether there exists a solution to (V, D, C) or not, that is, a function from V to D such that, for each constraint in C, the image of the constraint scope is a member of the constraint relation.

The optimisation problem that we are going to study, WEIGHTED MAXIMUM SOLUTION, can then be defined as follows: WEIGHTED MAXIMUM SOLUTION over the constraint language Γ, denoted W-MAX SOL(Γ), is defined to be the optimisation problem with

Instance: Tuple (V, D, C, w), where D is a finite subset of $\mathbb{N}$, (V, D, C) is a CSP(Γ) instance, and $w : V \to \mathbb{N}$ is a weight function.

Solution: An assignment $f : V \to D$ to the variables such that all constraints are satisfied.

Measure: $\sum_{v \in V} w(v) \cdot f(v)$

The problem W-MAX SOL should not be confused with the MAX CSP problem where the objective is to maximise the number of satisfied constraints.

W-MAX SOL restricted to Boolean domains is known as WEIGHTED MAX ONES and the approximability of (WEIGHTED) MAX ONES is completely understood for all constraint languages [20]: For any Boolean constraint language Γ, W-Max Sol(Γ) is either in **PO** or is **APX**-complete or **poly-APX**-complete or finding a solution of non-zero value is **NP**-hard or finding any solution is **NP**-hard. The exact borderlines between the different cases are given in [20].

While the approximability of W-Max Sol is well-understood for the Boolean domain, this is not the case for larger domains. For larger domains we are aware of two results: the first one is a tight (in)approximability result for linear equations over $\mathbb{Z}_p$ [21] and the second result is for so-called monotone constraints [13]. In this paper we show how the algebraic approach for CSPs [5,15] can be used to study the approximability of W-Max Sol. The algebraic approach has been very successful: it has, for instance, made it possible to design new efficient algorithms and to clarify the borderline between tractability and intractability in many important cases. In particular, the complexity of the CSP problem over three element domains is now completely understood [2]. By using this approach we are able to present the following two main results:

Result 1. We completely characterise the approximability of *maximal* constraint languages; a constraint language Γ is maximal if, for any $r \notin \Gamma$, $\Gamma \cup \{r\}$ has the ability to express (in a sense to be formally defined later on) every relation in R_D. Such languages have attracted much attention lately [3,6]. Our results shows that if Γ is maximal, then W-Max Sol(Γ) is either tractable, **APX**-complete, **poly-APX**-complete, finding any solution with non-zero measure is **NP**-hard, or CSP(Γ) is not tractable. The different cases can also be efficiently recognised given an arbitrary maximal constraint language.

Result 2. We completely characterise the approximability of W-Max Sol(Γ) when Γ contains all permutation constraints. Such languages are known as *homogenous* languages and Dalmau [10] has determined the complexity of CSP(Γ) for all such languages. We show that W-Max Sol(Γ) is either tractable, **APX**-complete, **poly-APX**-complete, or CSP(Γ) is not tractable.

When proving **Result 1**, we identified a new large tractable class of W-Max Sol(Γ): *generalised max-closed* constraints. This class (which may be of independent interest) significantly extend some of the tractable classes of Max Ones that were identified by Khanna et al.

The paper is structured as follows: Section 2 contains some basics on approximability and the algebraic approach to CSPs and Section 3 identifies certain hard constraint languages. Section 4 contains some tractability results, Section 5 contains **Result 1**, and Section 6 contains **Result 2**. Section 7 contains some final remarks. All the proofs omitted due to space limitations can be found in the technical report version of the paper [19].

2 Preliminaries

In this section we state some preliminaries which we will need throughout the paper. For a relation R with arity a we will sometimes write $R(x_1, \ldots, x_a)$ with the meaning $(x_1, \ldots, x_a) \in R$. Furthermore, the constraint $((x_1, \ldots, x_a), R)$ will sometimes be written as $R(x_1, \ldots, x_a)$. The intended meaning will be clear from the context.

2.1 Approximability, Reductions, and Completeness

A *combinatorial optimisation problem* is defined over a set of *instances* (admissible input data); each instance I has a finite set $\mathsf{sol}(I)$ of *feasible solutions* associated with it. The objective is, given an instance I, to find a feasible solution of *optimum* value with respect to some measure function $m : \mathsf{sol}(I) \to \mathbb{N}$. The optimal value is the largest one for *maximisation* problems and the smallest one for *minimisation* problems. A combinatorial optimisation problem is said to be an **NPO** problem if its instances and solutions can be recognised in polynomial time, the solutions are polynomially bounded in the input size, and the objective function can be computed in polynomial time (see, e.g., [1]).

We say that a solution $s \in \mathsf{sol}(I)$ to an instance I of an **NPO** problem Π is r-approximate if it satisfies $\max\left\{\frac{m(s)}{\text{OPT}(I)}, \frac{\text{OPT}(I)}{m(s)}\right\} \leq r$, where $\text{OPT}(I)$ is the optimal value for a solution to I. An approximation algorithm for an **NPO** problem Π has *performance ratio* $\mathcal{R}(n)$ if, given any instance I of Π with $|I| = n$, it outputs an $\mathcal{R}(n)$-approximate solution.

Let **PO** denote the class of **NPO** problems that can be solved (to optimality) in polynomial time. An **NPO** problem Π is in the class **APX** if there is a polynomial-time approximation algorithm for Π whose performance ratio is bounded by a constant. Similarly, Π is in the class **poly-APX** if there is a polynomial-time approximation algorithm for Π whose performance ratio is bounded by a polynomial in the size of the input.

Completeness in **APX** and **poly-APX** is defined using an appropriate reduction, called AP-reduction [9,20]: An **NPO** problem Π_1 is said to be *AP-reducible* to an **NPO** problem Π_2 if two polynomial-time computable functions F and G and a constant α exist such that (1) for any instance I of Π_1, $F(I)$ is an instance of Π_2; (2) for any instance I of Π_1, and any feasible solution s' of $F(I)$, $G(I, s')$ is a feasible solution of I; and (3) for any instance I of Π_1, and any $r \geq 1$, if s' is an r-approximate solution of $F(I)$ then $G(I, s')$ is an $(1 + (r-1)\alpha + o(1))$-approximate solution of I where the $o(1)$-notation is with respect to $|I|$. An **NPO** problem Π is **APX**-*hard* (**poly-APX**-*hard*) if every problem in **APX** (**poly-APX**) is AP-reducible to it. If, in addition, Π is in **APX** (**poly-APX**), then Π is called **APX**-*complete* (**poly-APX**-*complete*).It is a well-known fact (see, e.g., Section 8.2.1 in [1]) that AP-reductions compose.

We will sometime use another kind of reductions known as S-reductions. They are defined as follows: An **NPO** problem Π_1 is said to be *S-reducible* to an **NPO** problem Π_2 if two polynomial-time computable functions F and G exist such that (1) given any instance I of Π_1, algorithm F produces an instance $I' = F(I)$ of Π_2, such that the measure of an optimal solution for I', $\text{OPT}(I')$, is exactly

$\text{OPT}(I)$; and (2) given $I' = F(I)$, and any solution s' to I', algorithm G produces a solution s to I such that $m(G(s')) = m'(s')$.

Obviously, the existence of an S-reduction from Π_1 to Π_2 implies the existence of an AP-reduction from Π_1 to Π_2. The reason why we need S-reductions is that AP-reductions do not (generally) preserve membership in **PO** [20].

2.2 Algebraic Approach to CSPs

An operation on a finite set D (the domain) is an arbitrary function $f : D^k \to D$. Any operation on D can be extended in a standard way to an operation on tuples over D, as follows: Let f be a k-ary operation on D and let R be an n-ary relation over D. For any collection of k tuples, $\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_k} \in R$, the n-tuple $f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_k})$ is defined as follows: $f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_k}) = (f(\boldsymbol{t_1}[1], \boldsymbol{t_2}[1], \ldots, \boldsymbol{t_k}[1]), f(\boldsymbol{t_1}[2], \boldsymbol{t_2}[2], \ldots, \boldsymbol{t_k}[2]), \ldots, f(\boldsymbol{t_1}[n], \boldsymbol{t_2}[n], \ldots, \boldsymbol{t_k}[n]))$, where $\boldsymbol{t_j}[i]$ is the i-th component in tuple $\boldsymbol{t_j}$. A technique that has proved to be useful in determining the computational complexity of $\text{CSP}(\Gamma)$ is that of investigating whether Γ is invariant under certain families of operations [15].

Let $\varrho_i \in \Gamma$. If f is an operation such that for all $\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_k} \in \varrho_i$, it holds that $f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_k}) \in \varrho_i$, then we say that ϱ_i is *invariant* (or, in other words, closed) under f. If all constraint relations in Γ are invariant under f then Γ is invariant under f. An operation f such that Γ is invariant under f is called a polymorphism of Γ. The set of all polymorphisms of Γ is denoted $Pol(\Gamma)$. Given a set of operations F, the set of all relations that is invariant under all the operations in F is denoted $Inv(F)$.

We continue by defining a closure operation $\langle \cdot \rangle$ on sets of relations: for any set $\Gamma \subseteq R_D$ the set $\langle \Gamma \rangle$ consists of all relations that can be expressed using relations from $\Gamma \cup \{=_D\}$ ($=_D$ is the equality relation on D), conjunction, and existential quantification. Intuitively, constraints using relations from $\langle \Gamma \rangle$ are exactly those which can be simulated by constraints using relations from Γ. The sets of relations of the form $\langle \Gamma \rangle$ are referred to as relational clones, or co-clones. An alternative characterisation of relational clones is the following [22]: for every set $\Gamma \subseteq R_D$, $\langle \Gamma \rangle = Inv(Pol(\Gamma))$. The next theorem states that when we are studying the approximability of W-MAX SOL(Γ) it is sufficient to consider constraint languages that are relational clones.

Theorem 1. *Let Γ be a finite constraint language and $\Gamma' \subseteq \langle \Gamma \rangle$ finite. Then* W-MAX SOL(Γ') *is S-reducible to* W-MAX SOL(Γ).

We will use a number of operations in the sequel: An operation f over D is said to be a constant operation if f is unary and $f(a) = c$ for all $a \in D$ and some $c \in D$; a majority operation if f is ternary and $f(a, a, b) = f(a, b, a) = f(b, a, a) = a$ for all $a, b \in D$; a binary commutative idempotent operation if f is binary, $f(a, a) = a$ for all $a \in D$, and $f(a, b) = f(b, a)$ for all $a, b \in D$; and an affine operation if f is ternary and $f(a, b, c) = a - b + c$ for all $a, b, c \in D$ where $+$ and $-$ are the operations of an Abelian group $(D, +, -)$.

3 Hardness and General Containment Results

In this section we prove some general containment results in **APX** and **poly-APX** for W-Max Sol(Γ). We also prove **APX**-completeness and **poly-APX**-completeness for W-Max Sol(Γ) for some particular constraint languages Γ. Most of our hardness results in subsequent sections are based on these results.

We begin by making the following easy but interesting observation: we know from the classification of W-Max Sol(Γ) over the Boolean domain $\{0, 1\}$ that there exist many constraint languages Γ for which W-Max Sol(Γ) is **poly-APX**-complete. However, if 0 is not in the domain, then W-Max Sol(Γ) is always in **APX**: it is proved in [8] that if Csp(Γ) is in **P**, then we can also find a solution in polynomial time, and it is clear that this solution is a $\frac{\max(D)}{\min(D)}$-approximate solution.

Proposition 2. *If* Csp(Γ) *is in* **P** *and* $0 \notin D$*, then* W-Max Sol(Γ) *is in* **APX**.

Next we present a general containment result in **poly-APX** for W-Max Sol(Γ). The proof is similar to the proof of the corresponding result for the Boolean domain in [20, Lemma 6.2], so we omit the proof.

Lemma 3. *Let* $\Gamma^c = \{\Gamma \cup \{\{(d_1)\}, \ldots, \{(d_n)\}\}$*, where* $D = \{d_1, \ldots, d_n\}$*. If* Csp(Γ^c) *is in* **P***, then* W-Max Sol(Γ) *is in* **poly-APX**.

As for the hardness results, we begin by proving the **APX**-completeness and **poly-APX**-completeness of particular constraint languages that will be very useful in subsequent sections. The hardness parts of the proof consists of reductions from the **APX**-complete problem Independent Set restricted to degree-3 graphs and the **poly-APX**-complete problem (unrestricted) Independent Set, respectively.

Lemma 4. *Let* $a, b \in D$ *such that* $0 < a < b$*. If* $r = \{(a, a), (a, b), (b, a)\}$*, then* W-Max Sol($\{r\}$) *is* **APX***-complete. If* $r' = \{(0, 0), (0, a), (a, 0)\}$*, then* W-Max Sol($\{r'\}$) *is* **poly-APX***-complete.*

4 Tractable Constraint Languages

In this section, we present some tractability results that will be needed in the sequel. These classes are *injective* constraint languages and *generalised max-closed* constraint languages. The tractability result for injective constraints follows from Cohen *et al.* [7, Sec. 4.4] but we present a simple proof for increased readability. The tractability result for generalised max-closed constraints is new and its proof constitutes the main part of this section.

To the best of our knowledge these two classes subsumes all the known tractable classes of constraint languages presented in the literature for this problem. In particular, they can be seen as substantial and nontrivial generalisations of the tractable classes known for the corresponding (Weighted) Max Ones problem over the Boolean domain. There are only three tractable classes of constraint languages over the Boolean domain, namely width-2 affine, 1-valid, and weakly positive [20]. Width-2 affine constraint languages are examples of injective constraint languages and the classes of 1-valid and weakly positive constraint languages are examples of generalised max-closed constraint languages.

Definition 5. *A relation, $R \in R_D$, is called* injective *if there exists a subset $D' \subseteq D$ and an injective function $\pi : D' \rightarrow D$ such that $R = \{(x, \pi(x)) \mid x \in D'\}$.*

It is important to note that the function π is *not* assumed to be total on D. Let I^D denote the set of all injective relations on the domain D and let $\Gamma_I^D = \langle I^D \rangle$.

To see that W-Max Sol(Γ_I^D) is in **PO**, it is sufficient to prove that W-Max Sol(I^D) is in **PO** by Theorem 1. Given an instance of W-Max Sol(I^D), consider the graph having the variables as vertices and edges between the vertices/variables occurring together in the same constraint. Each connected component of this graph represents an independent subproblem that can be solved in isolation. If a value is assigned to a variable/vertex, all vertices/variables in the same component will be forced to take a value by propagating this assignment. Hence, each connected component have at most $|D|$ different solutions that can be easily enumerated and an optimum one can be found in polynomial time.

Now, we consider generalised max-closed constraint languages. The generalised max-closed constraint languages are a significant and non-trivial generalisation of the 1-valid and weakly positive constraint languages which were defined and proved to be tractable for Max Ones in [20].

Definition 6. *A constraint language Γ over a domain $D \subset \mathbb{N}$ is* generalised max-closed *if and only if there exists a binary operation $f \in Pol(\Gamma)$ such that f satisfies the following conditions; for all $a, b \in D$ such that $a < b$ it holds that $f(a, b) > a$, $f(b, a) > a$, and for all $a \in D$ it holds that $f(a, a) \geq a$.*

The following example will clarify the definition above.

Example 7. In this example the domain, D, is $\{0, 1, 2, 3\}$. As an example of a generalised max-closed relation consider $R = \{(0, 0), (1, 0), (0, 2), (1, 2)\}$. R is invariant under max and is therefore generalised max-closed as max satisfies the properties of functions which are polymorphisms of generalised max-closed relations. A subset of the relations invariant under max are called monotone relations and has been studied by Hochbaum and Naor in [13]. Now consider the relation Q defined as $Q = \{(0, 1), (1, 0), (2, 1), (2, 2), (2, 3)\}$. Q is not invariant under max because

$$\max((0, 1), (1, 0)) = (\max(0, 1), \max(1, 0)) = (1, 1) \notin Q.$$

However, if we let the commutative and idempotent function $f : D^2 \rightarrow D$ be defined as $f(0, 1) = 2, f(0, 2) = 2, f(0, 3) = 3, f(1, 2) = 2, f(1, 3) = 2$ and $f(2, 3) = 3$ then Q is invariant under f. Furthermore, from the definition of generalised max-closed constraints, it is easy to verify that $Inv(f)$ consists of generalised max-closed relations.

The tractability of generalised max-closed constraint languages crucially depends on the following lemma.

Lemma 8. *If Γ is generalised max-closed, then all relations $R = \{(d_{11}, d_{12}, \ldots, d_{1m}), \ldots, (d_{t1}, d_{t2}, \ldots, d_{tm})\}$ in Γ have the property that the tuple $\boldsymbol{t}_{\max} = (\max\{d_{11}, \ldots, d_{t1}\}, \ldots, \max\{d_{1m}, \ldots, d_{tm}\})$ is in R, too.*

Proof. Assume that there is a relation R in Γ such that the tuple $\boldsymbol{t}_{\max} = (\max\{d_{11}, \ldots, d_{t1}\}, \ldots, \max\{d_{1m}, \ldots, d_{tm}\})$ is not in R. Define the distance between two tuples to be the number of coordinates where they disagree (i.e. the Hamming distance). Let $\boldsymbol{a}$ be a tuple in R with minimal distance from $\boldsymbol{t}_{\max}$. By the assumption that $\boldsymbol{t}_{\max}$ is not in R, we know that the distance between $\boldsymbol{a}$ and $\boldsymbol{t}_{\max}$ is at least 1. Let T be the set of tuples $\boldsymbol{b}$ such that $\boldsymbol{b}$ is in R, and $\boldsymbol{b}$ and $\boldsymbol{t}_{\max}$ agree on at least one coordinate where $\boldsymbol{t}_{\max}$ and $\boldsymbol{a}$ disagrees. We can see that T is non-empty.

Let I denote the set of coordinates where $\boldsymbol{a}$ agrees with $\boldsymbol{t}_{\max}$. Order the tuples in T by extending the ordering on D to tuples over D componentwise, but only taking the coordinates in I into account (when comparing $\boldsymbol{b}$ and $\boldsymbol{b'}$), i.e., $\boldsymbol{b} = (d_1, \ldots, d_m) < (d'_1, \ldots, d'_m) = \boldsymbol{b'}$ if and only if $d_i \leq d'_i$ for all $i \in I$ and $d_j < d_j$ for some $j \in I$. We say that a tuple $\boldsymbol{t}$ (in T) is maximal if there exists no other tuple $\boldsymbol{t'}$ (in T) such that $\boldsymbol{t} < \boldsymbol{t'}$. Thus T contains (at least) one maximal tuple $\boldsymbol{m}$ under this ordering. Note that $\boldsymbol{a}$ and $\boldsymbol{m}$ disagree in at least one coordinate $k \in I$ where $\boldsymbol{a}$ agrees with $\boldsymbol{t}_{\max}$, otherwise we get a contradiction with the fact that $\boldsymbol{a}$ is of minimal distance from $\boldsymbol{t}_{\max}$.

Remember that since Γ is generalised max-closed there exists an operation $f \in Pol(\Gamma)$ such that for all $a, b \in D$, $a < b$ it holds that $f(a, b) > a$ and $f(b, a) > a$. Furthermore, for all $a \in D$ it holds that $f(a, a) \geq a$. Now, we apply f componentwise to $\boldsymbol{a}$ and $\boldsymbol{m}$, and let $\boldsymbol{x} = f(\boldsymbol{a}, \boldsymbol{m})$. Note that since $\boldsymbol{m}[k] < \boldsymbol{a}[k]$, we get that $\boldsymbol{x}[k] > \boldsymbol{m}[k]$. Now consider the tuple:

$$\underbrace{f(f(f(\ldots f(\boldsymbol{x}, \boldsymbol{m}), \ldots, \boldsymbol{m}), \boldsymbol{m}), \boldsymbol{m})}_{f \text{ applied } |D| \text{ times}} = \boldsymbol{x}^*.$$

It is easy to realise that $\boldsymbol{x}^*$ agrees with $\boldsymbol{m}$ in all coordinates where $\boldsymbol{m}$ agrees with $\boldsymbol{t}_{\max}$, so $\boldsymbol{x}^*$ is in T. Moreover $\boldsymbol{x}^*[i] \geq \boldsymbol{m}[i]$ for all $i \in I$, and $\boldsymbol{x}^*[k] > \boldsymbol{m}[k]$, so $\boldsymbol{x}^*$ cannot be in T since this would contradict the maximality of $\boldsymbol{m}$. We get a contradiction because $\boldsymbol{x}^*$ cannot be both out of T and in T. Hence our assumption was wrong and $\boldsymbol{t}_{\max}$ is in R. $\square$

Now we can use the same approach as was used in [16] to prove that the max-closed constraint languages are a tractable class for CSP. Namely, given an instance of W-MAX SOL(Γ) where Γ is generalised max-closed, first establish a form of consistency called pair-wise consistency. If it results in any empty constraints, then the instance has no solutions. Otherwise, assigning to each variable the largest value allowed to it by any constraint results in a solution to the instance. This follows from the fact that the set of underlying relations in the resulting pair-wise consistent instance is still generalised max-closed. This solution is obviously an optimum one and since pair-wise consistency can be established in polynomial time the problem is in **PO**.

Theorem 9. *If Γ is generalised max-closed, then* W-Max Sol(Γ) *is in* **PO**.

5 Result 1: Maximal Constraint Languages

A maximal constraint language Γ is a constraint language such that $\langle\Gamma\rangle \subset R_D$, and if $r \notin \langle\Gamma\rangle$, then $\langle\Gamma \cup \{r\}\rangle = R_D$. That is, the maximal constraint languages are the largest constraint languages that are not able to express all finitary relations over D. Recently a complete classification for the complexity of the Csp(Γ) problem for all maximal constraint languages was completed [3,6].

Theorem 10 ([3,6]). *Let Γ be a maximal constraint language on an arbitrary finite domain D. Then,* Csp(Γ) *is in* **P** *if $\langle\Gamma\rangle = Inv(\{f\})$ where f is a constant operation, a majority operation, a binary commutative idempotent operation, or an affine operation. Otherwise,* Csp(Γ) *is* **NP**-*complete.*

In this section, we classify the approximability of W-Max Sol(Γ) for all maximal constraint languages Γ.

Theorem 11. *Let $\langle\Gamma\rangle = Inv(\{f\})$ be a maximal constraint language on an arbitrary finite domain D.*

1) If Γ is generalised-max-closed or an injective constraint language, then W-Max Sol($\langle\Gamma\rangle$) *is in* **PO**.
2) Else if f is an affine operation, a constant operation different from the constant 0 *operation, or a binary commutative idempotent operation satisfying $f(0,b) > 0$ for all $b \in D \setminus \{0\}$ (assuming $0 \in D$); or if $0 \notin D$ and f is a binary commutative idempotent operation or a majority operation, then* W-Max Sol(Γ) *is* **APX**-*complete.*
3) Else if f is a binary commutative idempotent operation or a majority operation, then W-Max Sol(Γ) *is* **poly-APX**-*complete.*
4) Else if f is the constant 0 *operation, then finding a solution with non-zero measure is* **NP**-*hard.*
5) Otherwise, finding a feasible solution is **NP**-*hard.*

The proof of the preceding theorem consists of a careful analysis of the approximability of W-Max Sol(Γ) for all constraint languages $\Gamma = Inv(\{f\})$, where f is of one of the types of operations in Theorem 10. The approximability classifications when f is a majority operation or a constant operation are fairly straightforward, so we concentrate on the results for the remaining two cases (i.e., when f is affine or a binary idempotent commutative operation).

5.1 Binary Commutative Idempotent Operations

The approximability of W-Max Sol($Inv(\{f\})$) when f is a binary idempotent commutative operation and $Inv(\{f\})$ is a maximal constraint language is determined in the following lemma.

Lemma 12. *Let f be a binary idempotent commutative operation on D such that $Inv(\{f\})$ is a maximal constraint language.*

- *If $f = \frac{p+1}{2}(x+y)$, where $+$ is the operation of an Abelian group of prime order $p = |D|$, then* W-Max Sol($Inv(\{f\})$) *is* **APX**-*complete;*
- *else if there exist $a, b \in D$ such that $a < b$ and $f(a,b) \leq a$, let a be the minimal such element (according to $<$), then*
  - W-Max Sol($Inv(\{f\})$) *is* **poly-APX**-*complete if $a = 0$, and*
  - **APX**-*complete if $a > 0$.*
- *Otherwise,* W-Max Sol($Inv(\{f\})$) *is in* **PO**.

The proof of Lemma 12 crucially depends on the following result from universal algebra due to Szczepara [24].

Lemma 13 ([24]). *Let f be a binary operation such that $Inv(\{f\})$ is a maximal constraint language. Then f can be supposed either*

1. *to be the operation $f = \frac{p+1}{2}(x+y)$, where $+$ is the operation of an Abelian group of prime order $p = |D|$, or*
2. *to satisfy $f(f(x,y),y) = y$ and $\underbrace{f(x, f(x, \ldots f}_{n \text{ times}}(x,y)\ldots)) = y$, or*
3. *to satisfy $f(f(x,y),y) = f(x,y)$ and $f(x, f(x,y)) = y$, or*
4. *to satisfy $f(f(x,y),y) = f(x,y)$ and*

$$\underbrace{f(x, f(x, \ldots f}_{n+1 \text{ times}}(x,y)\ldots)) = \underbrace{f(x, f(x, \ldots f}_{n \text{ times}}(x,y)\ldots)).$$

A bird's eye view of the proof of Lemma 12 is this: For every binary idempotent commutative operation f of the four types in Lemma 13 we investigate the approximability of W-Max Sol($Inv(\{f\})$). If f is of type 1, then it can be proved that W-Max Sol($Inv(\{f\})$) is **APX**-complete using similar methods as those in Section 5.2 below, so we concentrate on the remaining three types. If f is of type 2, 3, or 4, but $Inv(\{f\})$ is not generalised max-closed, then we prove that there exist $a, b \in D$, such that $a < b$ and f acts as the min function on $\{a, b\}$. If the minimal a such that f acts as the min function on $\{a, b\}$ is 0, then W-Max Sol($Inv(\{f\})$) is **poly-APX**-complete. If the minimal such a is greater than 0, then W-Max Sol($Inv(\{f\})$) is **APX**-complete. The only remaining case is when $Inv(\{f\})$ is generalised max-closed, which is in **PO** by Theorem 9.

Lemma 14. *Let f be a binary commutative idempotent operation such that*

- *f satisfies condition 2, 3, or 4 from Lemma 13, and*
- *$Inv(\{f\})$ is not generalised max-closed (as defined in Definition 6).*

Then there exist two elements $a < b$ in D such that $f(b,a) = f(a,b) = f(a,a) = a$ and $f(b,b) = b$ (i.e., f acts as the min*-function on the set $\{a, b\}$).*

Proof. Given that f is binary commutative idempotent and that $Inv(\{f\})$ is not generalised max-closed, it follows immediately from the definition of generalised max-closed constraints (Definition 6) that there exist two elements $a < b$ in D such that $f(a, b) \leq a$. If $f(a, b) = a$, then f acts as the min-function on $\{a, b\}$ and we are done. Hence, we can assume that $f(a, b) < a$.

– If f satisfies condition 3 or 4 in Lemma 13, then $f(f(x, y), y) = f(x, y)$ for all $x, y \in D$. In particular, $f(f(a, b), b) = f(a, b)$. Hence, f acts as the min-function on the set $\{f(a, b), b\}$.
– If f satisfies condition 2 in Lemma 13, then $f(f(x, y), y) = y$ for all x, y in D. If there exist c, d in D such that $d < f(c, d)$, then $f(f(c, d), d) = d$ and f acts as the min-function on $\{f(c, d), d\}$. Otherwise, it holds for all x, y in D that $f(x, y) \leq x$ and $f(x, y) \leq y$. Let a be the minimal (least) element in D (according to $<$) and let b be the second least element, then f acts as the min-function on $\{a, b\}$. □

Lemma 15. *Let f be a binary commutative idempotent operation such that $Inv(\{f\})$ is not generalised max-closed and f satisfies condition* 2*,* 3*, or* 4 *from Lemma 13. Let a be the minimal element in D such that $f(a, b) = \min(a, b) = a$ and $a < b$ for some $b \in D$ (such elements a and b are known to exist from Lemma 14). Then,* W-Max Sol($Inv(\{f\})$) *is* **poly-APX**-*complete if* $a = 0$ *and* **APX**-*complete if* $0 < a$*.*

Proof. We know that f acts as the min-function on the set $\{a, b\}$. This implies that the relation $r = \{(a, a), (b, a), (a, b)\}$ is in $Inv(\{f\})$. We will consider three cases

1. $0 \notin D$. Then, W-Max Sol($Inv(\{f\})$) is in **APX** by Proposition 2 and the problem is **APX**-complete by Lemma 4.
2. $0 \in D$, and $a = 0$. Then, W-Max Sol($Inv(\{f\})$) is **poly-APX**-complete by Lemma 4.
3. $0 \in D$ and $a > 0$. We will show that W-Max Sol($Inv(\{f\})$) is **APX**-complete in this case.

First, we prove that the unary relation $u = D \setminus \{0\}$ is in $Inv(\{f\})$. Assume that $f(c, d) = 0$ for some $c < d$. If f satisfies condition 3 or 4 from Lemma 13, then $f(f(c, d), d) = f(c, d) = 0$ and hence f acts as the min-function on $\{f(c, d), d\}$. Note that, $f(c, d) < a$ which contradicts that a is the minimal such element. Now, if f satisfies condition 2 from Lemma 13 (i.e., $f(f(x, y), y) = y$ for all x, y in D), and if there exists an e, $0 < e$, such that $f(e, 0) > 0$, then f acts as the min-function on $\{f(e, 0), 0\}$ contradicting that $a > 0$ is the minimal such element. Thus, f must act as the min-function on $\{0, x\}$ for every $x \in D$, again contradicting that $a > 0$ is the minimal element such that f acts as the min-function on $\{a, y\}$ for any $y \in D$. So, our assumption was wrong and $f(c, d) > 0$ for all $c, d \in D \setminus \{0\}$. This implies that $u = D \setminus \{0\}$ is in $Inv(\{f\})$.

Let $I = (V, D, C, w)$ be an arbitrary instance of W-Max Sol($Inv(\{f\})$). Define $V' \subseteq V$ such that $V' = \{v \in V \mid M(v) = 0 \text{ for every model } M \text{ of } I\}$. We see that V' can be computed in polynomial time: a variable v is in V'

if and only if the CSP instance $(V, D, C \cup \{((v), u)\})$ is not satisfiable. Define $M : V \to D$ such that $M(v) = \max D$ if $v \in V \setminus V'$ and $M(v) = 0$, otherwise. Now, $\text{OPT}(I) \leq \sum_{v \in V \setminus V'} w(v)M(v)$.

Given two assignments $A, B : V \to D$, we define the assignment $f(A, B)$ such that $f(A, B)(v) = f(A(v), B(v))$. We note that if A and B are models of I, then $f(A, B)$ is a model of I: indeed, arbitrarily choose one constraint $((x_1, \ldots, x_k), r) \in C$. Then, $(A(x_1), \ldots, A(x_k)) \in r$ and $(B(x_1), \ldots, B(x_k)) \in r$ which implies that $(f(A(x_1), B(x_1)), \ldots, f(A(x_k), B(x_k))) \in r$, too.

Let $M_1, \ldots, M_m$ be an enumeration of all models of I and define

$$M^+ = f(M_1, f(M_2, f(M_3 \ldots f(M_{m-1}, M_m) \ldots))).$$

By the choice of V' and the fact that $f(c, d) = 0$ if and only if $c = d = 0$, we see that the model M^+ has the following property: $M^+(v) = 0$ if and only if $v \in V'$. Let p denote the second least element in D and define $M' : V \to D$ such that $M'(v) = p$ if $v \in V \setminus V'$ and $M'(v) = 0$, otherwise. Now, $\text{OPT}(I) \geq \sum_{v \in V \setminus V'} w(v)M'(v) = S$. Thus, by finding a model M'' with measure $\geq S$, we have approximated I within $(\max D)/p$ and W-MAX SOL($Inv(\{f\})$) is in **APX**. To find such a model, we consider the instance $I'' = (V, D, C', w)$, where $C' = C \cup \{((v), u) \mid v \in V \setminus V'\}$. This instance has feasible solutions (since M^+ is a model) and any solution has measure $\geq S$, furthermore a concrete solution can be found in polynomial time by the result in [8]. $\square$

5.2 Affine Constraints

We begin by showing that the class of constraints invariant under an affine operation give rise to **APX**-hard W-MAX SOL-problems. We will denote the affine operation on the group G with a_G.

Theorem 16. W-MAX SOL($Inv(\{a_G\})$) *is* **APX**-*hard for every affine operation* a_G.

This theorem is proved with two reductions, the first one from MAX-k-CUT which is **APX**-complete [1], to MAX SOL EQN (see [21] for definition). In the second reduction we reduce MAX SOL EQN to W-MAX SOL($Inv(\{a_G\})$).

We will now prove that constraint languages that are invariant under an affine operation give rise to problems which are in **APX**. It has been proved that a relation which is invariant under an affine operation is a coset of a subgroup of some Abelian group [15]. We will give a randomised approximation algorithm for the more general problem when the relations are cosets of subgroups of a general finite group. Our algorithm is based on the algorithm for the decision variant of this problem by Feder and Vardi [11].

We start with a technical lemma which states that the intersection of two cosets is either empty or a coset.

Lemma 17. *Let aA and bB $(a, b \in H)$ be two cosets of H. Then, $aA \cap bB$ is either empty or a coset of $C = A \cap B$ with representative $c \in H$.*

We can now prove our approximability results for affine closed constraints.

Theorem 18. W-Max Sol*(Γ) is in* **APX** *if every relation in Γ is a coset of some subgroup of G^k for some finite group G and integer k.*

Proof. Let $I = (V, D, C, w)$ be an arbitrary instance with $V = \{v_1, \ldots, v_n\}$. Feasible solutions to our problem can be seen as certain elements in $H = G^n$. Each constraint $C_i \in C$ defines a coset $a_i J_i$ of H with representative $a_i \in H$, for some subgroup J_i of H. The set of solutions to the problem is the intersection of all those cosets. Thus, $S = \bigcap_{i=1}^{|C|} a_i J_i$ denotes the set of all solutions.

Let $H' = J_1 \cap J_2 \cap \ldots \cap J_i$ for $1 \leq i \leq |C|$. By Lemma 17, we see that $S = xH'$ where x is one solution and H' is a subgroup of H. The solution x can be computed in polynomial time by the algorithm in [11, Theorem 33], and a set of generators for the subgroup H' can also be computed in polynomial time [14, Theorem II.12]. Given a set of generators for H', we can uniformly at random pick an element h from H' in polynomial time [14]. The element xh is then a solution to I which is picked uniformly at random from S.

Let V_i denote the random variable which corresponds to the value which will be assigned to v_i by the algorithm above. It is clear that each V_i will be uniformly distributed over some subset of G. Let A denote the set of indices such that for every $i \in A$, $\Pr[V_i = c_i] = 1$ for some $c_i \in G$. That is, A contains the indices of the variables V_i which are constant in every feasible solution. Let B contain the indices for the variables which are not constant in every solution, i.e., $B = [n] \setminus A$.

Let $S^* = \sum_{i \in B} w(v_i) \max G + \sum_{i \in A} w(v_i) c_i$ and note that $S^* \geq \textsc{opt}$. Furthermore, let

$$E_{\min} = \min_{X \subseteq G, |X|>1} \frac{1}{|X|} \sum_{x \in X} x$$

and note that $\max G > E_{\min} > 0$.

The expected value of a solution produced by the algorithm can now be estimated as

$$\begin{aligned} \mathrm{E}\left[\sum_{i=1}^{n} w(v_i) V_i\right] &= \sum_{i \in A} w(v_i) \mathrm{E}[V_i] + \sum_{i \in B} w(v_i) \mathrm{E}[V_i] \\ &\geq \sum_{i \in A} w(v_i) c_i + E_{\min} \sum_{i \in B} w(v_i) \geq \frac{E_{\min}}{\max G} S^* \geq \frac{E_{\min}}{\max G} \textsc{opt}. \end{aligned}$$

As $E_{\min}/\max G > 0$ we have a randomised approximation algorithm with a constant expected performance ratio. □

6 Result 2: Homogeneous Constraint Languages

Dalmau completely classified the complexity of CSP(Γ) when Γ is a *homogeneous* constraint language [10]. A constraint language Γ over D is homogeneous if every permutation relation $R = \{(x, \pi(x)) \mid x \in D\}$ (where π is a permutation

$\pi : D \rightarrow D$) is contained in the language. Let Q denote the set of all permutation relations on D. The following theorem (together with Dalmau's classification for CSP(Γ)) gives a complete classification for the approximability of W-Max Sol(Γ) when $Q \subseteq \Gamma$.

Theorem 19. *Assume that $Q \subseteq \Gamma$ and that* CSP(Γ) *is in* **P**. *If $\Gamma \subseteq \Gamma_I^D$ (where Γ_I^D is the class of injective relations from Definition 5), then* W-Max Sol(Γ) *is in* **PO**. *Otherwise, if $0 \notin D$ or Γ is invariant under an affine operation, then* W-Max Sol(Γ) *is* **APX**-*complete. Otherwise,* W-Max Sol(Γ) *is* **poly-APX**-*complete.*

The proof of this theorem makes heavy use of the structure of homogeneous algebras and is given in the technical report version of this paper [19].

As a direct consequence of the preceding theorem we get that the class of injective relations is a maximal tractable class for W-Max Sol(Γ). That is, if we add a single relation which is not an injective relation to the class of all injective relations, then the problem is no longer in **PO** (unless **P** = **NP**).

7 Conclusions

We view this paper as a first step towards a better understanding of the approximability of non-Boolean W-Max Sol. The ultimate long-term goal for this research is, of course, to completely classify the approximability for all finite constraint languages. However, we expect this to be somewhat difficult since not even a complete classification for the corresponding decision problem Csp is known. A more manageable task would be to completely classify W-Max Sol for constraint languages over small domains (say, of size 3 or 4). For size 3, this has already been accomplished for Csp [4] and Max Csp [18].

It is known from [20] that the approximability of the weighted and unweighted versions of (W)-Max Sol coincide for all Boolean constraint languages. We remark that the same result holds for all constraint languages considered in this paper (i.e., maximal constraint languages and homogeneous constraint languages). This can be readily verified by observing that the *AP*-reduction from W-Max Sol to Max Sol in the proof of Lemma 3.11 in [20] easily generalises to arbitrary finite domains and that our tractability proofs are given for the weighted version of the problem. Note that since *AP*-reductions do not in general preserve membership in **PO** it is still an open problem whether W-Max Sol(Γ) is in **PO** if and only if Max Sol(Γ) is in **PO** for arbitrary constraint languages Γ.

References

1. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti Spaccamela, and M. Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties.* Springer, 1999.
2. A. Bulatov. A dichotomy theorem for constraints on a three-element set. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pages 649–658, 2002.

3. A. Bulatov. A graph of a relational structure and constraint satisfaction problems. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 448–457, 2004.
4. A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. *Journal of the ACM*, 53(1):66–120, 2006.
5. A. Bulatov, P. Jeavons, and A. Krokhin. Classifying the complexity of constraints using finite algebras. *SIAM J. Comput.*, 34(3):720–742, 2005.
6. A. Bulatov, A. Krokhin, and P. Jeavons. The complexity of maximal constraint languages. In *Proceedings of the 33rd ACM Symposium on Theory of Computing (STOC 2001)*, pages 667–674, 2001.
7. D. A. Cohen, M. Cooper, P. Jeavons, and A. Krokhin. Soft constraints: complexity and multimorphisms. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, pages 244–258, 2003.
8. D.A. Cohen. Tractable decision for a constraint language implies tractable search. *Constraints*, 9(3):219–229, 2004.
9. N. Creignou, S. Khanna, and M. Sudan. *Complexity classifications of Boolean constraint satisfaction problems*. SIAM, Philadelphia, 2001.
10. V. Dalmau. A new tractable class of constraint satisfaction problems. *Annals of Mathematics and Artificial Intelligence*, 44(1–2):61–85, 2005.
11. T. Feder and M.Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. *SIAM J. Comput.*, 28(1):57–104, 1999.
12. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
13. D.S. Hochbaum and J. Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM J. Comput.*, 23(6):1179–1192, 1994.
14. C.M. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism*, volume 136 of *Lecture Notes in Computer Science*. Springer, 1982.
15. P. Jeavons, D. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of the ACM*, 44:527–548, 1997.
16. P. Jeavons and M. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79:327–339, 1996.
17. P. Jonsson. Near-optimal nonapproximability results for some NPO PB-complete problems. *Information Processing Letters*, 68(5):249–253, 1998.
18. P. Jonsson, M. Klasson, and A. Krokhin. The approximability of three-valued Max CSP. *SIAM J. Comput.*, 35(6):1329–1349, 2006.
19. P. Jonsson, F. Kuivinen, and G. Nordh. Approximability of integer programming with generalised constraints. *CoRR*, cs.CC/0602047, 2006.
20. S. Khanna, M. Sudan, L. Trevisan, and D.P. Williamson. The approximability of constraint satisfaction problems. *SIAM J. Comput.*, 30(6):1863–1920, 2001.
21. F. Kuivinen. Tight approximability results for the maximum solution equation problem over Z_p. In *Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science (MFCS 2005)*, pages 628–639, 2005.
22. R. Pöschel and L. Kaluznin. *Funktionen- und Relationenalgebren*. DVW, Berlin, 1979.
23. A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.
24. B. Szczepara. *Minimal clones generated by groupoids*. PhD thesis, Université de Móntreal, 1996.

When Constraint Programming and Local Search Solve the Scheduling Problem of Electricité de France Nuclear Power Plant Outages*

Mohand Ou Idir Khemmoudj[1], Marc Porcheron[2], and Hachemi Bennaceur[1]

[1] LIPN-CNRS UMR 7030, 99 Av J-B. Clément, 93430 Villetaneuse, France
{MohandOuIdir.Khemmoudj, Hachemi.Bennaceur}@lipn.univ-paris13.fr
[2] EDF R&D, 1 Av du Général-de-Gaulle 92141 Clamart, France
Marc.Porcheron@edf.fr

Abstract. The French nuclear park comprises 58 nuclear reactors distributed through the national territory on 19 geographical sites. They must be repeatedly stopped, for refueling and maintenance. The scheduling of these outages has to comply with various constraints, regarding safety, maintenance logistic, and plant operation, whilst it must contribute to the producer profit maximization. This industrial problem appears to be a hard combinatorial problem that conventional methods used up to now by Electricité de France (mainly based on Mixed Integer Programming) fail to solve properly. We present in this paper a new approach for modeling and solving this problem, combining Constraint Programming (CP) and Local Search. CP is used to find solutions to the outage scheduling problem, while Local Search is used to improve solutions with respect to a heuristic cost criterion. It leads to find solutions as good as with the conventional approaches, but taking into account all the constraints and in very reduced computing time.

1 Introduction

For modeling and solving optimization problems, Constraint Programming (CP) [2] has been proved to be a very flexible approach. The user states the problem as a set of decision variables and constraints, and express the enumeration strategy which will be adopted to solve the problem then the system find a valuation of the variables satisfying the constraints. The resulting program can be easily adapted to changing requirements, since constraints can be added in incremental way to a system without changing its overall architecture. And, the strategy used to enumerate a solution can be easily modified. This allows to include specific information or knowledge resulting from the expertise inside the enumeration strategy.

* This work is supported in part by the French Electricity Board (EDF).

Hence, CP can be applied to express in a very flexible way real-world problems and, using a good enumeration strategy, it can leads to find a good first solution. However, in most cases, it cannot be used to solve problem of big size to optimality. For such problems, Local Search methods [1] give, in general, very good practical results. They are repair methods which can give near-optimal solutions in reasonable computing time.

Local search methods are based on a simple and general idea. They start from an initial solution which can be randomly generated and try to improve it. To do this, they first define a neighborhood which is a restricted search space around the current solution. It is a subset of the overall solution space (e.g., the solutions which can be obtained by modifying the value of just one variable in the current solution). Then, they select the optimal solution contained in the defined neighborhood. If the selected solution is different from the current one, the process is iterated by considering it as the new current solution. Otherwise, we say that a local optimum has been reached. There are several techniques to escape from local optima [3,5,6].

In attempt to gain both efficiency of Local Search and the flexibility of constraint programming, recent research has started addressing the combination of these two techniques [4,7,8,9,10,11].

Our aim in this paper is to show that combining CP with Local Search constitutes a powerful approach to model and solve a real world problem. It consists of the scheduling of outages of the Nuclear Power Plants (NPP) of the French utility for electricity supply, Electricté de France (EDF). This is a core problem, critical for the whole optimization chain of EDF production, as nuclear energy forms the main part of it. Namely, NPP unavailability during periods of high demand can result in drastic loss, because costly conventional production means have to be triggered as a substitute, e.g. fuel, coal or gas plants.

Outages scheduling appears to be a hard combinatorial problem. Up to now, it has been addressed by means of classic Operations Research methods, mostly based on Mixed Integer Programming. Two decision-aid tools based on these approaches are currently used monthly by EDF operational staff to compute outage schedules. Because these systems are suffering from severe limitations, we initiated a couple of years ago a fundamental research in order to study the contribution of Constraint Programming and Local Search methods for modeling and solving this problem. This led us to design a new approach, taking advantages of both techniques. This paper describes this innovative approach and discusses its benefits. It is organized as follows : Section 2 provides with an informal description of the problem to be solved. Section 3 briefly presents the current modelisation and resolution techniques used, in order to furnish a precise estimation of the problem complexity and to point out the limitations of the current approaches. It also serves to introduce some notation used in the following of the paper. Section 4 is devoted to the presentation of the proposed alternative approach. Section 5 presents the entire process to solve completely the problem. We discuss our experimental results and present our future work in the section 6.

2 Problem Overview

The French Electricity board (Électricité de France, EDF) disposes from 58 nuclear power plants (NPP) distributed on 19 geographical sites. Their production management has to comply with various constraints, regarding safety, maintenance logistic, and plant operation, whilst it must contribute to the producer profit maximization. In particular, NPP must be repeatedly stopped, for refueling and maintenance. Thus, NPP operation consists of a succession of cycles (see Fig 1), each of them composed by the sequence of a production campaign and an outage.

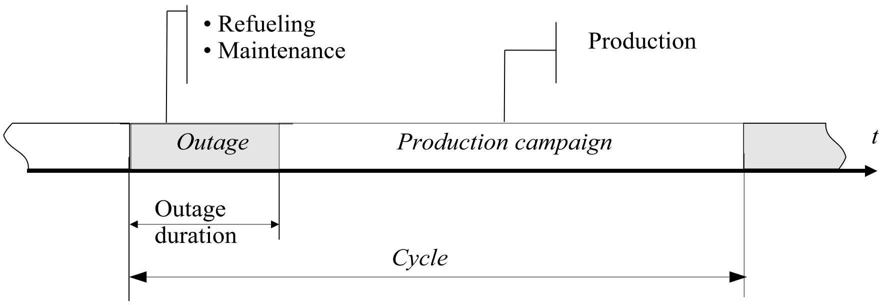


Fig. 1. Reactor cycle

The whole problem consists of two dependent sub-problems :

1. **NPP outage scheduling.** It consists of finding out a planning of outage dates, on a horizon of 5 years, for the 58 nuclear reactors. The schedule must satisfy some constraints designated in the following as *placement constraints.* They are of two types : *intra-site placement constraints* and *inter-site placement constraints.* The former relate outages of plants located on a single site. Mainly, they consist of minimal space or maximal covering tolerated between couple of outages. The latter constraints are resource limitation constraints which relate outages of plants located on different sites. For example, at each step, the number of stopped plants on the whole park must be limited to a given bound. In addition to the *intra-site* and *inter-site* placement constraints, individual outages can have unary constraints such as earliest or latest start dates.
2. **Production planning problem.** It consists of finding out an optimal production plan, i.e. a quantity of energy to produce by each plant at each step of time (week). The production plan must respect the *"local production constraints"*. During the *production campaign*, the power of a NPP is limited to an upper-bound. The plant is allowed to produce under this bound, but this ability to modulate production is constrained by an upper-bound called *Maximal Modulation.* After each refueling, the NPP stores a certain amount of energy for the next cycle. If this energy has not been exhausted

before the subsequent outage, the plant is said to stop with *anticipation. Anticipation* is constrained by an upper-bound *Amax.* On the contrary, if this energy has been consumed before the next outage, the plant is said to stop with *prolongation*, and it can continue to produce up to a fuel stock lower-bound called *Maximal Prolongation.* During this possible prolongation stage, the plant must follow a specific *decreasing production profile* and it is not allowed to modulate. During each outage, the plant core is restocked for the next campaign with respect to a specific refueling law. In addition, at each step of the studied period NPP and conventional plant production must fulfill a given demand. This is the only coupling production constraint.

The economic criterion to minimize is the global cost of nuclear fuel consumed during operations, augmented with the production cost of the conventional plants.

3 Current Resolution Framework

To solve the problem described above two complementary approaches, both based on Mixed Integer Programming (MIP), have been implemented and are currently used by the EDF operational staff.

Following are the major notations, constants and decision variables used in these frameworks.

- Notations
  - $\langle i, k\rangle$: k-the outage of NPP i;
- Constants
  - $\top$: number of weeks of the study period (horizon length);
  - m : number of nuclear power plants (58);
  - n_i : number of expected outages of NPP i (namely about 5);
  - $\underline{t}_i^k$: the earliest start time of the outage $\langle i, k\rangle$;
  - $\overline{t}_i^k$: the latest start time of the outage $\langle i, k\rangle$;
  - d_i^k : outage $\langle i, k\rangle$ duration;
  - L_i^k : nuclear fuel restocked during the outage $\langle i, k\rangle$.
- Decision variables
  - $a_i^k(t), i = 1...m; t = \underline{t}_i^k...\overline{t}_i^k$: boolean variables which takes the value 1 iff, the k-the outage of plant i begins at week t, 0 otherwise;
  - $u_i(t), i = 1...m; t = 1...\top$: real variables denoting the production of NPP i at week t;
  - $v(t), t = 1...\top$: real variables denoting the production of the conventional plants at week t (non-nuclear productions);
  - f_i : remaining stock fuel of the NPP i at the end of the study period.

The intra-site placement constraints are modelled as follows. For each outage couple $(\langle i, k\rangle, \langle j, l\rangle)$ the constraints below are stated.

$$a_i^k(t) + \sum_{t'=t}^{\overline{t}_j^l}(1 - r_{i_k j_l}(t, t')).a_j^l(t') \leq 1,\ t = \underline{t}_i^k...\overline{t}_i^k \tag{1}$$

$$a_j^l(t) + \sum_{t'=t}^{\overline{t}_i^k} (1 - r_{i_k j_l}(t', t)).a_i^k(t') \leq 1,\ t = \underline{t}_j^l ... \overline{t}_j^l \tag{2}$$

where $r_{i_k j_l}$ is the relation specifying the binary constraint relating the outage couple $(\langle i, k\rangle, \langle j, l\rangle)$.

Constraints above mean that if one of the outages $\langle i, k\rangle$ or $\langle j, l\rangle$ begins at time t, the other outage must not begin at any time t' such that $\neg r_{i_k j_l}(t, t')$. Since, there is no restriction on the relation which specifies the constraint, the formulation can be employed to express any binary constraint. In our case, the general form of the relations is :

$$r_{i_k j_l}(t, t') = \begin{cases} 1 \text{ if } t + d_i^k - e_i^j + \phi_{i_k j_l}(t, t').o_i^j \leq t' \\ \quad \vee\ t' + d_j^l - e_i^j + \phi_{i_k j_l}(t, t').o_i^j \leq t \\ 0 \text{ otherwise} \end{cases}$$

where e_i^j is the maximum covering allowed between outages of plants i and j. When it is negative, we say that it is a required minimum spacing. This maximum covering (or minimum spacing) must be augmented by o_i^j if there is a so-called specific period overlapped by the two intervals $[t, t + d_i^k - 1]$ and $[t', t' + d_j^l - 1]$. Namely, there is a given set of specific periods $\{[\underline{s}_1, \overline{s}_1], ..., [\underline{s}_p, \overline{s}_p]\}$ and the quantity $\phi_{i_k j_l}$ is equal to 1 if there exists q $(1 \leq q \leq p)$ such that $[t, t + d_i^k - 1] \cap [\underline{s}_q, \overline{s}_q] \neq \emptyset$ and $[t', t' + d_j^l - 1] \cap [\underline{s}_q, \overline{s}_q] \neq \emptyset$. Otherwise, $\phi_{i_k j_l}(t, t')$ is null.

The inter-site placement constraints, which are resources limitation constraints, are classically formulated as follow:

$$\sum_{\langle i,k\rangle} \sum_{t'=max(\underline{t}_i^k, t-\rho_{ik}^q+1)}^{t} a_i^k(t').R_{ik}^q \leq \delta_q\ q = 1...M; t = 1...\top. \tag{3}$$

where,

- M : number of different necessary resources for refueling and maintenance operations;
- δ_q : the amount of the resource q, $(q = 1...M)$;
- R_{ik}^q : the amount of resource q used during the outage $< i, k >$ (0 if the resource is not used);
- ρ_{ik}^q : the time during which the resource q is used since the outage $< i, k >$ beginning. It can be greater than the duration d_i^k of the outage $\langle i, k\rangle$. This is the case for example when the resource is a tool which requires a decontamination after its use.

As an illustration of local production constraints, we present below the modelisation of the constraint enforcing production of a given NPP to be null during outages and to be limited to an upper bound during campaigns.

$$\begin{aligned} &u_i(t) \leq E_i,\ \forall(i, t) : 1 \leq t \leq \top \\ &u_i(t) \leq \left(1 - \sum_{t'=max(\underline{t}_i^k, t-d_i^k+1)}^{t} a_i^k(t')\right) \cdot E_i,\ \forall(i, k, t) : \underline{t}_i^k \leq t \leq \overline{t}_i^k \end{aligned} \tag{4}$$

where E_i stands for the weekly maximal energy producible by the NPP i. It is worth noting that this type of constraints relates outage scheduling and production planning sub-problems.

Demand satisfaction constraints take the following form :

$$\sum_{i=1}^{m} u_i(t) + v(t) = Dem_t, t = 1...\top. \tag{5}$$

where Dem_t is the demand to be satisfied at week t.

Constraints (5) mean that at each step of the scheduling period, the nuclear and conventional production must fulfill the given demand.

After some approximations have been performed, the objective function to minimize takes the following form:

$$\left[\sum_{t=1}^{\top}\sum_{i=1}^{m} c_i(t).L_i^k.a_i^k(t) - \sum_{i=1}^{m} c_i(\top).f_i\right] + \left[\sum_{t=1}^{\top} g(v(t))\right] \tag{6}$$

with the following data :

- L_i^k : the amount of nuclear fuel restocked during the k-the outage of plant i;
- $c_i(t)$: the nuclear fuel unitary cost at week t;
- g : production cost function of the non-nuclear plants. It is a convex increasing piecewise linear function (Fig 2);
- $\sum_{t=1}^{\top}\sum_{i=1}^{m} c_i(t).L_i^k.a_i^k(t) - \sum_{i=1}^{m} c_i(\top).f_i$: cost of the nuclear fuel consumed during the study period;
- $\sum_{t=1}^{\top} g(v(t))$: production cost of the non-nuclear plants. The form of function g is illustrated on the following plot. It is important to notice its convex character which will be exploited in the following.

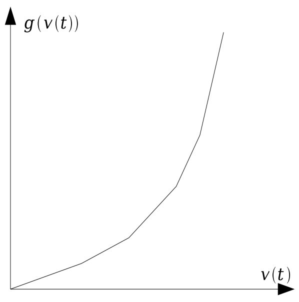


Fig. 2. Production cost function of the non-nuclear plants

Within this framework, and after some simplifications have been performed, the problem comprises about 5000 boolean variables, 20000 real variables, and 15000 constraints.

To face such a complexity, a first approach based on decomposition techniques has been developed. It consists of separating the global problem into sub-problems, each one dealing with a particular site, while a coordination module ensures the satisfaction of the coupling constraint on demand. The evident and major drawback of this method is that the placement constraints which relate NPP of different sites are not taken into account. In addition, in order to maintain an acceptable computing time, the search space of outage schedule is not completely explored and solutions found are thus only locally optimal.

A second approach tackling frontally the entire problem has been worked out allowing coupling inter-site placement constraints to be satisfied. But, despite the use of intensive MIP acceleration methods, it stills find in an acceptable computing time only solutions on a sub-part of the search space.

4 An Alternative Approach

The limitations of both approaches introduced above were the incentive for the work presented here. Basically, the intention was to investigate the capabilities of Constraint Programming together with Local Search techniques to contribute to problem solving with both following objectives : increasing the quality of the solutions found without relaxing any placement constraint. In addition to these two objectives, we are interested by a flexible tool offering the possibility to separate the sub-problem of outage scheduling from that of production planning. The current section describes this approach. We first show that Constraint Programming can be used to find an outage schedule for which there is a realizable production plan. Then we show that Constraint Programming can be combined with Local Search to improve the quality of the solution found.

4.1 Using CP to Solve the Outage Scheduling Sub-problem

Modeling. The CP modeling of the outage scheduling sub-problem introduces for each outage $\langle i, k\rangle$ a finite domain variable T_{ik}. The lower and upper bounds of each variable T_{ik} are respectively $\underline{t}_i^k$ and $\overline{t}_i^k$. The intra-site placement constraints are stated by using the generic global constraint *cumulative [12]*. For each plant couple (i, j) belonging to same geographical site the cumulative constraint below is stated :

$$cumulative(LT_{ij}, LD_{ij}, \mathbb{1}_{n_i+n_j}, 1) \tag{7}$$

- LT_{ij} is the variable collection $\{T_{ik}, T_{jl} : k = 1...n_i, l = 1...n_j\}$;
- LD_{ij} is the constant collection $\{d_i^k - e_i^j, d_j^l - e_i^j : k = 1...n_i, l = 1...n_j\}$;
- $\mathbb{1}_{n_i+n_j}$ is a collection of $n_i + n_j$ constants all equal to 1.

Constraints (7) above expresses a part of the relation r_{ij} introduced section 3. The complementary part is handled by exploiting the *conditional propagation* technique[1]. Namely, for each outage couple $(\langle i,k\rangle, \langle j,l\rangle)$ and each specific period $[\underline{s}_q, \overline{s}_q]$ the conditional constraint below is stated.

$$\begin{aligned}&(T_{ik} \leq \overline{s}_q \wedge T_{ik} + d_i^k - 1 \geq \underline{s}_q \wedge T_{jl} \leq \overline{s}_q \wedge T_{jl} + d_j^l - 1 \geq \underline{s}_q) \\ &\Rightarrow \ cumulative([T_{ik}, T_{jl}], [d_i^k - e_i^j + o_i^j, d_j^l - e_i^j + o_i^j], [1,1], 1)\end{aligned} \tag{8}$$

Like the intra-site placement constraints, the inter-site placement constraints are modeled by the cumulative global constraint. For resource q $(q = 1...M)$ we state :

$$cumulative(LT_q, LD_q, LR_q, \delta_q) \tag{9}$$

where :

- $LT_q = \{T_{ik} : R_{ik}^q > 0, i = 1...q, k = 1...n_i\}$;
- $LD_q = \{\rho_{ik}^q : R_{ik}^q > 0, i = 1...q, k = 1...n_i\}$;
- $LR_q = \{R_{ik}^q : R_{ik}^q > 0, i = 1...q, k = 1...n_i\}$;

We will design by $CSP(T)$ the set of all outage schedules satisfying the placement constraint (7-9), where T is the vector of the start time dates of the outages $(T = (T_{ik})_{i=1...m,k=1...n_i})$.

Search. The heuristic search for outage schedule consists of firstly assigning the variable with the smallest upper bound. Thus, the outages of each plant i $(i = 1...m)$ are considered in the order $\langle i,1\rangle, \langle i,2\rangle, ..., \langle i,n_i\rangle$. If there is two variables T_{ik} and $T_{jl}(i \neq j)$ with the same upper bound, we first consider the most-constrained one. The value ordering heuristic is based on the expertise of NPP operators and consists of minimizing the number of outages in winters (periods of keen demand).

Propagation of local production constraints. The outage schedule must be determined in such a way that there is a production plan satisfying all the local production constraints. These ones are thus integrated in the search process.

At the root of the search process, we calculate for each NPP the earliest and latest dates for the start of the first outage. This is done by exploiting the local production constraints and taking into account the fuel stock levels.

More precisely, let Ln_{i1} the current stock fuel level at the beginning of the study period for each plant i $(i = 1...m)$. The maximal amount of energy which can be produced before the outage $< i, 1 >$ is then $Ln_{i1} + Prmax_{i1}$, where $Prmax_{i1}$ is the *Maximal Prolongation* introduced section 2.

Secondly, the maximal length l_1 of the period during which the plant can modulate is equal to $\lceil \frac{Ln_{i1} + Mod_{max}}{E_i} \rceil$, where E_i is the maximal power of NPP i and Mod_{max} is the upper bound limiting modulation as introduced section 2. Thirdly, we use the mandatory *decreasing production profile* introduced section 2 to compute the maximal length l_2 of the prolongation period. Finally, the

[1] A *conditional constraint* $p \Rightarrow q$ enforces the consequent q once the antecedent p is verified.

maximal length $\bar{l} = l_1 + l_2$ of the production campaign is used to reduce the upper bound of T_{i1} ($T_{i1} \leq \bar{l}$).

An analogue reasoning is used to compute a lower bound $\underline{l}$ for T_{i1} using the *Maximal Anticipation* introduced section 2.

Each time a variable T_{ik} has been bound, an interval can be computed for the stock level reached at the end of the current campaign and the stock level at the beginning of the next campaign can be consequently constrained by means of the refueling law.

The same process is then repeated to calculate upper and lower bounds for the variable T_{ik+1} and so on.

We will denote by **FindSolution()** the function which performs this search mechanism.

4.2 Combining CP and Local Search to Improve Solutions

Experiments carried out on real data showed that the search process presented above finds solutions satisfying all the placement and local production constraints in reduced computing time(1 to 2 seconds). We then developed a Local Search approach to find outage schedules minimizing a heuristic criterion based on demand satisfaction cost.

Strategy. Basically, the idea is to compute a target NPP outage schedule which minimizes the cost of the conventional plant production.

First, note that because we know the number of outages on the study period and their durations, it is possible to estimate the unavailability rate of the nuclear park, due to outages. Once completed with some other kinds of unavailability (random faults, power decreasing laws, ...), the unavailability rate (μ) enables us to estimate the maximal nuclear energy (NE) producible by the whole nuclear park on the study period : $NE = \top.(1-\mu)\sum_{i=1}^{n} E_i$ where $\sum_{i=1}^{m} E_i$ is the weekly maximal energy producible by the NPP park.

Because the nuclear energy is insufficient to satisfy demand on the study period, we have to apply a certain complementary non-nuclear production (CE) in order to fulfil demand : $CE = \sum_{t=1}^{\top} Dem_t - NE = \sum_{t=1}^{\top} v(t)$. The price to pay on the whole study period for this non-nuclear production is $\sum_{t=1}^{\top} g(v(t))$.

As a consequence of the convex character of the production cost function g (see Fig. 2) we have $\sum_{t=1}^{\top} g(v(t)) \geq \sum_{t=1}^{\top} g(\frac{CE}{\top})$. This demonstrates that in absence of any constraint, the best command strategy consists of applying a command that leave with always the same non-nuclear complementary production ($\frac{CE}{\top}$) to reach the demand. Note that this merely means that the *marginal cost* stays the same all along the studied period.

The quantity $\frac{CE}{\top}$ is used to compute a weekly target production for the nuclear park, TNE : $TNE = (TNE_t)_{t=1\ldots\top} = (Dem_t - \frac{CE}{\top})_{t=1\ldots\top}$.

We then search for an outage schedule such that the weekly availability of the nuclear park is sufficient to produce at each week t the quantity TNE_t.

To be able to produce quantity TNE_t at the step time t, the number of stopped NPP should not exceed a certain limit l_t initialized as follows :

$$l_t = \left\lceil n(\sum_{i=1}^{m} E_i - TNE_t) / \sum_{i=1}^{m} E_i \right\rceil .$$

Heuristic cost modeling. The vector $l = (l_t)_{t=1...\top}$ is exploited to construct an heuristic criterion used to guide the search for a good outage schedule. More precisely, we search for an outage schedule satisfying as possible the fixed unavailability limits $(l_t, t = 1...\top)$. For each fixed vector $l = (l_t)_{t=1...\top}$ of unavailability limits, the following constraints are added to the system.

$$\varphi_t(T) \leq L_t + \eta_t,\ t = 1...\top \tag{10}$$

where,

- T : vector of the start time dates of the outages $(T = (T_{ik})_{i=1...m,k=1...n_i})$;
- $\varphi_t(T) = |\{\langle i, k\rangle : T_{ik} \leq t < T_{ik} + d_i^k\}|$: number of stopped plants at the step time t;
- L_t : finite domain variable whose upper bound is equal to l_t;
- η_t: finite domain variable.

Constraints (10) above mean that at each week t the nuclear park unavailability must be less or equal to the fixed limit l_t. Otherwise, the additional unavailability η_t will be considered as constraint violation cost. They are modelled by the cumulative global constraint.

The heuristic cost criterion introduced above takes thus the following form :

$$Cost = \eta.\mathbb{1} = \sum_{t=1}^{\top} \eta_t. \tag{11}$$

We will denote the obtained optimization problem as follows :

$$CSOP(l, Cost) \begin{cases} Cost = \eta.\mathbb{1} = \sum_{t=1}^{\top} \eta_t \\ \varphi(T) \leq L + \eta \\ CSP(T) \end{cases} \tag{12}$$

Local search. To find an outage schedule minimizing the heuristic cost criterion introduced above, we use a Local Search approach based on very simple noising technique [3]. More precisely, it exploits the following observations :

1. If additional constraints are added to the problem (12), the search space will be reduced and the problem can become easier to solve ;
2. If we consider another objective (for example by noising the vector of unavailability limits), the good solutions can be different from the ones currently obtained.

The former observation is used to construct neighborhoods in which good solutions can be computed by standard CP, the latter is used to escape from local optima.

The optimization process begins by posting the constraints system (12) and storing the obtained search state, then the first solution is searched by the **Find-Solution()** function.

Let $v = (v_{ik})_{i=1\ldots m, k=1\ldots n_i}$ represents the current solution vector and c its cost (the current lower bound of $Cost$). In order to improve this solution, a repair method performing local moves takes place.

A local move consists of changing the start time date of some outages of a single NPP. Let j be this plant. After restoring the stored search state, the constraints $\{T_{ik} = v_{ik},\ \forall \langle i,k \rangle : i \neq j\}$ are added to the system (12). We obtain the following system, defining a neighborhood:

$$N(j,v,c) \begin{cases} CSOP(l, Cost) \\ T_{ik} = v_{ik},\ \forall \langle i,k \rangle : i \neq j \end{cases} \tag{13}$$

System (13) is then solved to optimality with a branch and bound search. If a solution v' of cost $c' < c$ is found, it is considered as the new current solution ($v \leftarrow v', c \leftarrow c'$). The process is iterated by considering another plant j' and solving the new neighborhood $N(j', v, c)$ and so on until a local optimum is found. Note that when we solve the neighborhood sub-problems the local production constraints are taken into account. We denote by **Repair**(v, c) the function performing these local moves starting from an initial solution. The obtained local optimum and its cost are then saved : $v^* \leftarrow v, c^* \leftarrow c$.

Obviously, the current local optimum depends on the vector of unavailability limits l. Indeed, if the vector l were initialized differently, the local optimum found by **Repair**(v, c) could be different from the current one. Thus, in order to escape from the current local optimum we post the constraints $L_t \leq l_t - 1$, $1 \leq t \leq \top$. The objective is now to find outage schedule satisfying as possible the new unavailability limits $l - \mathbb{1}$. We denote $CSOP(l - \mathbb{1}, Cost)$ the obtained problem. It has the same set of solutions that of $CSOP(l, Cost)$. Since the objective is noised, the **Repair**(v, c) method can now performs locals moves and produces a local optimum v for $CSOP(l - \mathbb{1}, Cost)$.

We dispose now from two solution : the vector v^* which is a local optimal for $CSOP(l, Cost)$ and the vector v which is a local optimum for $CSOP(l - \mathbb{1}, Cost)$. The constraints posted over L are then deleted and the repair method **Repair**(v, c) is restarted again. The vector v becomes a new local optimum for $CSOP(l, Cost)$. If its cost c is less than that of v^* ($c < c^*$), it is considered as the best solution encountered : $v^* \leftarrow v, c^* \leftarrow c$. The process is iterated until a stop criterion is reached. The chosen stop criterion is the number of time that the objective is noised without improving the current best solution.

5 Putting It All Together

Our approach finds outage schedules leading to satisfy the demand at lower cost. It does not determine a production plan. That is done by an existing module named **"PLAFIGE"**. It is the coordination module used by the decomposition based approach presented in the section 3. It is worth noting that

The following **OSOPAN**[2] procedure sums up the entire resolution process:

```
OSOPAN

1. %Problem stating
   $post(CSOP(L, Cost));$
2. %Outage schedule searching
     – $(v, c) \leftarrow$ FindSolution();
     – Repair$(v, c)$
     – $v^* \leftarrow v, c^* \leftarrow c, Stop \leftarrow N$
     – while($Stop \geq 1$)
         • $post(L \leq l - \mathbb{1})$
         • Repair$(v, c)$
         • $delete(L \leq l - \mathbb{1})$
         • Repair$(v, c)$
         • if($c < c^*$) then $v^* \leftarrow v, c^* \leftarrow c, Stop \leftarrow N$
           else $Stop \leftarrow Stop - 1$
3. %Production plan searching
   $u \leftarrow PLAFIGE(v*)$
```

The architecture of the obtained tool **(OSOPAN)** consists of three separated components. In the first one, the problem is stated. The second component serves to determine, relatively to a heuristic cost criterion a good outage schedule. At this stage, all the start time dates of the outages are fixed and the remainder variables are continuous. Taking into account the fixed outage schedule, the third component, named **"PLAFIGE"** and based on linear programming techniques, determines an optimal production plan. This leads to evaluate in a precise way the quality of the outage schedule and produces interesting marginal costs currently used by EDF operational staff to perform local moves in order to improve solutions.

This architecture allows to focus on the second component performing Local Search. Indeed, our local search is very simple and can be improved. In particular, local moves can be based on marginal costs provided by the module **"PLAFIGE"**.

6 Results and Perspectives

The approach described in this paper has been entirely implemented[3] and tested against the tools currently used by operational staff to compute NPP outage schedules. Let us call CM1 and CM2 the two MIP based resolution methods implemented in these tools and outlined section 3.

[2] Outil pour la Satisfaction et l'Optimisation du Problème des Arrêts Nucléaires.

[3] We have used the CHIP C++ library [12].

First of all, solutions found by our new approach are comparable to the cost (6) defined section 3. This justifies the heuristic cost criterion we stated, and the Local Search procedure we use to optimize it. But in addition, we have the following major valuable benefits :

– Computing time is similar to the one exhibited by CM1, namely about 5 mn for a five years long study period, *but with all the placement constraints satisfied.*
– Computing time is much better that the one exhibited by CM2 to find comparable solutions.
– Declarativity of the resolution process is drastically improved. Firstly, new placement constraints can be easily added. This is a typical benefits of CP approach. In addition, and more fondamentally the heuristic cost criterion used by Local Search can be sophisticated without altering the overall architecture.

As an illustration of the latter benefit above, current works consists of using the precise tool based on Linear Programming (**PLAFIGE**) to estimate the cost of the solutions in terms of distance to an optimal production plan rather than to the target nuclear park availability vector which proceeds from it. This should improve quality of the solutions, with respect to the criterion (6).

References

1. E. Aarts and J. K. Lenstra. Local search in Combinatorial Optimisation. *John Wiley & Sons.* (1997).
2. K. R. Apt. Principles of Constraint Programming", *Cambridge University Press.* (2003).
3. I. Charon and O. Hudry. The noising method: A new method for combinatorial optimization. *Operations Research Letters.* 14:133-137 (1993).
4. F. Focacci, F. Laburthe and A. Lodi. Local Search and Constraint Programming. In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics, volume 57 of *International Series in Operations Research and Management Science.* Kluwer Academic Publisherss, Norwell, MA, (2002).
5. F. Glover, M. Laguna, É. Taillard, and D. de Werra. Tabu Search. *Annals of Operations Research.* 41:3-28 (1993).
6. S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. Optimization by Simulated Annealing. *Scienxce.* 220:671-680 (1983).
7. L. Michel and P. Van Hentryck. Localizer: A modeling language for local search. *CP'97*: 237-251 (1997).
8. L. Michel and P. Van Hentenryck: A constraint-based architecture for local search. *OOPSLA'02*: 83-100 (2002).
9. A. Nareyek. Using Global Constraints for Local Search. *DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization:* 1-18 (1998).
10. G. Pesant and M. Gendreau. A Constraint Programming Framework for Local Search Methods. *J. Heuristics* 5(3): 255-279 (1999).
11. P. Van Hentryck and L. Michel. Control Abstractions for Local Search. *CP'03*: 65-80 (2003).
12. CHIP User's Guide (2004).

Generalized Arc Consistency for Positive Table Constraints

Christophe Lecoutre[1] and Radoslaw Szymanek[2]

[1] CRIL-CNRS FRE 2499,
Université d'Artois, Lens, France
lecoutre@cril.univ-artois.fr
[2] Cork Constraint Computation Centre
University College, Cork, Ireland
radsz@4c.ucc.ie

Abstract. In this paper, we propose a new algorithm to establish Generalized Arc Consistency (GAC) on positive table constraints, i.e. constraints defined in extension by a set of allowed tuples. Our algorithm visits the lists of valid and allowed tuples in an alternative fashion when looking for a support (i.e. a tuple that is both allowed and valid). It is then able to jump over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. Our approach, that can be easily grafted to any generic GAC algorithm, admits on some instances a behaviour quadratic in the arity of the constraints whereas classical approaches, i.e. approaches that focus on either valid or allowed tuples, admit an exponential behaviour. We show the effectiveness of this approach, both theoretically and experimentally.

1 Introduction

Arc Consistency (AC) plays a central role in Constraint Programming (CP). It is an essential component of the Maintaining Arc Consistency (MAC) algorithm, which is commonly used to solve instances of the Constraint Satisfaction Problem (CSP). It is also at the heart of a stronger consistency called Singleton Arc Consistency (SAC), which has recently attracted a lot of attention (e.g., [1,7]).

A CSP instance is arc consistent iff there is no variable value pair which violates any of the binary constraints. Many algorithms for establishing AC have been proposed. They can be classified into two categories coarse-grain and fine-grain. The former one consists of algorithms (e.g. AC3 [10], AC2001/3.1 [4]), which reason about arcs, i.e. pairs composed of a constraint and a variable, to infer inconsistent values. The latter one contains algorithms whose reasoning is based on values (e.g. AC7 [2]). The algorithms from both classes share a characteristic of being generic since they can be applied to any kind of constraints (i.e. constraints defined by a predicate or a list of tuples).

In many applications, non-binary constraints naturally arise. For such constraints, AC is then said to be Generalized (GAC). Although GAC extensions of the generic algorithms mentioned above exist, it is not always relevant to use them since specific approaches to establish GAC by exploiting the semantics/structure of the constraints can be far more efficient.

In this paper, we propose a new algorithm to establish GAC on positive table constraints. A table constraint is a constraint which is defined in extension by a set of tuples. A tuple represents a combination of values for constraint variables that is allowed in case of positive table constraint and disallowed in case of negative table constraint. Table constraints are commonly used in configuration applications or applications related to databases.

The approach that we propose is a refinement of two generic approaches, which we call in this work GAC-valid and GAC-allowed. GAC-valid iterates over valid tuples (i.e. tuples that can be built from the current domains of constraint variables) in order to find supports. A tuple is called a support iff it is valid and allowed. GAC-allowed iterates over allowed tuples (i.e. combinations of values which are allowed by a constraint) in order to find supports. Roughly speaking, GAC-valid and GAC-allowed correspond respectively to GAC-schema-predicate and GAC-schema-allowed presented in [3].

We use the following example to illustrate the potential drawbacks of both schemas with respect to positive table constraints. Let us consider a constraint C involving r variables, $X_1, ..., X_r$, such that the domain of each variable is initially $\{0, 1, 2\}$. Let us assume (see Figure 1.a) that there are exactly 2^{r-1} tuples allowed by C (all tuples allowed by C correspond to the binary representation of all values between 0 and $2^{r-1} - 2$ in addition to the tuple $(2, 2, ..., 2, 2)$).

```
(0,0,...,0,0)                  (0,1,...,1,1)
(0,0,...,0,1)                  (0,1,...,1,2)
(0,0,...,1,0)                  (0,1,...,2,1)
...                            ...
(0,1,...,1,0)                  (0,2,...,2,2)
(2,2,...,2,2)
     (a)                            (b)
```

Fig. 1. A list of allowed tuples (a) and a list of valid tuples (b)

Assume that, due to propagation caused by other constraints, domains of all variables have been reduced to $\{1, 2\}$ except for variable X_1 whose domain has been reduced to $\{0\}$. After this propagation, there are exactly 2^{r-1} valid tuples (see Figure 1.b) that can be built for C.

Let us consider that we have to check first if (X_1,0) is consistent with C. The time complexity of determining that (X_1,0) violates C when one uses GAC-valid is $\Omega(2^{r-1})$. Indeed, to prove it, we need to consider 2^{r-1} valid tuples. GAC-allowed has also time complexity $\Omega(2^{r-1})$ since it has to consider $2^{r-1} - 1$ allowed tuples to prove that (X_1,0) violates C.

The behaviour of both schemas does not seem satisfactory as it is immediate that (X_1,0) violates C. The solution that we propose is to alternate the visits to both lists of valid and allowed tuples. The idea is to jump over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. Let us consider our example, which checks if (X_1,0) violates C. The first task to be performed is to compute, in O(r), the first valid tuple $t = (0, 1, ..., 1, 1)$. Afterwards, the first allowed tuple t' greater than or equal to t is searched for. This task involves, using a dichotomic search, $log_2(2^{r-1})$ comparisons of tuples, which is O(r^2) as comparing two tuples is O(r). As no such tuple exists for (X_1,0), (X_1,0) is proven to violate C.

We can observe that our approach is able to skip a number of tuples that grows exponentially with the arity of the constraints, but in a different manner of that proposed in [9]. This related work, exploiting some ideas introduced in [8,12], proposed a refinement of GAC-schema-allowed [3] that iterates over allowed tuples while skipping irrelevant ones by exploiting the current domains of variables.

2 Preliminaries

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of variables and $\mathscr{C}$ a finite set of constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted *dom*(X), which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called the scope of C and denoted by *vars*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for the variables of its scope. We will respectively denote the number of variables and constraints of a CN by n and e. The arity of a constraint corresponds to the size of its scope. A solution to a CN is an assignment of values to all the variables such that all the constraints are satisfied. A CN is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN is satisfiable. A CSP instance is then defined by a CN, and solving it involves either finding one (or more) solution or determining its unsatisfiability.

A CSP instance can be solved by modifying the network using inference or search methods. Usually, domains of variables are reduced by removing inconsistent values, i.e. values that cannot occur in any solution. The initial domain of a variable X will be denoted dom$^{init}(X)$ whereas the current domain of X will be simply denoted dom(X). For each r-ary constraint C such that vars(C) = $\{X_1, \ldots, X_r\}$, we have: $rel(C) \subseteq \prod_{i=1}^{r} dom^{init}(X_i)$ where $\prod$ denotes the Cartesian product. Also, for any element $t = (a_1, \ldots, a_r)$, called r-tuple or more simply tuple, of $\prod_{i=1}^{r} dom^{init}(X_i)$, $t[X_i]$ (and also $t[i]$) denotes the value a_i. We can now introduce the notion of support.

Definition 1. *Let C be a r-ary constraint such that vars(C) = $\{X_1, \ldots, X_r\}$, a r-tuple t of $\prod_{i=1}^{r} dom^{init}(X_i)$ is allowed by C iff $t \in rel(C)$, valid wrt C iff $\forall X_i \in$ vars(C), $t[X_i] \in$ dom(X_i), and a support in C iff it is allowed and valid.*

A tuple t is a support of (X_i, a) in C when t is a support in C such that $t[X_i] = a$. Note that a solution guarantees the existence of a support in each constraint. In addition, assuming a total order on variables, tuples can be lexicographically ordered. Two r-tuples t and t' are such that $t \prec t'$ iff $\exists i \in 1..r \mid t[i] < t'[i] \wedge \forall j \in$

$1..i-1, t[j] = t'[j]$. The i^{th} variable of the scope of a constraint C will be denoted by $var(C,i)$. To simplify the presentation of some algorithms, we introduce one special symbol $\top$ such that any tuple t verifies $t \prec \top$.

It is possible to filter domains by considering some properties of constraint networks. Generalized Arc Consistency (GAC) remains the most important: it guarantees the existence of a support for each value in each constraint. Establishing GAC on a CN involves removing all generalized arc inconsistent values.

Definition 2. *Let $P = (\mathscr{X},\mathscr{C})$ be a CN. A pair (X,a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is Generalized Arc Consistent (GAC) iff $\forall C \in \mathscr{C} | X \in vars(C)$, there exists a support of (X,a) in C. P is GAC iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X,a) is GAC.*

3 Representing Finite Domains

Before presenting different approaches to establish GAC on positive table constraints, we need to introduce a precise description (in the general context of a backtracking search) of domains representation. Indeed, this is partly required for our complexity analysis given later.

To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a decision (most of the time, a variable assignment) is performed followed by a filtering process called constraint propagation. It is then necessary to keep track of values removed during search, as upon backtracking, they have to be restored. In fact, for each removed value a, we need to record the level at which a has been removed. This mechanism is called trailing and is used in most current CP systems [13]. Considering a constant denoted NO whose value is -1, to represent the current state of a domain (which is initially composed of d values) during search, we introduce the following structures:

- *value* is an array of size d which contains the set of values
- *absent* is an array of size d which indicates which values are removed from the domain. More precisely, $absent[i] = NO$ indicates that $value[i]$ belongs to the current domain and $absent[i] = k$ (≥ 0) indicates that $value[i]$ has been removed at level k of search.
- *next* is an array of size d which allows to link (from first to last) all values of the current domain. When $absent[i] = NO$, $next[i]$ gives the index $j > i$ of the next value in the current domain (we have $absent[j] = NO$ and $\forall k \in i+1..j-1, absent[k] \neq NO$), or NO if $value[i]$ is the last value.
- *prev* is an array of size d which allows to link (from last to first) all values of the current domain. When $absent[i] = NO$, $prev[i]$ gives the index $j < i$ of the previous value in the current domain (we have $absent[j] = NO$ and $\forall k \in j+1..i-1, absent[k] \neq NO$), or NO if $value[i]$ is the first value.
- *prevAbsent* is an array of size d which allows to link all values that do not belong to the current domain. When $absent[i] \neq NO$, $prevAbsent[i]$ gives the index j of the value removed during search just before $value[i]$, or NO if $value[i]$ is the first removed element.

Algorithm 1. removeValue(index : int, level : int)

```
1: $absent[index] \leftarrow level$
2: $prevAbsent[index] \leftarrow lastAbsent$
3: $lastAbsent \leftarrow index$
4: if prev[index] $= NO$ then $first \leftarrow next[index]$
5: else $next[prev[index]] \leftarrow next[index]$
6: if next[index] $= NO$ then $last \leftarrow prev[index]$
7: else $prev[next[index]] \leftarrow prev[index]$
```

Algorithm 2. reduceTo(index : int, level : int)

```
1: $current \leftarrow first$
2: while $current \neq NO$ do
3:     if $current \neq index$ then removeValue(current, level)
4:     $current \leftarrow next[current]$
5: end while
```

We also need three variables denoted $first$, $last$ and $lastAbsent$ which respectively indicate the indices of the first value, the last value and the last removed value. Using $first$ and $last$ variables in conjunction with $next$ and $prev$ arrays, we obtain a behaviour similar to a doubly-linked list. Using $lastAbsent$ and $prevAbsent$, we obtain a behaviour similar to a stack (last-in first-out structure). The initialisation of these structures (not described in this paper, due to lack of space) is rather straightforward.

Then, one can wonder how to get the position (index) of a given value. In fact, in the context of establishing GAC wrt positive table constraints, it is never necessary as we can always reason about the indices of values. In a more general context, one can introduce a hash map which allows, under reasonable assumptions, to obtain the index of a given value in constant time. For more information, see Section 4 in [6] and implementation details in [2]. From now on, to simplify the presentation of algorithms and without any loss of generality, we will assume that values and indices match, i.e. $\forall i \in 1..d, value[i] = i$.

When a value is removed by propagation, the function $removeValue$ (see Algorithm 1) is called. It just updates the stack of removed values (lines 1 to 3) and the doubly-linked list of present elements (lines 4 to 7). When an assignment is performed during search, the function $reduceTo$ (see Algorithm 2) is called. It involves removing all values different from the given one. When the solver backtracks up to level k, the function $restoreUpto$ (see Algorithm 4) is called. It adds values, which have been removed at a level greater than or equal to k, back to the domain. Indeed, for such elements, the function $addValue$ is called. Hence, the stack is updated (lines 1 and 2) as well as the doubly-linked list (lines 3 to 6). Here, it is interesting to note that as removals are managed using a last-in first-out structure (stack), we automatically obtain correct indexes at $prev[index]$ and $next[index]$ when a value is restored. Indeed, they are never overwritten once $value[index]$ is removed.

Algorithm 3. addValue(index : int)

```
1: $absent[index] \leftarrow NO$
2: $lastAbsent \leftarrow prevAbsent[index]$
3: if prev[index] $= NO$ then $first \leftarrow index$
4: else $next[prev[index]] \leftarrow index$
5: if next[index] $= NO$ then $last \leftarrow index$
6: else $prev[next[index]] \leftarrow index$
```

Algorithm 4. restoreUpto(level : int)

```
1: $current \leftarrow lastAbsent$
2: while $current \neq NO \wedge absent[current] \geq level$ do
3:    addValue(current)
4:    $current \leftarrow prevAbsent[current]$
5: end while
```

The space complexity of this representation is $\theta(|dom(X)|)$ for any variable X, which is optimal. The time complexity of all elementary operations (determining if a value is present, getting next value, previous value, etc.) is O(1). As a consequence, the time complexity of $removeValue$ and $addValue$ is O(1).

4 Establishing Generalized Arc Consistency

To establish generalized arc consistency on a given constraint network, one can use a coarse-grained algorithm such as GAC3 [11] or GAC2001 [4]. To simplify and without any loss of generality for what follows, we only present GAC3. We first introduce the general schema (to be compared with the fine-grained one introduced in [3]) of the algorithm and then describe a general implementation.

4.1 GAC3

To establish generalized arc consistency on a given CN involving any kind of constraints, the coarse-grained *GAC* algorithm (Algorithm 5) can be called. It returns *true* when the given CN can be made GAC. Initially, all pairs (C, X), called arcs, are put in a set Q. Once Q has been initialized, each arc is revised in turn (line 4), and when a revision is effective (at least, one value has been removed), the set Q has to be updated (line 6). A revision is performed by a call to the function *revise* specific to the chosen coarse-grained GAC algorithm, and entails removing values that have become inconsistent with respect to C. This function returns *true* when the revision is effective. The algorithm is stopped when a domain wipe-out occurs (line 5) or the set Q becomes empty. For GAC3 [10,11], each revision is performed by a call to the function *revise* depicted in Algorithm 6. This function iteratively calls the function *seekSupport* which determines from scratch whether or not there exists a support of (X, a) in C.

Algorithm 5. GAC (P = $(\mathscr{X},\mathscr{C})$: Constraint Network): Boolean

```
1: $Q \leftarrow \{(C,X) \mid C \in \mathscr{C} \wedge X \in vars(C)\}$
2: while $Q \neq \emptyset$ do
3:     pick and delete $(C,X)$ from $Q$
4:     if revise($C$,$X$) then
5:         if $dom(X) = \emptyset$ then return false
6:         $Q \leftarrow Q \cup \{(C',Y) \mid C' \in \mathscr{C}, \{X,Y\} \subseteq vars(C'), C' \neq C, Y \neq X\}$
7: return true
```

Algorithm 6. revise(C : Constraint, X : Variable) : Boolean

```
1: nbElements $\leftarrow \mid dom(X) \mid$
2: for each $a \in dom(X)$ do
3:     if seekSupport$(C,X,a) = \top$ then remove $a$ from $dom(X)$
4: return nbElements $\neq |dom(X)|$
```

4.2 GAC3-Valid

We naturally assume that it is always possible to check that a given tuple is allowed by a constraint. Hence, a general approach (called GAC3-valid) to implement *seekSupport* is to iterate the set of valid tuples until an allowed one is found. This approach is the coarse-grained correspondence of the general schema described in [3]. It can be used to deal with any kind of constraints.

In order to find a support, valid tuples are then considered. We will denote $valid(C,X,a)$ the set of valid tuples involving (X,a) built from C. We have $valid(C,X,a) = \{t \in \prod_{Y \in vars(C)} dom(Y) \mid t[X] = a\}$. To iterate valid tuples, we introduce two functions called $setFirstValid$ and $setNextValid$. But first, remember that we assume a total order on the scope of each constraint such that tuples can be ordered using a lexicographic order $\prec$. Also, for each domain, one can exploit a linked list (see Section 3) of present elements using the variable $first$ and the array $next$. The call $setFirstValid(C,X,a)$ (see Algorithm 7) returns the smallest valid tuple t built from C such that $t[X] = a$. Its time complexity is $\theta(r)$ where r is the arity of the constraint C. Indeed, to build t, it suffices to get the first element of all domains associated with variables (except X) involved in C. The call $setNextValid(C,X,a,t)$ (see Algorithm 8) returns either the smallest valid tuple t' built from C such that $t \prec t'$ and $t'[X] = a$, or $\top$ if it does not exist. Its worst-case time complexity is O(r). It is important to remark that $setNextValid$ is always called here with a parameter t being a valid tuple. Remember also that we assume to simplify (and without any loss of generality) that values and indices (of values) match. Hence, $t[Y]$ at lines 2 and 4 of Algorithm 8 can be understood as representing the index of a value.

As an illustration, consider a ternary constraint C involving variables X, Y and Z. We have $var(C,1) = X$, $var(C,2) = Y$ and $var(C,3) = Z$. If dom$(X) = \{1,4,5\}$, dom$(Y) = \{2,4\}$ and dom$(Z) = \{1,2\}$, then we obtain: $setFirstValid(C,Y,4) = (1,4,1)$, $setNextValid(C,Y,4,(1,4,1)) = (1,4,2)$ and $setNextValid(C,Y,4,(1,4,2)) = (4,4,1)$.

Algorithm 7. setFirstValid(C, X, a) : Tuple

```
1: $t[X] \leftarrow a$
2: for each variable Y $\in vars(C)$ such that $Y \neq X$ do $t[Y] \leftarrow dom(Y).first$
3: return $t$
```

Algorithm 8. setNextValid(C, X, a, t) : Tuple

```
1: for each $Y = var(C, i) \mid Y \neq X$ with $i$ ranging from $|vars(C)|$ down-to 1 do
2:     if $dom(Y).next[t[Y]] = NO$ then $t[Y] \leftarrow dom(Y).first$
3:     else
4:         $t[Y] \leftarrow dom(Y).next[t[Y]]$
5:         return t
6: return $\top$
```

Looking at function *seekSupport-valid* (see Algorithm 9) which implements *seekSupport*, we remark that a constraint check corresponds to determining if $t \in rel(C)$. It can be implemented in different ways (by evaluating a Boolean expression, querying a database, looking for a tuple in a list, etc.). The proof of the following proposition is immediate as in the worst-case, each valid tuple has to be checked.

Proposition 1. *The worst-case time complexity of seekSupport-valid*(C, X, a) *is O(V.K) where* $V = |valid(C, X, a)|$ *and K is the worst-case time complexity of performing a constraint check.*

5 Establishing GAC on Positive Table Constraints

In this section, we first discuss about positive table constraints. Then, we present two classical algorithms to establish GAC on positive table constraints that respectively involve iterating lists of valid tuples and lists of allowed tuples. Finally, we introduce an original algorithm that involves visiting, in turn, both lists, looking for a valid tuple, then for an allowed tuple, and so on. It can allow skipping a number of valid tuples and a number of allowed tuples that grows exponentially with the arity of the constraints.

5.1 Positive Table Constraints

A positive table constraint is a constraint given in extension and defined by a set of allowed tuples. Such constraints arise in practice in configuration problems, and more generally, in problems whose data come from databases. The set of allowed tuples associated with any constraint C is a table denoted $allowed(C)$. The worst-case space complexity to record this set is O$(r.|allowed(C)|)$ where r denotes the arity of C and $|allowed(C)|$ the size of the table (i.e. the number of allowed tuples).

Most of the time, we are interested in the list of allowed tuples involving a pair (X,a). We will denote this list $allowed(C, X, a)$. It is then interesting

Algorithm 9. seekSupport-valid(C, X, a) : Tuple

```
1: $t \leftarrow setFirstValid(C, X, a)$
2: while $t \neq \top$ do
3:     if $t \in rel(C)$ then return $t$
4:     $t \leftarrow setNextValid(C, X, a, t)$
5: return $\top$
```

to introduce an array of pointers to such supports for any pair (X,a). Each support being referenced exactly r times, the space complexity of these new arrays is $O(r.|allowed(C)|)$. So, the overall worst-case space complexity remains $O(r.|allowed(C)|)$. Assuming that each list is ordered (according to the lexicographic order of referenced tuples), the worst-case time complexity of checking that a tuple t involving (X,a) is allowed is $O(r.log(|allowed(C, X, a)|))$.

An alternative is to introduce a hash map to "directly" access allowed tuples (proposed in [3] wrt negative table constraints). The worst-case space complexity remains $O(r.|allowed(C)|$. Besides, if the hash function is $O(r)$ and disperses the elements properly, the worst-case time complexity of performing a constraint check is only $O(r)$. For our experimentation, we will not consider this alternative.

5.2 GAC3-Valid

This approach involves iterating the set of valid tuples until an allowed one is found. It has been presented at subsection 4.2. Here, we simply describe the way a constraint check is performed. In fact, the test $t \in rel(C)$ at line 3 of Algorithm 9 corresponds to $t \in allowed(C, X, a)$. As mentioned above, it is $O(r.log(|allowed(C, X, a)|))$.

Proposition 2. *For a r-ary positive table constraint C, the worst-case time complexity of seekSupport-valid(C, X, a) is O(V.r.log(S)) where we have $V = |valid(C, X, a)|$ and $S = |allowed(C, X, a)|$.*

Corollary 1. *If one uses a hash map with a hash function in O(r) that has dispersed properly allowed tuples, the worst-case time complexity of calling seekSupport-valid(C, X, a) is O(V.r).*

5.3 GAC3-Allowed

This second approach involves iterating allowed tuples until a valid one is found. This approach, called GAC3-allowed, can be seen as the coarse-grained correspondence of GAC-scheme-allowed proposed in [3]. In order to find a support, allowed tuples are then considered. To do this, we introduce two functions denoted *setFirstAllowed* and *setNextAllowed*. *setFirstAllowed*(C, X, a) returns the smallest support t of C such that $t[X] = a$ while *setNextAllowed*(C, X, a, t) returns either the smallest support t' of C such that $t \prec t'$ and $t'[X] = a$, or $\top$ if it does not exist. Considering that we have an ordered list for $allowed(C, X, a)$ and a position of tuple t, which is not explicitly used in *setNextAllowed*, the time complexity of any such call is $O(1)$.

Algorithm 10. seekSupport-allowed(C, X, a) : Tuple

```
1: $t \leftarrow setFirstAllowed(C, X, a)$
2: while $t \neq \top$ do
3:     if $seekInvalidPosition(C, t) = NO$ then return $t$
4:     $t \leftarrow setNextAllowed(C, X, a, t)$
5: return $\top$
```

Algorithm 11. seekInvalidPosition(C,t) : int

```
1: for each variable $Y = var(C, i)$ with $i$ ranging from 1 to $|vars(C)|$ do
2:     if t[Y] $\notin$ dom(Y) then return $i$
3: return $NO$
```

Looking at function *seekSupport-allowed* (see Algorithm 10) which implements *seekSupport*, we remark that a validity check is performed for each allowed tuple until a support is found. It just involves determining if all values of the given tuple belong to current domains. The function *seekInvalidPosition* (see Algorithm 11) returns NO if the given tuple is valid and the position of the first invalid value, otherwise. The application of this position will be shown later. The worst-case time complexity of calling *seekInvalidPosition* is O(r).

Proposition 3. *For a r-ary positive table constraint C, the worst-case time complexity of seekSupport-allowed(C, X, a) is O(S.r) with S$= |allowed(C, X, a)|$*

5.4 GAC3-Valid+Allowed

The original approach that we propose now involves visiting both lists of valid and allowed tuples. The idea is to jump over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. It is described in Algorithm 12. At each execution of the while loop body, a valid tuple is considered (initially, the first one is computed). The call *binarySearch* at line 3 performs a dichotomic search which returns the smallest allowed tuple t' of C such that $t \preceq t'$ and $t'[X] = a$. If $t' = \top$, no support can be found anymore (line 4). Otherwise, t' corresponds to an allowed tuple whose validity must be checked (line 5). If $seekInvalidPosition(C, t')$ returns NO, it means that t' is also valid, and so, it can be returned (line 6) since it is a support. If t' is not valid, we have to execute $setNextValid$ (line 7) to compute the smallest valid tuple t built from C such that $t' \prec t$ and $t[X] = a$ (or $\top$ if it does not exist). This function, described by Algorithm 13, is different from the one described by Algorithm 8 as the tuple given as parameter is not valid. First (lines 1 and 2), any variable Y whose position in C is strictly greater than *limit* must be given in t the first valid value in $dom(Y)$. Then, to find a tuple strictly greater than t, we have to iteratively (lines 6 and 7 of Algorithm 13) look for the next (index of) value following $t[Y]$ where Y is the first encountered variable such that $t[Y] < dom(Y).last$. In our implementation, we know that if $t[Y] \in dom(Y)$ then $dom[Y].next[t[Y]]$ gives the smallest value of $dom(Y)$ greater than $t[Y]$, but we

Algorithm 12. seekSupport-valid+allowed(C, X, a) : Tuple

```
1: $t \leftarrow setFirstValid(C, X, a)$
2: while $t \neq \top$ do
3:     $t' \leftarrow binarySearch(allowed(C, X, a), t)$
4:     if $t' = \top$ then return $\top$
5:     $j \leftarrow seekInvalidPosition(C, t')$
6:     if $j = NO$ then return $t'$
7:     $t \leftarrow setNextValid(C, X, a, t', j)$
8: return $\top$
```

Algorithm 13. setNextValid(C, X, a, t, limit) : Tuple

```
1: for each $Y = var(C, i) \mid Y \neq X$ with $i$ ranging from $limit$+1 to $|vars(C)|$ do
2:     $t[Y] \leftarrow dom(Y).first$
3: for each $Y = var(C, i) \mid Y \neq X$ with $i$ ranging from $limit$ down-to 1 do
4:     if $t[Y] \geq dom(Y).last$ then $t[Y] \leftarrow dom(Y).first$
5:     else
6:         $t[Y] \leftarrow dom(Y).next[t[Y]]$
7:         while $dom(Y).absent[t[Y]] \neq NO$ do $t[Y] \leftarrow dom(Y).next[t[Y]]$
8:         return t
9: return $\top$
```

also know (the proof is omitted) that if $t[Y] \notin dom(Y)$ then $dom[Y].next[t[Y]]$ gives a value less than or equal to the smallest value of $dom(Y)$ greater than $t[Y]$. If $setNextValid$ returns $\top$, it means that it was not possible to find a valid tuple greater than t'.

As an illustration, consider a 5-ary constraint C involving variables V, W, X, Y and Z. We have $var(C, 1) = V$, $var(C, 2) = W$ etc. Let us assume that we have dom(V) = $\{1, 3\}$, dom(W) = $\{3, 4\}$, dom(X) = $\{1, 4, 5\}$, dom(Y) = $\{2, 4\}$ and dom(Z) = $\{1, 2\}$. Then, calling $seekInvalidPosition(C, (3, 4, 4, 2, 2))$ returns NO. Calling $seekInvalidPosition(C, (3, 4, 6, 2, 2))$ returns 3 (the position of X) and $setNextValid(C, Y, 2, (3, 4, 6, 2, 2), 3)$ returns $\top$ since it is not possible to find a valid tuple strictly greater than $(3, 4, 6, *, *)$. Finally, calling $seekInvalidPosition(C, (3, 3, 6, 2, 3))$ returns 3 (the position of X) and calling $setNextValid(C, Y, 2, (3, 3, 6, 2, 2), 3)$ returns $(3, 4, 1, 2, 1)$.

Proposition 4. *For a r-ary positive table constraint C, the worst-case time complexity of seekSupport-valid+allowed(C, X, a) is O($N.(d+r.log(S))$) where N is the number of sequences of valid tuples containing no support, d is the greatest domain size and $S = |allowed(C, X, a)|$.*

Proof. The worst-case time complexity of $binarySearch$ is O($r.log(S)$) with $S = |allowed(C, X, a)|$. The worst-case time complexity of $seekInvalidPosition$ is O(r). The overall worst-case time complexity of $setNextValid$ is O($r+d$). The overall worst-case time complexity for one execution of loop body is then O($d+r.log(S)$). Let us prove now that N bounds the number of turns of the main loop. In fact, if it was not the case, it would mean that there will be two turns involving the same sequence of valid tuples containing no support. It is not possible as, following the algorithm, it would imply that a support belongs to this sequence. □

Observation 1 *There exist r-ary positive table constraints such that, for some current domains of variables, applying GAC3-valid+allowed is O(r^2) whereas applying GAC3-valid or GAC3-allowed is O(2^{r-1}).*

Let us consider the example of the introduction. Applying GAC is equivalent to call *seekSupport*$(C, X_1, 0)$ since, after removing $(X_1,0)$, we obtain a domain wipe-out. As shown in the introduction, calling *seekSupport-valid+allowed* for $(C, X_1, 0)$ is $O(r^2)$ while calling *seekSupport-valid* or *seekSupport-allowed* is $O(2^{r-1})$. Note that, here, we have $N = 1$.

Related Work. The algorithm that we propose is closely related to the approach introduced in [9] which skips irrelevant allowed tuples based on a reasoning about lower bounds derived from information about valid tuples. More precisely, when looking for a support of (X,a), the principle in [9] is to jump (by exploiting current domains) from an allowed tuple t (which is not valid) to a tuple lb which is a lower bound of the smallest valid tuple t' such that $t \prec t'$ and no support of (X,a) strictly less than t' exists. Then, an allowed tuple involving (X,a) and greater than lb is computed using a function *nextIn* (and if this allowed tuple is not valid, one restarts this two-step procedure). One drawback of this approach is that the computed lower bound lb does not necessarily involve (X,a). This can be remedied by an additional data structure which multiplies the space complexity of the GAC-scheme by a factor d [9], or by using a more complex procedure [8].

6 Experiments

To compare the different approaches presented in this paper, we have implemented the three schemas GAC-valid, GAC-allowed and GAC-valid+allowed in Abscon. We have made our experiments with MGAC2001, that is to say, the algorithm that maintains GAC2001[1] during search. Performances have been measured in terms of the CPU time in seconds (cpu) and the number of tuples visited by MGAC (when looking for supports) during search.

First, we have tested the instances used as benchmarks for the first competition of CSP solvers (see `http://cpai.ucc.ie/05/Benchmarks.html`). Of course, we have only selected instances involving positive table constraints. Table 1 gives the results (obtained on a PC Pentium IV 2.4GHz 512MB under Linux when running MGAC2001 with *dom/wdeg* [5] as variable ordering heuristic) on some representative[2] structured instances (the maximum constraint arity is given between brackets). MGAC2001-*valid+allowed* makes it possible to greatly reduce the number of visited tuples, and the effect seems to be magnified with constraint arity increase. However, the reduction of cpu time is limited

[1] It is immediate to exploit for GAC2001 the different schemas presented wrt GAC3.

[2] Note that we obtained similar results for all other instances of the series *tsp*, *Golomb*, *geo* and *series*.

Table 1. Results obtained for some instances of the first competition of CSP solvers. GAC2001 is maintained during search according to 3 different schemas.

Instances		*allowed*	*valid*	*valid + allowed*
geo-1-*sat*	*cpu*	488	449	461
$(r = 2)$	*visits*	$1344M$	$459M$	$419M$
geo-12-*unsat*	*cpu*	822	765	778
$(r = 2)$	*visits*	$2195M$	$749M$	$684M$
tsp-20-901	*cpu*	22.0	33.6	20.1
$(r = 3)$	*visits*	$151M$	$226M$	$21M$
tsp-25-163	*cpu*	352	479	317
$(r = 3)$	*visits*	$2457M$	$3174M$	$314M$
Golomb-10-*sat*	*cpu*	39.3	37.9	37.6
$(r = 3)$	*visits*	$91M$	$69M$	$39M$
Golomb-10-*unsat*	*cpu*	465	455	449
$(r = 3)$	*visits*	$1045M$	$807M$	$478M$
series-13	*cpu*	65.3	65.0	66.4
$(r = 3)$	*visits*	$122M$	$118M$	$80M$
series-14	*cpu*	282	291	294
$(r = 3)$	*visits*	$543M$	$529M$	$354M$
renault	*cpu*	0.07	12.5	0.06
$(r = 10)$	*visits*	$286K$	$42M$	$5,437$

Table 2. Results obtained for series of 10 random instances (see [9] for similar series). GAC2001 is maintained during search according to 3 different schemas.

Instances		*allowed*	*valid*	*valid+allowed*
$\langle$14-24-2-8-0.5$\rangle$	*cpu*	0.07	0.03	0.03
	visits	$113K$	777	446
$\langle$14-24-2-16-0.5$\rangle$	*cpu*	6.5	0.7	0.6
	visits	$15M$	90589	52768
$\langle$14-24-2-24-0.5$\rangle$	*cpu*	1782	114	91
	visits	$5604M$	$13M$	$7312K$
$\langle$20-40-2-2-0.02$\rangle$	*cpu*	0.16	0.04	0.02
	visits	$377K$	3474	136
$\langle$20-40-2-4-0.02$\rangle$	*cpu*	15.5	1.9	0.2
	visits	$40M$	$287K$	16226
$\langle$20-40-2-6-0.02$\rangle$	*cpu*	2218	408	45
	visits	$5951M$	$64M$	$3446K$
$\langle$6-10-10-6-0.1$\rangle$	*cpu*	0.60	0.05	0.03
	visits	$2071K$	7016	1368
$\langle$6-10-10-8-0.1$\rangle$	*cpu*	3.9	0.3	0.1
	visits	$13M$	62396	11744
$\langle$6-10-10-10-0.1$\rangle$	*cpu*	2413	174	45
	visits	$8400M$	$42M$	$6894K$

since moving from a valid tuple to an allowed one has a cost in $O(r.log(S))$. Nevertheless, one can remark the robustness of our approach and the very bad behaviour of GAC-valid on the *renault* instance. Indeed, there is a gap of more than two orders of magnitude between this schema and the two other ones.

Next, we have reproduced instances similar to the ones presented in [9]. These random instances belong to classes of the form $\langle k, n, d, e, t\rangle$ where k, n, d, e, t denote the arity of the constraints, number of variables, uniform domain size, number of constraints and the constraint tightness, respectively. The first series include instances involving 24 Boolean variables and constraints of arity 14 and tightness 0.5 (exactly 2^{13} allowed tuples). The second series include instances

Table 3. Results obtained for series of 10 random instances. GAC2001 is maintained during search according to 3 different schemas.

Instances		*allowed*	*valid*	*valid + allowed*
⟨6-20-6-32-0.55⟩	*cpu*	6.9	0.6	0.8
	visits	$133M$	$394K$	$333K$
⟨6-20-6-36-0.55⟩	*cpu*	93.3	8.6	11.4
	visits	$1832M$	$5683K$	$4574K$
⟨6-20-8-22-0.75⟩	*cpu*	21.2	1.2	1.5
	visits	$401M$	$871K$	$657K$
⟨6-20-8-24-0.75⟩	*cpu*	375	21.8	25.2
	visits	$7119M$	$16M$	$11M$
⟨6-20-10-13-0.95⟩	*cpu*	101	18.2	11.3
	visits	$1754M$	$20M$	$7078K$
⟨6-20-10-14-0.95⟩	*cpu*	312	77.8	50.5
	visits	$5728M$	$82M$	$33M$
⟨6-20-20-5-0.99⟩	*cpu*	0.03	287	0.06
	visits	$194K$	$739M$	$80K$
⟨6-20-20-10-0.99⟩	*cpu*	0.02	257	0.05
	visits	$160K$	$673M$	$64K$

involving 40 Boolean variables and constraints of arity 20 and tightness ≈ 0.02 (exactly $30,000$ allowed tuples). The third series include instances involving 10 variables with domains of size 10 and constraints of arity 6 and tightness 0.1 (exactly $100,000$ allowed tuples). Table 2 shows the results that we have obtained by performing an experimentation in a context roughly similar to [9]: we used a PC Pentium III 600Mhz, running MGAC2001 with *dom* as variable ordering heuristic. Clearly, GAC-*valid+allowed* is the best approach, outperforming GAC-*valid* by about one order of magnitude and GAC-*allowed* by about two orders of magnitude. Surprisingly, it appears that GAC-*allowed* is beaten by GAC-*valid*. In fact, it can be explained by the fact that, during inference or/and search, the number of valid tuples becomes smaller than the number of allowed tuples), which favours GAC-*valid*. Finally, one should note that we obtain with *Abscon* running times of same order than [9].

Finally, we have generated instances involving variables whose domains contain 6, 8, 10 and 20 values, and constraints of arity 6 and of tightness equal to 0.55, 0.75, 0.95 and ≈ 0.99 (exactly 640 allowed tuples). Table 3 gives the results for these new series obtained on a P.IV 2.4GHz when running MGAC2001 with *dom/wdeg*. Not surprisingly, when the tightness is low (i.e. the number of allowed tuples is high), GAC-*allowed* is penalized (here, by one order of magnitude) and when the tightness is high, GAC-*valid* is penalized (here, by more than two orders of magnitude). GAC-*valid+allowed* remains robust at both extremes.

7 Conclusion

In this paper, we have introduced a new algorithm to establish GAC on positive table constraints. Its principle is to avoid considering irrelevant tuples (when a support is looked for) by jumping over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. We have shown that the new schema (GAC-valid+allowed) admits on some instances a behaviour quadratic in the arity of the constraints whereas classical schemas (GAC-valid and GAC-allowed) admit an exponential behaviour.

On the practical side, the results that we have obtained demonstrate the robustness of GAC-valid+allowed. In fact, they are comparable to the ones obtained with a GAC-allowed+valid scheme [9] which allows to skip irrelevant allowed tuples from a reasoning about lower bounds on valid tuples. On the one hand, we believe that our model is simpler, and, importantly, can be easily grafted to any generic GAC algorithm. On the other hand, as the two approaches are different, it should be worthwhile combining them. Indeed, it is possible to conceive a scenario where GAC-allowed+valid would allow skipping tuples visited by GAC-valid+allowed through the use of lower bounds. The reverse is also true since we can consider the illustration given in the introduction while assuming that lower bounds (e.g. *last* pointers of AC2001) are useless (e.g. not initialized). Devising such a hybrid approach is one perspective of this work.

References

1. C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
2. C. Bessière, E.C. Freuder, and J. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.
3. C. Bessière and J. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, 1997.
4. C. Bessière, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
6. P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
7. C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.
8. O. Lhomme. Arc-consistency filtering a1lgorithms for logical combinations of constraints. In *Proceedings of CPAIOR'04*, pages 209–224, 2004.
9. O. Lhomme and J.C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.
10. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
11. A.K. Mackworth. On reading sketch maps. In *Proceedings of IJCAI'77*, pages 598–606, 1977.
12. J.C. Régin. AC-*: a configurable, generic and adaptive arc consistency algorithm. In *Proceedings of CP'05*, pages 505–519, 2005.
13. C. Schulte. Comparing trailing and copying for constraint programming. In *Proceedings of ICLP'99*, pages 275–289, 1999.

Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs

Michele Lombardi and Michela Milano

DEIS, University of Bologna
V.le Risorgimento 2, 40136, Bologna, Italy

Abstract. This paper describes a complete and efficient solution to the stochastic allocation and scheduling for Multi-Processor System-on-Chip (MPSoC). Given a conditional task graph characterizing a target application and a target architecture with alternative memory and computation resources, we compute an allocation and schedule minimizing the expected value of communication cost, being the communication resources one of the major bottlenecks in modern MPSoCs. Our approach is based on the Logic Based Benders decomposition where the stochastic allocation is solved through an Integer Programming solver, while the scheduling problem with conditional activities is faced with Constraint Programming. The two solvers interact through no-goods. The original contributions of the approach appear both in the allocation and in the scheduling part. For the first, we propose an exact analytic formulation of the stochastic objective function based on the task graph analysis, while for the scheduling part we extend the timetable constraint for conditional activities. Experimental results show the effectiveness of the approach.

1 Introduction

The increasing levels of system integration in Multi-Processor Systems on Chips (MPSoCs) emphasize the need for new design flows for efficient mapping of multi-task applications onto hardware platforms. The problem of allocating and scheduling conditional, precedence-constrained tasks on processors in a distributed realtime system is NP-hard. As such, it has been traditionally tackled by means of heuristics, which provide only approximate or near-optimal solutions, see for example [1], [2], [3].

In a typical embedded system design scenario, the platform always runs the same application. Thus, extensive analysis and optimization can be performed at design time. This paper proposes a complete decomposition approach to the allocation and scheduling of a conditional multi-task application on a multi-processor system-on-chip (MPSoCs) [4]. The target application is pre-characterized and abstracted as a Conditional Task Graph (CTG). The task graph is annotated with computation time, amount of communication, storage requirements. However, not all tasks will run on the target platform: in fact, the application contains conditional branches (like if-then-else control structures). Therefore, after an accurate application profiling step, we have a probability distribution on each conditional branch that intuitively gives the probability of choosing that branch during execution.

This paper proposes a non trivial extension to [5] that used Logic Based Benders decomposition [6] for resource assignment and scheduling in MPSoCs. In that paper, however, task graphs did not contain conditional activities. Allocation and scheduling were therefore deterministic. The introduction of stochastic elements complicates the problem.

We propose two main contributions: the first concerns the allocation component. The objective function we consider depends on the allocation variables. Clearly, having conditional tasks, the exact value of the communication cost cannot be computed. Therefore our objective function is the expected value of the communication cost. We propose here to identify an analytic approximation of this value. The approximation is based on the Conditional Task Graph analysis for identifying two data structures: the activation set of a node and the coexistence set of two nodes. The approximation turns out to be exact and polynomial. The second contribution concerns scheduling. We propose an extension of the time-table constraint for cumulative resources, taking into account conditional activities. The propagation is polynomial if the task graph satisfies a condition called *Control Flow Uniqueness* which is quite common in many conditional task graphs for system design. Its deterministic version [7] is available in ILOG Scheduler. The use of the so called *optional activities* (what we call conditional tasks) has been taken into account in [8] for filtering purposes into the precedence graph, originally introduced by Laborie in [9]. To the best of our knowledge, only disjunctive constraints have been defined in presence of conditional activities in [10].

In the system design community, this problem is extremely important and many researchers have worked extensively on it, mainly with incomplete approaches: for instance in [1] a genetic algorithm is devised on the basis of a conditional scheduling table whose (exponential number of) columns represent the combination of conditions in the CTG and whose rows are the starting times of activities that appear in the scenario. The number of columns is indeed reasonable in real applications. The same structure is used in [10], which is the only approach that uses Constraint Programming for modelling the allocation and scheduling problem. Indeed the solving algorithm used is complete only for small task graphs (up to 10 activities).

Besides related literature for similar problems, the Operations Research community has extensively studied stochastic optimization in general. The main approaches are: sampling [11] consisting in approximating the expected value with its average value over a given sample; the *l-shaped* method [12] which faces two phase problems and is based on Benders Decomposition [13]. The master problem is a deterministic problem for computing the first phase decision variables. The subproblem is a stochastic problem that assigns the second phase decision variables minimizing the average value of the objective function. A different method is based on the branch and bound extended for dealing with stochastic variables, [14].

The CP community has recently faced stochastic problems: in [15] stochastic constraint programming is formally introduced and the concept of solution is replaced with the one of *policy*. In the same paper, two algorithms have been proposed based on backtrack search. This work has been extended in [16] where an algorithm based on the concept of scenarios is proposed. In particular, the paper shows how to reduce the number of scenarios, maintaining a good expressiveness.

This paper is organized as follows: in section 2 we present the architecture and the target application considered. In section 3 we present the allocation and scheduling models used. Experimental results are shown in section 4.

2 Problem Description

2.1 The Architecture

Multi Processor Systems on Chips (MPSoCs) are multi core architectures developed on a single chip. They are finding widespread application in embedded systems (such as cellular phones, automotive control engines, etc.). Once deployed in field, these devices always run the same application, in a well-characterized context. It is therefore possible to spend a large amount of time for finding an optimal allocation and scheduling off-line and then deploy it on the field, instead of using on-line, dynamic (sub-optimal) schedulers [17,18].

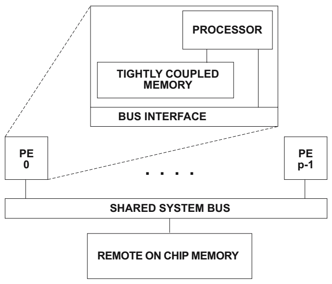


Fig. 1. Single chip multi-processor architecture

The multi-processor system we consider consists of a pre-defined number of distributed Processing Elements (PE) as depicted in Figure 1. All nodes are assumed to be homogeneous and composed by a processing core and by a low-access-cost local scratchpad memory. Data storage onto the scratchpad memory is directly managed by the application, and not automatically in hardware as it is the case for processor caches.

The scratchpad memory is of limited size, therefore data in excess must be stored externally in a remote on-chip memory, accessible via the bus. The bus for state-of-the-art MPSoCs is a shared communication resource, and serialization of bus access requests of the processors (the bus masters) is carried out by a centralized arbitration mechanism. The bus is re-arbitrated on a transaction basis (e.g., after single read/write transfers, or bursts of accesses of pre-defined length), based on several policies (fixed priority, round-robin, latency-driven, etc.). Modeling bus allocation at such a fine granularity would make the problem overly complex, therefore a more abstract additive bus model was devised, explained and validated in [5] where each task can simultaneously access the bus requiring a portion of the overall bandwidth. The communication resource in most cases ends up to be the most congested resource. Communication cost is therefore critical for determining overall system performance, and will be minimized in our task allocation framework.

2.2 The Target Application

The target application to be executed on top of the hardware platform is the input to our algorithm. It is represented as a Conditional Task Graph (CTG). A CTG is a triple $\langle T, A, C\rangle$, where T is the set of nodes modeling generic tasks (e.g. elementary operations, subprograms, ...), A the set of arcs modeling precedence constraints (e.g. due to data communication), and C is a set of conditions, each one associated to an arc, modeling what should be true in order to choose that branch during execution (e.g. the condition of a if-then-else construct). A node with more than one outgoing arc is said to be a *branch* if all arcs are conditional, a *fork* if all arcs are not conditional; mixed nodes are not allowed. A node with more than one ingoing arc is an *or-node* if all arcs are mutually exclusive, it is instead an *and-node* if all arcs are not mutually exclusive; again, mixed nodes are not allowed.

Since the truth or the falsity of conditions is not known in advance, the model is stochastic. In particular, we can associate to each branch a stochastic variable $\mathcal{B}$ with probability space $\langle C, \mathcal{A}, p\rangle$, where C is the set of possible branch exit conditions c, $\mathcal{A}$ the set of events (one for each condition) and p the branch probability distribution (in particular $p(c)$ is the probability that condition c is true).

We can associate to each node and arc an activation function, expressed as a composition of conditions by means of the logical operators $\wedge$ and $\vee$. We call it $f_i(X(\omega))$, where X is the stochastic variable associated to the composite experiment $\mathcal{B}_0 \times \mathcal{B}_1 \times ... \times \mathcal{B}_b$ ($b =$ number of branches) and $\omega \in \mathcal{D}(\mathcal{B}_0) \times \mathcal{D}(\mathcal{B}_1) \times ... \times \mathcal{D}(\mathcal{B}_b)$ (i.e. ω is a scenario).

Computation, storage and communication requirements are annotated onto the graph. In detail, the worst case execution time (WCET) is specified for each node/task and plays a critical role whenever application real time constraints (expressed here in terms of deadlines) are to be met.

Each node/task also has three kinds of associated memory requirements: **Program Data**: storage locations are required for computation data and for processor instructions; **Internal State**; **Communication queues**: the task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different processors. Each of these memory requirement can be allocated either locally in the scratchpad memory or remotely in the on-chip memory.

Finally, the communication to be minimized counts two contributions: one related to single tasks, once computation data and internal state are physically allocated to the scratchpad or remote memory, and obviously depending on the size of such data; the second related to pairs of communicating tasks in the task graph, depending on the amount of data the two tasks should exchange.

3 Model Definition

As already presented and motivated in [5], the problem we are facing can be split into the resource allocation master problem and the scheduling sub-problem.

3.1 Allocation Problem Model

With regards to the platform described in section 2.1, the allocation problem can be stated as the one of assigning processing elements to tasks and storage devices to their memory requirements. First, we state the stochastic allocation model, then we show how this model can be transformed into a deterministic model through the use of existence and co-existence probabilities of tasks. To compute these probabilities, we propose two polynomial time algorithms exploiting the CTG structure.

Stochastic integer linear model. Suppose n is the number of tasks, p the number of processors, and n_a the number of arcs. We introduce for each task and each PE a variable T_{ij} such that $T_{ij} = 1$ iff task i is assigned to processor j. We also define variables M_{ij} such that $M_{ij} = 1$ iff task i allocates its program data locally, $M_{ij} = 0$ otherwise. Similarly we introduce variables S_{ij} for task i internal state requirements and C_{rj} for arc r communication queue. X is the stochastic variable associated to the scenario ω. The allocation model, where the objective function is the minimization of bus traffic expected value, is defined as follows:

$$\begin{aligned}
&\min z = E(\mathit{busTraffic}(M,S,C,X(\omega))) && \\
&s.t. \quad \sum_{j=0}^{p-1} T_{ij} = 1 \qquad \forall i = 0,..,n-1 && (1)\\
&S_{ij} \le T_{ij} \qquad \forall i = 0,..,n-1, j = 0,..,p-1 && (2)\\
&M_{ij} \le T_{ij} \qquad \forall i = 0,..,n-1, j = 0,..,p-1 && (3)\\
&C_{rj} \le T_{ij} \qquad \forall arc_r = (t_i,t_k), r = 0,..,n_a-1, j = 0,..,p-1 && (4)\\
&C_{rj} \le T_{kj} \qquad \forall arc_r = (t_i,t_k), r = 0,..,n_a-1, j = 0,..,p-1 && (5)\\
&\sum_{i=0}^{n-1} [s_i S_{ij} + m_i M_{ij}] + \sum_{r=0}^{n_a-1} c_r C_{rj} \le Cap_j \qquad \forall j = 0,..,p-1 && (6)
\end{aligned}$$

Constraints (1) force each task to be assigned to a single processor. Constraints (2) and (3) state that program data and internal state can be locally allocated on the PE j only if task i runs on it. Constraints (4) and (5) enforce that the communication queue of arc r can be locally allocated only if both the source and the destination tasks run on processor j. Finally, constraints (6) ensure that the sum of locally allocated internal state (s_i), program data (m_i) and communication (c_r) memory cannot exceed the scratchpad device capacity (Cap_j). All tasks have to be considered here, regardless they execute or not at runtime, since a scratchpad memory is, by definition, statically allocated. In addition, some symmetries breaking constraints have been added to the model.

The bus traffic expression is composed by two contributions: one depending on single tasks and one due to the communication between pairs of tasks.

$$\mathit{busTraffic} = \sum_{i=0}^{n-1} \mathit{taskBusTraffic}_i + \sum_{arc_r=(t_i,t_k)} \mathit{commBusTraffic}_r$$

where

$$\mathit{taskBusTraffic}_i = f_i(X(\omega)) \left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij}) \right]$$

$$\mathit{commBusTraffic}_r = f_i(X(\omega)) f_k(X(\omega)) \left[c_r(1 - \sum_{j=0}^{p-1} C_{rj}) \right]$$

In the *taskBusTraffic* expression, if task i executes (thus $f_i(X(\omega)) = 1$), then $(1 - \sum_{j=0}^{p-1} M_{ij})$ is 1 iff the task i program data is remotely allocated. The same holds for the internal state. In the *commBusTraffic* expression we have a contribution if both the source and the destination task execute ($f_i(X(\omega)) = f_k(X(\omega)) = 1$) and the queue is remotely allocated ($1 - \sum_{j=0}^{p-1} C_{rj} = 1$).

Transformation in a deterministic model. In most cases, the minimization of a stochastic functional, such as the expected value, is a very complex operation (even more than exponential), since it often requires to repeatedly solve a deterministic subproblem [12]. The cost of such a procedure is not affordable for hardware design purposes since the deterministic subproblem is by itself NP-hard. **One of the main contributions of this paper is the way to reduce the bus traffic expected value to a deterministic expression.** Since all tasks have to be assigned before running the application, the allocation is a stochastic *one phase* problem: thus, for a given task-PE assignment, the expected value depends only on the stochastic variables. Intuitively, if we properly weight the bus traffic contributions according to task probabilities we should be able to get an analytic expression for the expected value.

Now, since both the expected value operator and the bus traffic expression are linear, the objective function can be decomposed into task related and arc related blocks:

$$E(\mathit{busTraffic}) = \sum_{i=0}^{n-1} E(\mathit{taskBusTraffic}_i) + \sum_{arc_r=(t_i,t_k)} E(\mathit{commBusTraffic}_r)$$

Since for a given allocation the objective function depends only on the stochastic variables, the contributions of decision variables are constants: we call them $KT_i = \left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij})\right]$, and $KC_r = \left[c_r(1 - \sum_{j=0}^{p-1} C_{rj})\right]$. Let us call $p(\omega)$ the probability of scenario ω.

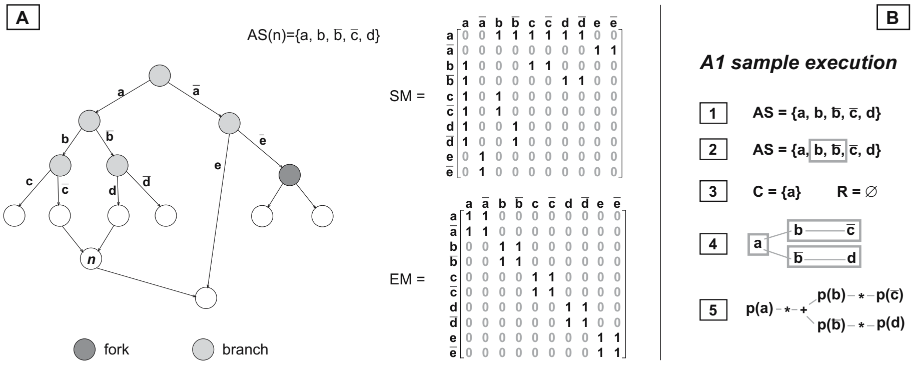
The expected value of each contribution to the objective function is a weighted sum on all scenarios. Weights are scenario probabilities.

$$E(\mathit{taskBusTraffic}_i) = \sum_{\omega \in \Omega} p(\omega) f_i(X(\omega)) KT_i = KT_i \sum_{\omega \in \Omega_i} p(\omega)$$

$$E(\mathit{commBusTraffic}_r) = \sum_{\omega \in \Omega} p(\omega) f_i(X(\omega)) f_k(X(\omega)) KC_r = KC_r \sum_{\omega \in \Omega_i \cap \Omega_k} p(\omega)$$

where r is the index of arc (t_i, t_j) and $\Omega_i = \{\omega \mid \text{task } i \text{ executes}\}$ is the set of all scenarios where task i executes. Now every stochastic dependence is removed and the expected value is reduced to a deterministic expression. Note that $\sum_{\omega \in \Omega_i} p(\omega)$ is simply the existence probability of node/task i while $\sum_{\omega \in \Omega_i \cap \Omega_k} p(\omega)$ is the coexistence probability of nodes i and k. To apply the transformation we need both those probabilities; moreover, to achieve an effective overall complexity reduction, they have to be computed in a reasonable time. We developed two polynomial cost algorithms to compute these probabilities.

Existence and co-existence probabilities. All developed algorithms are based on three types of data structures derived from the CTG. In Figure 2A we show an example of a CTG on the left and the related data structures:



$$SM = \begin{array}{c|cccccccccc} & a & \bar{a} & b & \bar{b} & c & \bar{c} & d & \bar{d} & e & \bar{e} \\ \hline a & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \bar{a} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ b & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \bar{b} & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ c & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{c} & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{d} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{e} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$EM = \begin{array}{c|cccccccccc} & a & \bar{a} & b & \bar{b} & c & \bar{c} & d & \bar{d} & e & \bar{e} \\ \hline a & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{a} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ b & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{b} & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \bar{c} & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ \bar{d} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \bar{e} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Fig. 2. A: An example of the three data structures; **B:** a sample execution of A1

- the *activation set* of a node n ($AS(n)$). It is computed by traversing all paths from the starting node to n and collecting the conditions on the paths.
- a binary $c \times c$ *exclusion matrix (EM)* where c is the number of conditions. $EM_{ij} = 1$ iff c_i and c_j are mutually exclusive (i.e. they originate at the same branch)
- a binary $c \times c$ *sequence matrix (SM)*. $SM_{ij} = 1$ iff c_i and c_j are both needed to activate some node in the CTG.

All these data structures can be extracted from the graph in polynomial time. Once they are available, we can determine the existence probability of a node or an arc using algorithm A1, which has $O(c^3)$ complexity representing sets as bit vectors; in the algorithm the notation SM_i stands for the set of conditions "sequenced" with a given one ($SM_i = \{c_j | SM_{ij} = 1\}$); the same holds for EM_i.

algorithm: Activation set probability (A1) – probability of a node or an arc

```
 1. let $S$ be the input set for this iteration; initially $S = AS(n)$
 2. find a condition $c_h \in S$ such that $(EM_h \setminus \{c_h\}) \cap S \neq \emptyset$
 3. if such a condition doesn't exist return $p = \prod_{c \in S} p(c)$
 4. otherwise, set $B = EM_h \cap S$
 5. compute set $C = S \cap \bigcap_{c_i \in B} SM_i$
 6. compute set $R = \bigcap_{c_i \in B} (S \setminus SM_i)$
 7. set $p = 0$
 8. for each condition $c_i \in B$:
     8.1. set $p = p + A1((S \cap SM_i) \setminus (C \cup R))$
 9. set $p = p * A1(C) * A1(R)$
10. return $p$
```

2: find a group of exclusive conditions in S and choose one of them
4: get all conditions in S originating from the chosen branch ("Branch" set)
5: get conditions "in sequence" with all branch outcomes ("Common" set)
6: get conditions not "in sequence" with any of the branch outcomes ("Rest" set)
8.1: for each branch outcome, get the conditions "in sequence" and compute probability
9: multiply computed value for the probability of "Common" and "Rest" set

end

Algorithm A1 works recursively partitioning the activation set of the target node: let us follow the algorithm on the example in figure 2B. We have to compute the probability of node n, while activation set is `AS(n) =` {`a,b, not b, not c, d`}. The algorithm looks for a group of mutually exclusive conditions (the B set), see `b` and `not` b in AS(n). If there is no such condition the probability of the activation set S is the product of the probabilities of its elements (step 3). If there is are at least two exclusive conditions, the algorithm then builds a "common" (C) and a "rest" (R) set: the first contains conditions c_j such that $SM_{ij} = 1 \quad \forall c_i \in B$, the second conditions c_h such that $SM_{ih} = 0 \quad \forall c_i \in B$. In the example C = $\{$a$\}$ and R = $\emptyset$. Finally A1 builds for each found branch condition a set containing the sequenced conditions ($S \cap SM_i$ at step 8.1), and chains b and not c and not b and d in figure 2. A1 is then recursively called on all these sets. The probabilities of sets corresponding to mutually exclusive conditions are summed (step 8.1), the ones of C and R are multiplied (step 9).

algorithm: Coexistence set determination (A2)

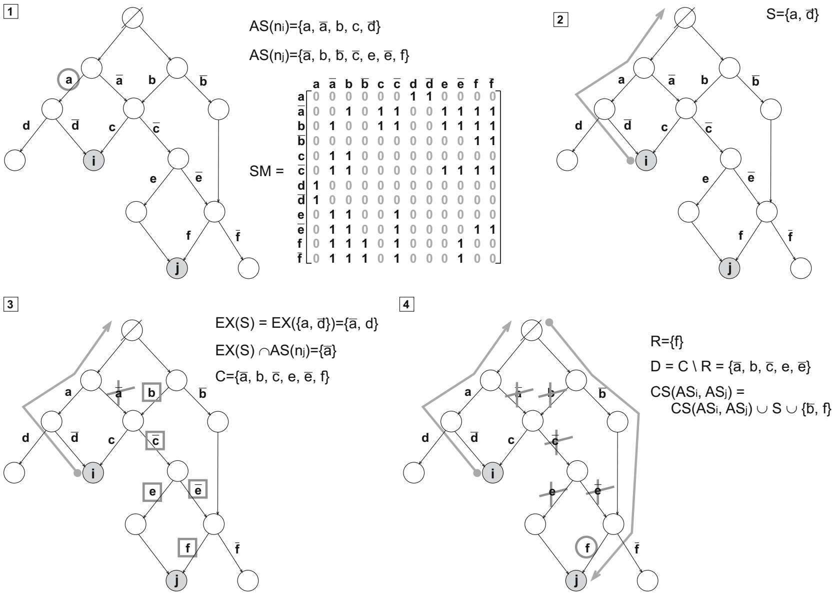
```
1. if $AS_i = \emptyset$ then $CS = AS_j$; the same if $AS_j = \emptyset$
2. otherwise, if there are still not processed conditions in $AS_i$,
   let $c_h$ be the first of them:
    2.1. compute set $S = AS_i \cap SM_h$
    2.2. compute the exclusion set $EX(S)$
    2.3. compute set:
         $C = AS_j \cap \bigcup_{c_k \in AS_j \cap EX(S)} SM_k$
    2.4. compute set:
         $R = AS_j \cap \bigcup_{c_k \in AS_j \setminus C} SM_k$
    2.5. set $D = C \setminus R$ (conditions to delete)
    2.6. if $AS_j$ is not a subset of $D$:
        2.6.1. set $CS(AS_i, AS_j) =$
               $CS(AS_i, AS_j) \cup S \cup (AS_j \setminus D)$
end
```

1: *If one of the input set is $\emptyset$, then the coexistence set is simply the other activation set*
2.1: *Select conditions "in sequence" with the chosen one; they individuate a group of backward paths*
2.3: *Find in AS_j conditions excluded by the selected forward paths ($AS_j \cap EX(S)$); all conditions "in sequence" with those are canditates to be deleted (i.e. excluded from the feasible forward paths)*
2.4: *Consider not candidate conditions in AS_j: all conditions "in sequence" with them must not be deleted*
2.6: *If AS_j has not been completely deleted, add to CS all conditions in backward and forward paths*

Given a set of nodes, we can determine a kind of common activation set (*coexistence set (CS)*) using algorithm A2, whose inputs are two AS (AS_i, AS_j) and whose complexity is again $O(c^3)$. The notation $EX(S)$ stands for the exclusion set, i.e. the set of conditions surely excluded by those in S; it can be computed in $O(c^2)$.

Suppose we have the activation sets of two nodes n_i and n_j: then A2 works trying to find all paths from n_i to a source (backward paths) and from the source to n_j (forward paths). The algorithm starts building a group of backward paths; it does it by choosing a condition (for instance condition a in $\boxed{1}$ figure 3) and finding all other conditions sequenced with it (set S in $\boxed{2}$ figure 3).

Then the algorithm finds the exclusion set ($EX(S)$) of set S and intersects it with $AS(n_j)$. In $\boxed{3}$ figure 3 the only condition in the intersection is not a (crossed arc): conditions in the intersection and those sequenced with them are called "candidates conditions" (set C in $\boxed{3}$ figure 3). These conditions will be removed from $AS(n_j)$, unless they are sequenced with one or more non-candidate conditions, i.e., they belong to the set R (for instance condition f is in sequence with not b and is not removed from $AS(n_j)$ in $\boxed{4}$, figure 3). The conditions not removed from $AS(n_j)$ identify a set of forward paths we are interested in. The algorithm goes on until all conditions in $AS(n_i)$ are processed. If there is no path from n_i to n_j (i.e. the coexistence set is empty) the two nodes are mutually exclusive.



$AS(n_i)=\{a, \bar{a}, b, c, \bar{d}\}$

$AS(n_j)=\{\bar{a}, b, \bar{b}, \bar{c}, e, \bar{e}, f\}$

$$SM = \begin{array}{c|cccccccccccc} & a & \bar{a} & b & \bar{b} & c & \bar{c} & d & \bar{d} & e & \bar{e} & f & \bar{f} \\ \hline a & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \bar{a} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ b & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ \bar{b} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ c & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{c} & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ d & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{d} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{e} & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ f & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \bar{f} & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

$EX(S) = EX(\{a, \bar{d}\})=\{\bar{a}, d\}$

$EX(S) \cap AS(n_j)=\{\bar{a}\}$

$C=\{\bar{a}, b, \bar{c}, e, \bar{e}, f\}$

$D = C \setminus R = \{\bar{a}, b, \bar{c}, e, \bar{e}\}$

$CS(AS_i, AS_j) =$
$CS(AS_i, AS_j) \cup S \cup \{\bar{b}, f\}$

Fig. 3. Coexistence set computation

The probability of a coexistence set can be computed once again by means of A1: thus, with A1 and A2 we are able to compute the existence probability of a single node and the coexistence probability of a group of nodes or arcs. Since the algorithms complexities are polynomial, the reduction of the bus traffic to a deterministic expression can be done in polynomial time.

3.2 Scheduling Model

The scheduling subproblem has been solved by means of Constraint Programming. Since the objective function depends only on the allocation of tasks and memory requirements, scheduling is just a feasibility problem. Therefore we decided to provide a unique worst case schedule, forcing each task to execute after all its predecessors in any scenario. Tasks using the same resources can overlap if they are on alternative paths (under two mutually exclusive conditions).

Tasks have a five phases behavior: they read all communication queues (INPUT), eventually read their internal state (RS), execute (EXEC), write their states (WS) and finally write all the communications queues (OUTPUT). Each task is modeled as a group of not breakable activities; the adopted schema and precedence relations vary with the type of the corresponding node (or/and, branch/fork). For the lack of space we do not explain these relations here, but they can be found in [21].

Each activity duration is an input parameter and can vary depending on the allocation of internal state and program data. The processing elements are unary resources: we modeled them defining a simple disjunctive constraint proposed in [10].

The bus, as in [5], is modeled as a cumulative resource, according with the so called "additive model", which allows an error less than 10% until bandwidth usage is under

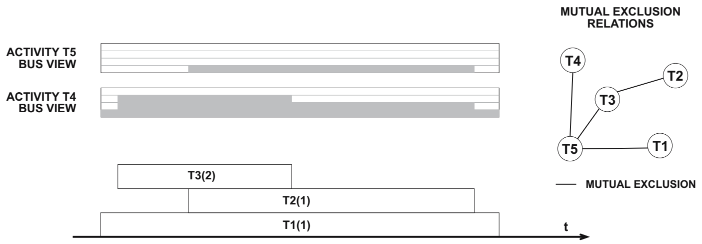


Fig. 4. Activity bus view

60% of the real capacity. Computing the bus usage in presence of alternative activities is not trivial, since the bus usage varies in a not linear way and every activity can have its own bus view (see fig 4).

Suppose for instance we have the five tasks of figure 4; activities **T1**, **T2**, **T3** have already been scheduled: the bus usage for each of them is reported between round brackets, while all the mutual exclusion relations are showed on the right. Let's consider activity **T4**, which is not mutually exclusive with any of the scheduled tasks. As long as only **T1** is present, the bus usage is 1. It becomes 1 + 2 = 3 when also activity **T3** starts, but when both **T1**, **T2** and **T3** execute the bus usage remains 3, since **T2** and **T3** are alternative. Thus the bus usage at a given time is always the maximum among all the combinations of not alternative running tasks. Furthermore, let's consider activity **T5**: since it is mutually exclusive with all tasks but **T2**, it only sees the bus usage due to that task. Therefore the bus view at a given time depends on the activity we are considering.

We modeled the bus creating a new global timetable constraint for cumulative resources and conditional tasks in the not preemptive case. The global constraint keeps a list of all known entry and exit points of activities: given an activity A, if $lst(A) \leq eet(A)$ then the entry point of A is $lst(A)$ and $eet(A)$ is its exit point (where lst stands for latest start time and so on).

Let A be the target activity: A3 scans the interval $[est(A), finish)$ checking the bus usage at all entry points (as long as $good = true$). If it finds an entry point with not enough bandwidth left it starts to scan all exit points ($good = false$) in order to determine a new possible starting time for activity A. If such an instant is found its value is stored ($lastGoodTime$) and the finish line is updated (step 4.2.2.2), then A3 restarts to scan other entry points, and so on. When the finish line is reached the algorithm updates $est(A)$ or fails. A3 has $O(a(c + b))$ complexity, where a is the number of activities, b the one of branches, c the number of conditions. The algorithm can be easily extended to update also $let(A)$: we tried to do it, but the added filtering is not enough to justify the increased propagation time.

A3 is able to compute the bandwidth usage seen from each activity in $O(b + c)$ by taking advantage of a particular data structure we introduced, named Branch Fork Graph (BFG). For lack of space we suggest the reader look at [21]. The BFG makes it possible to compute bus usage in a very efficient way, by making direct use of the graph structure: if we only took into account the exclusion relations it would be an NP-hard problem.

To have a polynomial time algorithm however the graph should satisfy a particular condition (called "Control Flow Uniqueness") which states that each "and" node must have a main ingoing arc, whose activation implies the activation of the other ingoing arcs. This is not a very restrictive condition since it is satisfied by every graph resulting from the natural parsing of programs written in a language such C++ or Java.

algorithm: Propagation of the cumulative resource constraint with alternative activities (A3)

```
1. $time = est(a)$, $finish = eet(a)$
2. $latestGoodTime = time$
3. $good = true$
4. While $\neg$ $[(good = false \wedge time > lst(a)) \vee (good = true \wedge time >= finish)]$:
  4.1. if $busreq(a) + usedBandwith > busBandwidth$:
    4.1.1. $time =$ next exit point
    4.1.2. $good = false$
  4.2. else:
    4.2.1. $time =$ next entry point
    4.2.2. if $good = false$:
      4.2.2.1. $lastGoodTime = time$
      4.2.2.2. $finish = max(finish, time + mindur(a))$
      4.2.2.3. $good = true$
5. if $good = true$: $est(a) = lastGoodTime$
6. else: $fail$
end
```

3.3 Benders Cuts and Subproblem Relaxation

Each time the master problem solution is not feasible for the scheduling subproblem a cut is generated which forbids that solution. Moreover, all solutions obtained by permutation of PEs are forbidden, too.

Unfortunately, this kind of cut, although sufficient, is weak; this is why we decided to introduce another cut type, generated as follows: (1) solve to feasibility a single machine scheduling model with only one PE and tasks running on it; (2) if there is no solution the tasks considered cannot be allocated to any other PE.

The cut is very effective, but we need to solve an NP-hard problem to generate it; however, in practice, the problem can be quickly solved.

With the objective to limit iteration number (which strongly influences the solution method efficiency) we also inserted in the master problem a relaxation of the subproblem. This forbids the allocator to store in a single processor a set of non mutually exclusive tasks whose duration exceeds the time limit, and to assign memory devices in such a way that the total length of a track is greater than the deadline.

4 Experimental Results

We implemented all exposed algorithms in C++, using the state of the art solvers ILOG Cplex 9.0 (for ILP) and ILOG Solver 6.0 and Scheduler 6.0 (for CP). We tested all instances on a Pentium IV pc with 512MB RAM. The time limit for the solution process was 30 minutes.

We tested the method on two set of instances: the first ones are characterized by means of a synthetic benchmark; peculiar input data of this problem (such as the branch

Table 1. Results of the tests on the first group of instances (slightly structured)

acts	PEs	inst.	inf.	time	init	master	sub	nogood	it	A/S/I
10-12	2	6	0	0.0337	0.0208	0.0075	0.0027	0.0027	1.1667	0/0/0
13-15	2	8	1	0.5251	0.1600	0.0076	0.0040	0.0020	1.1250	0/0/0
16-18	2-3	12	0	0.1091	0.0922	0.0089	0.0067	0.0013	1.0833	0/0/0
19-21	2-3	14	1	0.1216	0.0791	0.0279	0.0079	0.0046	1.2143	0/0/0
22-24	2-3	23	4	0.2336	0.1520	0.0259	0.0061	0.0081	1.1739	0/0/0
25-27	2-3	16	3	1.7849	0.0319	1.7285	0.0108	0.0088	1.3125	0/0/0
28-30	2-3	13	2	0.3331	0.0284	0.0770	0.1900	0.0338	1.6667	0/1/0
31-33	3-4	4	2	0.3008	0.2303	0.0510	0.0040	0.0000	1.0000	0/0/0
34-36	3-4	13	4	0.6840	0.0204	0.4245	0.0132	0.0108	1.2308	0/0/0
37-39	3-4	7	0	1.5670	0.0399	1.2010	0.1384	0.1877	4.4286	0/0/0
40-42	3-4	6	3	2.9162	0.0182	0.5857	2.2267	0.0390	1.6667	0/0/0
43-45	3-4	6	1	5.3670	0.2757	4.8200	0.0630	0.2005	4.1667	0/0/0
46-48	4-5	11	0	3.2719	0.0508	0.6913	2.4616	0.0683	2.0000	1/2/0
49-51	4-5	11	1	1.9950	0.1840	1.7900	0.0071	0.0087	1.1111	1/1/0
52-54	5-6	6	0	8.0000	1.3398	1.5743	4.8788	0.2073	2.7500	1/1/0
55-67	6	8	0	2.2810	0.8333	1.4377	0.0100	0.0000	1.0000	1/4/0

probabilities) were estimated via a profiling step. Instances of this first group are only slightly structured, i.e. they have very short tracks and quite often contain singleton nodes: therefore we decided to generate a second group of instances, completely structured (one head, one tail, long tracks)[1].

The results of the tests on the first group are summarized in table 1. Instances are grouped according to the number of activities (acts); beside this, the table reports also the number of processing elements (PEs), the number of instances in the group (inst.), the instances which were proven to be infeasible (inf.), the mean overall time (in seconds), the mean time to analyze the graph (init), to solve the master and the subproblem, to generate the no-good cuts and the mean number of iterations (it). The solution times are of the same order of the deterministic case (scheduling of Task Graphs), which is a very good result, since we are working on conditional task graphs and thus dealing with a stochastic problem.

For a limited number of instances the overall solving time was exceptionally high: the last column in the table shows the number of instances for which this happened, mainly due to the master problem (A), the scheduling problem (S) or the number of iterations (I). The solution time of these instances was not counted in the mean; in general it was greater than than thirty minutes.

Although this extremely high solution time occurs with increasing frequency as the number of activities grows, it seems it is not completely determinated by that parameter: sometimes even a very small change of the deadline or of some branch probability makes the computation time explode.

We guess that in some cases, when the scheduler is the cause of inefficiency, this happens because of search heuristic: for some input graph topologies and parameter configurations the heuristic does not make the right choices and the solution time

[1] All instances are available at http://www-lia.deis.unibo.it/Staff/MichelaMilano/tests.zip

Table 2. Result of the tests on the second group of instances (completely structured)

acts	PEs	inst.	inf.	time	init	master	sub	nogood	it	A/S/I
20-29	2	7	2	0.5227	0.0200	0.0134	0.0090	0.0021	8.8571	0/0/0
30-39	2-3	6	0	1.7625	0.0283	1.2655	0.2057	0.2630	5.8333	0/0/0
40-49	3	3	0	0.4380	0.0313	0.3493	0.0573	0.0000	1.0000	0/0/0
50-59	3-4	7	0	1.1403	0.0310	0.6070	0.2708	0.2315	3.6667	0/0/1
60-69	4-5	4	0	10.1598	0.0385	6.8718	1.2798	1.9698	18.0000	0/0/0
70-79	4-5	4	0	88.9650	0.0428	88.6645	0.2578	0.0000	1.0000	0/0/0
80-90	4-6	7	0	202.4655	0.0755	184.0177	6.5008	11.8715	28.6667	0/0/1

Table 3. Number of iterations without and with scheduling relaxation based cuts

		mean time to gen. a cut			
basic case:					0.0074
with relaxation based cuts (RBC):					0.0499
	number of iterations		**excution time**		
deadline	**basic case**	**with RBC**	**basic case**	**with RBC**	**result**
8557573	2	3	1.18	0.609	opt. found
625918	1	1	0.771	0.765	opt. found
590846	1	1	0.562	0.592	opt. found
473108	19	6	6.169	1.186	opt. found
464512	190	14	201.124	9.032	opt. found
454268	195	24	331.449	10.189	opt. found
444444	78	15	60.747	6.144	opt. found
433330	9	4	4.396	1.657	opt. found
430835	5	3	3.347	1.046	opt. found
430490	5	3	3.896	1.703	opt. found
427251	3	2	2.153	0.188	inf.

dramatically grows. Perhaps this could be avoided by randomizing the solution method and by using restart strategies [19].

The results of the second group of instances (completely structured) are reported in table 2. In this case the higher number of arcs (and thus of precedence constraints) reduces the time windows and makes the scheduling problem much more stable: no instance solution time exploded due to the scheduling problem. On the other hand the increased number of arcs makes the allocation more complex and the scheduling problem approximation less strict, thus increasing the number of iterations and their duration. In two cases we go beyond the time limit.

We also ran a set of tests to verify the effectiveness of the cuts we proposed in section 3.3 with respect to the basic cuts removing only the solution just found: table 3 reports results for a 34 activities instance repeatedly solved with a decreasing deadline values, until the problem becomes infeasible. The iteration number greatly reduces. Also, despite the mean time to generate a cut grows by a factor of ten, the overall solving time per instance is definitely advantageous with the tighter cuts.

Finally, to estimate the quality of the chosen objective function (bus traffic expected value), we tested it against an easier, heuristic technique of deterministic reduction.

Table 4. Comparison with heuristic deterministic reduction

			quality improvement		
instance	**activities**	**scenarios**	**mean**	**min**	**max**
1	53	10	4.72%	-0.88%	13.08%
2	57	10	2.59%	-0.11%	8.82%
3	54	24	12.65%	-0.72%	39.22%

The chosen heuristic simply optimizes bus traffic for the scenario when each branch is assigned the most likely outcome; despite its simplicity, this is a particularly relevant technique, since it is widely used in modern compilers ([20]).

We ran tests on three instances: we solved them with our method and the heuristic one (obtaining two different allocations) and we computed the bus traffic for each scenario with both the allocations. The results are shown in table 4, where for each instance are reported the mean, minimum and maximum quality improvement against the heuristic method. Note that on the average our method always improves the heuristic solution; moreover, our solution seems to be never much worse then the other, while it is often considerably better.

5 Conclusion and Future Works

We have proposed a stochastic method for planning and scheduling in the stochastic case. The method proposed has two main contributions: the first is a polynomial transformation of a stochastic problem into a deterministic one based on the conditional task graph analysis. Second, the implementation of two constraints for unary and cumulative resources in presence of conditional activities. We believe the results obtained are extremely encouraging. In fact, computation times are comparable with the deterministic version of the same instances. We still have much work to do: first we have to solve the extremely hard instances possibly through randomization; second we have to take into account other aspects where stochasticity could come into play, like task duration which could not be known in advance. Third, we have to validate these results on a real simulation platform to have some feedback on the model.

Acknowledgement. This work has been partially supported by MIUR under the COFIN 2005 project *Mapping di applicazioni multi-task basate su Programmazione a vincoli e intera.*

References

1. Wu, D., Al-Hashimi, B., Eles, P.: Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. In: Computers and Digital Techniques, IEE Proceedings. Volume 150 (5). (2003) 262–273
2. Eles, P.P.P., Peng, Z.: Bus access optimization for distributed embedded systems based on schedulability analysis. In: International Conference on Design and Automation in Europe, DATE2000, IEEE Computer Sociery (2000)

3. Shin, D., Kim, J.: Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In: International Symposium on Low Power Electronics and Design (ISLPED), ACM (2003)
4. Wolf, W.: The future of multiprocessor systems-on-chips. In: In Procs. of the 41st Design and Automation Conference - DAC 2004, San Diego, CA, USA, ACM (2004) 681–685
5. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSOCs via decomposition and no-good generation. In: Proc. of the Int.l Conference in Principles and Practice of Constraint Programming. (2005)
6. Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming **96** (2003) 33–60
7. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling. Kluwer Academic Publisher (2003)
8. Vilim, P., Bartak, R., Cepek, O.: Extension of o(n.log n) filtering algorithms for the unary resource constraint to optional activities. Constraints **10** (2005) 403–425
9. Laborie, P.: Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. Journal of Artificial Intelligence **143** (2003) 151–188
10. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems **8** (2003)
11. Ahmed, S., Shapiro, A.: The sample average approximation method for stochastic programs with integer recourse. In: Optimization on line. (2002)
12. Laporte, G., Louveaux, F.V.: The integer l-shaped method for stochastic integer programs with complete recourse. Operations Research Letters **13** (1993)
13. Benders, J.F.: Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik **4** (1962) 238–252
14. Norkin, V.I., Pflug, G., Ruszczynski, A.: A branch and bound method for stochastic global optimization. Mathematical Programming **83** (1998)
15. Walsh, T.: Stochastic constraint programming. In: Proc. of the European Conference on Artificial Intelligence, ECAI. (2002)
16. Tarim, A., Manandhar, S., Walsh, T.: Stochastic constraint programming: A scenario-based approach. Constraints **11** (2006) 53–80
17. Culler, D.A., Singh, J.P.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann (1999)
18. Compton, K., Hauck, S.: Reconfigurable computing: A survey of systems and software. ACM Computing Surveys **34** (1999) 171–210
19. Gomes, C.P., Selman, B., McAloon, K., Tretkoff, C.: Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In: AIPS. (1998) 208–213
20. Faraboschi, P.; Fisher, J.A.; Young, C.: Instruction scheduling for instruction level parallel processors. In: Proceedings of the IEEE , vol.89, no.11 pp.1638-1659, Nov 2001
21. M. Lombardi, M. Milano Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs. Technical Report 77 LIA-003-06.

Boosting Open CSPs[⋆]

Santiago Macho González[1], Carlos Ansótegui[2], and Pedro Meseguer[1]

[1] Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, E-08193 Bellaterra, Catalonia, Spain
smacho@iiia.csic.es, pedro@iiia.csic.es
[2] Universitat de Lleida (UdL)
Jaume II, 69, 25001 Lleida, Spain
carlos@diei.udl.es

Abstract. In previous work, a new approach called Open CSP (OCSP) was defined as a way of integrate information gathering and problem solving. Instead of collecting all variable values before CSP resolution starts, OCSP asks for values dynamically as required by the solving process, starting from possibly empty domains. This strategy permits to handle unbounded domains keeping completeness. However, current OCSP algorithms show a poor performance. For instance, the FO-Search algorithm uses a Backtracking and needs to solve the new problem from scratch every time a new value is acquired. In this paper we improve the original algorithm for the OCSP model. Our contribution is two-fold: we incorporate local consistency and we avoid solving subproblems already explored in previous steps. Moreover, these two contributions guarantee the completeness of the algorithm and they do not increase the number of values needed for finding a solution. We provide experimental results than confirm a significant speed-up on the original approach.

1 Problem-Solving in Open Environments

The increasing desire to automate problem-solving for scenarios that are distributed over a network of agents can be addressed with existing tools, first collecting all the options and constraints in an information gathering phase, and second solving the resulting problem using a centralized constraint solver. This conventional approach of collecting values from all servers and then running a CSP solver has been implemented in many distributed information systems [2,1,3]. However, it is very inefficient because it asks for more values than the strictly needed to find a solution, and it does not work with unbounded number of values.

In the real world, choices (values) and constraints are collected from different sources. With the increasing use of the Internet, those classical CSP problems could be defined in an *open-world* environment. Imagine you want to configure a

⋆ Research partially supported by TIN2004-07933-C03-03, TIN2005-09312-C03-01 and TIC2003-00950 funded by the *Ministerio de Educación y Ciencia*.

PC using web data sources. Querying all the possible PC parts in all data sources on the web is just not feasible. We are interested in querying the minimum amount of information until finding a solution. Since the classical CSP approach (querying all values before search starts) is not applicable here, the new *Open CSP* approach [6] was proposed. Solving an *Open CSP* implies obtaining values for the variables, one by one. If the collected information does not allow to solve the problem, new values are requested. The process stops when a solution is found.

Although there are several models which in principle are suitable for open environments such as the *Iterative CSP* model [14] and the *Dynamic CSP* model [15], our model deals with a different problem. These approaches assume bounded domains while *Open CSP* assumes unbounded domains (domains with a possibly unlimited number of values). The ability to handle unbounded domains poses an interesting challenge when designing algorithms. For this reason, we do not include experiments comparing these approaches. See [5] for a detailed relation among these models.

Several algorithms for solving *Open CSPs* were proposed in [6]. These algorithms have a poor performance due to the lack of local consistency and that to the fact they solve from scratch a problem every time new values are acquired. Local consistency allows CSP algorithms to be very powerful, thus in this paper we present a new algorithm called FCO-Search that uses local consistency. We also show how the Factoring Out Failure strategy [4] can be used to avoid solving a problem from scratch.

This paper is organized as follows: firstly a brief description of the *Open CSP* model is given, followed by an explanation of the FO-Search algorithm. In the next sections we discuss how to improve this algorithm by incorporating local consistency and by avoiding to solve from scratch every instance of the OCSP. Then, we describe the FCO-Search algorithm which incorporates the mentioned improvements and finally the benefits of our approach are shown.

2 Open Constraint Satisfaction Problems

In Figure 1 we show the important elements of an open setting. The problem-solving process is modeled abstractly as the solution of a CSP. The choices that make up domains and permitted tuples of the CSP are distributed throughout an unbounded network of information servers $IS_1, IS_2, ...$, and accessed through a mediator [7]. For the purpose of this paper, we assume that this technology allows the CSP solver to obtain additional domain values:

– Using the $more(x_i, \ldots, (x_i, x_j), \ldots)$ message, it can request the mediator to gather more values for these variables. In this paper we assume that this method returns just one new value every time it is called.
– Using $options(x_i, \ldots,)$ and $options((x_i, x_j), \ldots)$ messages, the mediator informs the CSP solver of additional domain values or constraint tuples found in the network.

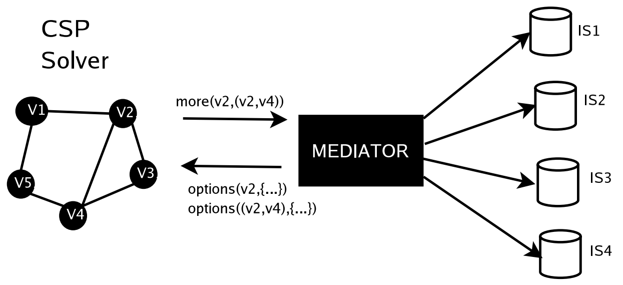


Fig. 1. Elements of an open constraint satisfaction problem

– When there are no more values to be found, the mediator returns *nomore* $(x_i, \ldots)$.

Formally, an *Open CSP*(OCSP) [6] is a possibly infinite sequence $\langle$CSP(0), CSP(1), $\ldots\rangle$ of CSP instances. An instance CSP(i) is the tuple $\langle X, D(i), C(i)\rangle$ where,

– $X = \{x_1, x_2, ..., x_n\}$ is a set of n variables.
– $D(i) = \{D_1(i), D_2(i), ..., D_n(i)\}$ is the set of domains for CSP(i) where variable v_k takes values in $D_k(i)$. Initially domains are empty, $D_k(0) = \emptyset$, and they grow monotonically with i, $D_k(i) \subseteq D_k(i+1)$ for all k.
– $C(i) = \{c_1(i), c_2(i), \ldots, c_r(i)\}$ is a set of r constraints. A constraint $c(i)$ involves a sequence of variables $var(c(i)) = \langle v_p, \ldots, v_q\rangle$ denominated its scope. The extension of $c(i)$ is the relation $rel(c(i))$ defined on $var(c(i))$, formed by the permitted value tuples on the constraint scope. Initially, relations are empty, $rel(c_k(0)) = \emptyset$, and they grow monotonically, $rel(c_k(i)) \subseteq rel(c_k(i+1))$ for all k.

A *solution* is a set of value assignments involving all variables such that for some i, each value belongs to the corresponding domain in $D(i)$ and all value combinations are allowed by the constraints $C(i)$ of $CSP(i)$. Solving an OCSP requires an integration of search and information gathering. It starts from a state where all domains are empty, and the first action is to find values that fill the domains and allow the search to start. As long as the available information does not include enough values to make the CSP solvable, the problem solver initiates further information gathering requests to obtain additional values. The process stops as soon as a solution is found. Thus, with this OCSP model, we are solving a satisfiability problem, but also we are interested in optimizing the number of queries needed to find a solution.

The definition of an OCSP assumes that all constraints are binary. For experimentation, we assume that only variable domains change over time. We made these assumptions for the simplicity of the algorithms. They are not strong restrictions because using the **hidden variable encoding** method explained in [12,13], any problem whose constraints are non binary or that are incrementally discovered could be turned into a variable which has as values the tuples allowed

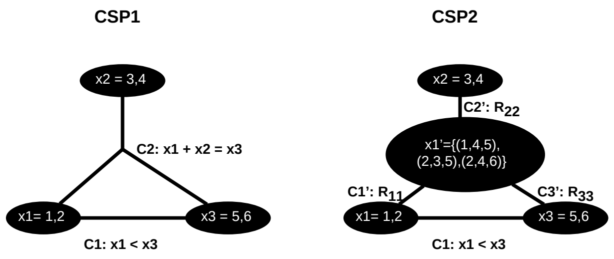


Fig. 2. Hidden variable encoding of a non-binary CSP

by the constraints. These new variables are linked to variables involved in the original problem by new binary constraints that enforce equality between the variable values and the corresponding elements of the tuple. Figure 2 shows an example of the hidden variable encoding for a non binary CSP.

3 The FO-Search Algorithm

The idea behind the FO-Search algorithm is that new values have to be gathered only when the current instance $CSP(i)$ has no solution. In that case, it usually contains a subproblem that already has no solution, and $CSP(i)$ could be made solvable only by creating a solution to that subproblem. Information gathering thus should focus on the variables of this subproblem.

An *Unsolvable Subproblem* of size k is a set of variables $S = \{x_{s1}, x_{s2}, \ldots, x_{sk}\}$ such that there is no value assignment $x_{s1} \in D_{s1}, \ldots, x_{sk} \in D_{sk}$ (where D_{si} represents a domain) that satisfies all constraints between these variables. If any subset $S' \subset S$ is solvable, we call this set of variables *Minimal Unsolvable Subproblem*.

Note that any strategy that does not assure the selection of a variable that belongs to a *Minimal Unsolvable Subproblem* may lead us to an incomplete algorithm. Actually, this is the key point of working with unbounded domains. Think for example about an strategy of selecting the most constrained variable. This variable is not forced to belong to a *Minimal Unsolvable Subproblem*, thus, adding new values to this variable may loop infinitely without solving the unsolvable subproblems that made inconsistent the instance. Although it seems difficult to choose a correct variable, the following result provides a method to identify a variable that belongs to a *Minimal Unsolvable Subproblem*.

Proposition 1. Let a CSP be explored by a failed backtrack search algorithm (BT) with static variable ordering $(x_1, ..., x_n)$, and let x_k be the deepest node reached in the search with inconsistency detected at x_k. Then x_k, called the *failed variable*, is part of every unsolvable subproblem of the CSP involving variables in the set $\{x_1..x_k\}$.

Using this result (proved in [6]), a failed CSP search allows us to identify the failed variable, for which an additional value should be collected. When there are no additional values for this variable, the mediator returns a `nomore` message, and other variables are then considered.

The resulting algorithm **FO-search** (failure-driven open search) is shown in Algorithm 1. It makes the assumption that variables are ordered by the

Algorithm 1. *The* **FO-search** *algorithm*

```
1: procedure FO-search(X(i),D(i),C(i))
2: $i \leftarrow 1, k \leftarrow 1$
3: repeat {backtrack search}
4:    if $exhausted(d_i)$ then {backtrack}
5:       $reset - values(d_i)$, $i \leftarrow i - 1$
6:    else
7:       $k \leftarrow max(k, i), x_i \leftarrow nextvalue(d_i)$
8:       if $consistent(\{x_1..x_i\})$ then {extend assignment}
9:          $i \leftarrow i + 1$
10:      end if
11:      if $i > n$ then
12:         return $\{x_1, ..., x_n\}$ as a solution
13:      end if
14:   end if
15: until $i = 0$
16: if $e_k = CLOSED$ then
17:    if $(\forall i \in 1..k - 1) e_k = CLOSED$ then
18:       return failure
19:    end if
20: else
21:    $nv \leftarrow$ more$(x_k)$
22:    if $nv =$ nomore$(x_k)$ then
23:       $e_k \leftarrow CLOSED$
24:    end if
25:    $d_k \leftarrow nv \cup d_k$
26: end if
27: reorder variables so that $x_k$ becomes $x_1$ (relative order of others remains the same)
28: FO-search(X(i),D(i),C(i)) {search again}
```

index i, and uses the array $E = \{e_1, .., e_n\}$ to indicate whether the domain for the corresponding variable is completely known (CLOSED). The algorithm assumes that no constraint propagation is used, although the chronological backtracking can be replaced with backjumping techniques (jumping directly to the last constraint violation) to make it more efficient.

In [6] it is shown that if the current instance $CSP(i)$ contains a minimal unsolvable subproblem, the FO-Search algorithm (algorithm 1) is complete even in the presence of unbounded domains. The main drawbacks of the algorithm are: (i) it does not use local consistency (ii) every instance $CSP(i)$ is solved from scratch. In next sections we will study how the FO-Search algorithm could be improved with these suitable properties.

4 Local Consistency for OCSPs

Given an unsolvable CSP instance and a static ordering of its variables $o = x_1, \ldots, x_n$, the *failed variable* along the ordering o is the deepest variable in the search tree developed by BT that detects inconsistency, following the ordering o . Different variable orderings may identify different failed variables, all being equally acceptable.

As shown in previous section, the unbounded domains of the OCSP model could lead us to incomplete algorithms if we do not choose the correct variable. The interest for finding a failed variable x_k of an unsolvable problem in the OCSP context is clear: x_k belongs to a minimal unsolvable subproblem, for which more values are required in order to make it solvable (otherwise the whole problem will continue being unsolvable). Variable x_k is the natural candidate to get more values, to extend this minimal unsolvable subproblem.

Local consistency has been shown to be essential in constraint satisfaction to increase the solving performance. We think that local consistency should plays a similar role in OCSP. With this aim, we explore how the popular Forward Checking (FC) algorithm [11] can be used in the OCSP context.

In order to identify a failed variable when having local consistency in OCSPs, we provide the next proposition.

Proposition 2. Let an unsolvable CSP be explored by a FC algorithm with static variable ordering $o = (x_1, ..., x_n)$, and let x_k be the deepest variable in the search tree for which an empty domain was detected. Then there exists an ordering for which x_k is the failed variable in the BT algorithm.

Proof. Let us build the search tree that BT will traverse following the static ordering o. We know that BT will not visit any branch below x_k level (otherwise, BT will find as consistent an assignment that FC detected as inconsistent). BT may fail before x_k level at some branches, but it will not go below that level in any branch. It may happen:

1. There is at least one branch for which BT reaches x_k level.
2. In all branches, BT fails before reaching x_k level.

If 1, x_k is the failed variable for o. If 2, however, x_k is not the failed variable for o since BT never reaches it in any branch. Assuming that x_j is the deepest inconsistent variable found by BT along the ordering o, we construct the ordering o' that is equal to o but where x_j and x_k exchange places. Along this new ordering o', BT finds x_k as the failed variable. To see this, it is enough to realize that FC never instantiates more than $x_1, \ldots, x_{j-1}$ variables (otherwise FC would have instantiated an inconsistent assignment, something impossible [10]) and instantiating these variables is enough to detect inconsistency on x_k. BT following ordering o' will find x_k as the failed variable. Therefore, there is an ordering for which x_k is the failed variable. □

It is important to point that this failed variable could be different from the failed variable obtained in Proposition 1. Therefore, we cannot assure that this failed variable is part of every unsolvable subproblem of the CSP involving variables in the set $\{x_1..x_k\}$.

Using proposition 2, we are able to identify a failed variable even when having local consistency. This property will be useful to create a complete algorithm which uses local consistency for solving OCSP.

5 Using Factoring Out Failure in OCSPs

Beside local consistency, we can improve the FO-Search algorithm avoiding to re-explore the search space already explored. To achieve this, two solutions could be performed:

- Reuse the no-goods found in the earlier search. This no-good recording method is explained in [8,9]. It seems that this is the most obvious solution, but is not practical because new acquired values may invalidate the no-goods inferred in previous searches.
- Using the technique of decomposing a CSP into subproblems proposed by Freuder and Hubbe [4] called *Factoring Out Failure*.

The decomposition process and the algorithm called *Factoring Out Failure* are described in [4]. The algorithm extracts unsolvable subproblems from a CSP, thus limiting search effort to smaller and possible solvable subproblems. This idea seems to apply well to OCSP: we can decompose the new instance $CSP(i)$ obtained after collecting new values, into the old problem just searched $CSP(i-1)$ (which is known to be unsolvable) and a new one based on the values just obtained. This extraction method is shown in Algorithm 2. We have to be careful here, because limiting search to the new found values may cause incompleteness of the algorithm for solving the OCSP.

Algorithm 2. The Extract procedure

```
procedure Extract ($CSP(i-1), CSP(i), decomposition$)
begin
repeat
    Pick a variable, $x_k \in CSP(i)$ whose domain does not match in $CSP(i-1)$.
    Divide $CSP(i)$ into two subproblems $CSP_1$ and $CSP_2$ that differ only in that the domain of
    $x_k$ matches the first subproblem $CSP_1$ while the remaining values of $x_k$ matches the second
    subproblem $CSP_2$.
    $CSP(i) \leftarrow CSP_1; decomposition \leftarrow decomposition \cup CSP_2$
    Apply Extract to the updated problem $CSP(i)$ and $decomposition$ with the same subproblem
    $CSP(i-1)$.
until $CSP(i) = CSP(i-1)$
return $decomposition$;
end Extract
```

To obtain a complete algorithm, we study the relation between the failed variables of two consecutive instances $CSP(i-1)$ and $CSP(i)$.

Let be $CSP(i) = CSP(i-1) \cup CSP_{nv}$ where:

$$CSP_{nv} =< X, D'_{nv}, C(i) >= \begin{cases} D'_{nv}(x_k) = D(x_k) \cup nv & x_k \text{ failed variable in } CSP(i-1) \\ D'_{nv}(x_i) = D(x_i) & \forall x_i \neq x_k \end{cases} \quad (1)$$

Let say we have solved instance $CSP(i-1)$ with variable order $x_0, \dots, x_k, \dots, x_n$. Assume that it was found unsolvable with x_k as failed variable. Thus, following the FCO-Search algorithm, instance $CSP(i)$ will be solved with variable order $x_k, x_0, \dots, x_j, \dots, x_{k-1}, \dots, x_n$. Let say that problem CSP_{nv} was solved using the same variable order $x_k, x_0, \dots, x_j, \dots, x_{k-1}, \dots, x_n$, and it was found unsolvable with x_j as failed variable. Comparing both failed variables, x_k and x_j, we can identify one of these situations:

– The depth level of variable x_j in CSP_{nv} is greater than or equal to the depth level of variable x_k in $CSP(i-1)$. Note that solving instance $CSP(i)$ with variable order $x_k, x_0, \dots, x_j, \dots, x_{k-1}, \dots, x_n$ has as possible candidates for failed variable the set $\{x_0, \dots, x_{k-1}\}$ where x_{k-1} is the deepest variable. Also note that the depth level of variable x_{k-1} in the variable order $x_k, x_0, \dots, x_j, \dots, x_{k-1}, \dots, x_n$ is the same as x_k in the variable order $x_0, \dots, x_k, \dots, x_n$ used for solving instance $CSP(i-1)$. Thus, we can conclude that x_j is the failed variable of instance $CSP(i)$, because it is deeper that the deepest possible candidate x_{k-1} which has the same depth level that x_k in instance $CSP(i-1)$. Note that in this situation, we just need to solve CSP_{nv} in order to know the failed variable of instance $CSP(i)$ or found a possible solution.
– The depth level of variable x_j in CSP_{nv} is lower than the depth level of variable x_k in instance $CSP(i-1)$. As before, solving instance $CSP(i)$ has as possible candidates for failed variable the set $\{x_0, \dots, x_{k-1}\}$ where x_{k-1} is the deepest variable. In this case, the set of failed variables is $\{x_0, \dots, x_{k-1}\} \cup \{x_j\}$. Therefore, we have to solve $CSP(i)$ from scratch with the new variable ordering $x_k, x_0, \dots, x_j, \dots, x_{k-1}, \dots, x_n$ to know which is the failed variable.

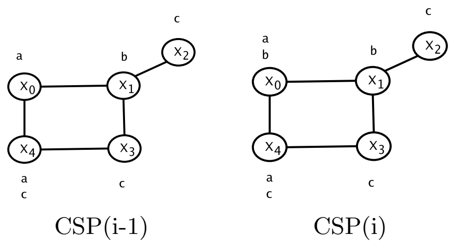


Fig. 3. An Open graph coloring example

Figures 3 and 4 show an example of solving instance $CSP(i)$ using the *Factoring Out Failure* algorithm. Figure 3 (left) shows an unsolvable instance $CSP(i-1)$ of a graph coloring OCSP with variable ordering $\{x_1, x_2, x_3, x_4, x_0\}$. In the example, x_0 is the failed variable, thus following the FO-Search algorithm we collect a new value $x_0 = b$ obtaining the instance $CSP(i)$ as shown in Figure 3(right) with a new variable ordering $\{x_0, x_1, x_2, x_3, x_4\}$. The FO-Search algorithm will solve again from scratch instance $CSP(i)$ although we know that

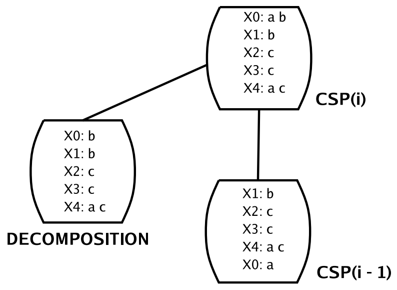


Fig. 4. An example of instance $CSP(i)$ decomposition

instance $CSP(i-1)$ is an unsolvable subset of $CSP(i)$. Before solve $CSP(i)$, we can use the Algorithm 2 to extract the known unsolvable subproblem $CSP(i-1)$ and then focusing on solving the decomposition subproblem obtained [1] (for more details of the *Factoring Out Failure* method, please refer to [4]). When we extract the subproblem $CSP(i-1)$ from problem $CSP(i)$ we obtain a new subproblem (called *DECOMPOSITION* in figure 4) which is smaller and probably easy to solve than instance $CSP(i)$.

6 The FCO-Search Algorithm

Based on the previous FO-Search algorithm, we developed a new algorithm with the same properties (new values have to be gathered only when the current instance is unsolvable) but including local consistency and using the Factoring Out Failure decomposition. We call the obtained algorithm FCO-Search.

The FCSearch function, implements the classical FC algorithm. In line 7, it assigns a new value to variable x_i and starts the propagation (lines 8-14). Lines 15-17 check if there is a variable x_j which domain was exhausted by the previous assignment of variable x_i, recording the deepest variable with empty domain. It returns either a solution or a failed variable if instance $CSP(i)$ is unsolvable.

The initial call of the FCO-Search (Algorithm 3) is $FCO-Search(CSP(0), x_0)$ where $CSP(0)$ is the initial instance and x_0 is the first variable in the variable order given by this initial instance. The FCO-Search algorithm has two parameters: $< X(i), D(i), C(i) >$ as instance $CSP(i)$ and x'_k as the failed variable of instance $CSP(i-1)$. In line 2 we use the *Extract* procedure to extract the unsolvable instance $CSP(i-1)$ from instance $CSP(i)$. From this decomposition, we obtain the subproblem $< X_d, D_d, C_d >$ which is solved by the FCSearch function in line 3. If there is no solution, then it compares in line 7 if the failed variable x_k of problem $CSP(i)$ is deeper than the previous failed variable x'_k of instance $CSP(i-1)$. When this condition is true, we know that x_k is the failed variable of instance $CSP(i)$. Otherwise we need to solve $CSP(i-1)$ to calculate the new failed variable (line 8). Once the new failed variable is calculated, we call the function **more** in line 15

[1] Problem $CSP(i)$ differs from problem $CSP(i-1)$ only in the new queried value.

```
1: function FCSearch(X,D,C)
2: $i \leftarrow 1, k \leftarrow 1$
3: repeat {main loop}
4:    if $exhausted(d_i)$ then
5:       $i \leftarrow i - 1$
6:    else
7:       $x_i \leftarrow nextvalue(d_i)$
8:       for all $x_j \in (x_{i+1}, \ldots, x_n)$ do
9:          for all $a \in d_j$ do
10:            if $\neg consistent(x_1, \ldots, x_i, x_{i+1}, \ldots, x_n)$ then
11:               $d_j \leftarrow d_j - a$
12:            end if
13:         end for
14:      end for
15:      if $\{\exists x_j \in (x_{i+1}, \ldots, x_n) \mid exhausted(d_j)\}$ then {backtrack}
16:         reset each $x_j \in (x_{i+1}, \ldots, x_n)$ to value before $x_i$ was set
17:         $k \leftarrow max(k, j), i \leftarrow i - 1$
18:      else
19:         $i \leftarrow i + 1$
20:      end if
21:      if $i > n$ then
22:         return $\{x_1, ..., x_n\}$ as a solution
23:      end if
24:   end if
25: until $i = 0$
26: return $x_k$ as failed variable
27: end FCSearch
```

and a new value is added. Line 21 reorders variables so that x_k becomes the first variable and the relative order of other variables remains the same. Finally line 22 calls recursively algorithm FCO-Search with instance $CSP(i)$ and with the new failed variable.

Proposition 3. Algorithm FCO-Search is complete.

Proof. We know that the FO-Search algorithm is complete [6], where search is performed by BT. We will show that the FCO-Search algorithm, where BT is replaced by FC, is also complete. Let x_k be the failed variable found by FC, and x_j the failed variable found by BT, both along the ordering o. Either (i) $x_j = x_k$ or (ii) x_j appears before x_k in o. In (i) the FCO-Search algorithm behaves like FO-search, so it is obviously complete. Then, let us assume (ii). In this case, there is an ordering o' equal to o but with x_j and x_k exchanging places. Along o' BT would have found x_k as failed variable. We show that FCO-Search with ordering o behaves like FO-Search with ordering o'. First, both algorithms ask for one more value for x_k and put it as the first variable, forming the ordering $x_k, x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_j, \ldots, x_n$. We know that the subset $x_k, x_1, \ldots, x_{j-1}$ formed a unsatisfiable subproblem in the previous iteration, but we do not know if the new value of x_k has made it solvable. If it is solvable, then the algorithm will continue looking for the next unsatisfiable subproblem (if any). If not, FCO-Search will find a new failed variable x_p in the sequence $x_k, x_1, \ldots, x_{j-1}$ using the FC algorithm. Obviously, x_p appears before or it is equal to x_{j-1}. Since BT found a consistent instantiation from x_1 until x_{j-1}, in this subset BT cannot find a constraint that will stop search before reaching x_p. So BT would find the same

Algorithm 3. *The* **FCO-Search** *algorithm*

```
1: procedure FCO-Search($< X(i), D(i), C(i) >, x'_k$)
2: $< X_d, D_d, C_d >= Extract(< X(i), D(i), C(i) >, < X(i-1), D(i-1), C(i-1) >, \emptyset)$
3: if $solution(FCSearch(X_d, D_d, C_d))$ then
4:    return $\{x_1, ..., x_n\}$ as a solution
5: end if
6: $x_k$ = returned $x_k$ by the FCSearch function
7: if $x'_k > x_k$ then
8:    $x_k = FCSearch(X(i-1), D(i-1), C(i-1))$
9: end if
10: if $e_k = CLOSED$ then
11:    if $(\forall i \in 1..k-1) e_k = CLOSED$ then
12:       return failure
13:    end if
14: else
15:    $nv \leftarrow$ more($x_k$)
16:    if $nv =$ nomore($x_k$) then
17:       $e_k \leftarrow CLOSED$
18:    end if
19:    $d_k \leftarrow nv \cup d_k$
20: end if
21: reorder variables so that $x_k$ becomes $x_1$ (relative order of others remains the same)
22: FCO-Search($X(i), D(i), C(i), x_k$)
23: end FCO-Search
```

failed variable as FC. Therefore, FCO-Search will behave exactly as FO-search, until finding a solution for that unsatisfiable subproblem.

In both cases (i) and (ii), FCO-search behaves like FO-search with ordering (i) o, or (ii) o'. Since FO-Search is complete, FCO-Search is complete. □

7 Experiments

We compared the performance of the new FCO-Search algorithm against the FO-Search algorithm [2] on solvable random OCSPs. We performed several experiments.

As a first experiment, we compared the performance of the algorithms until a solution is found when density and tightness change. At this point we want to emphasize that our experimental results are done with finite domain random classes (following the B random model) in order to allow researches to reproduce the experiments. For this experiment we generated 1000 random OCSPs with 7 variables and with a domain size of 10 values. Figures 5 (a)(b) compare the number of checks needed to find a solution for the OCSP when (a) density = 0.2 (b) density = 0.8 and tightness moves from 0.1 to 0.8. In Figure 5 (c) density = 0.8 but tightness moves from 0.1 to 0.6 due to the difficulty of generating solvable problems when tightness > 0.6 .

Figure 5 shows an important improvement of the FCO-Search algorithm over the FO-Search in any case. Combining local consistency and avoid solving from scratch the same problem reduces dramatically the number of constraint checks

[2] The FCO-Search algorithm is compared against the backtracking version of the FO-Search without backjumping.

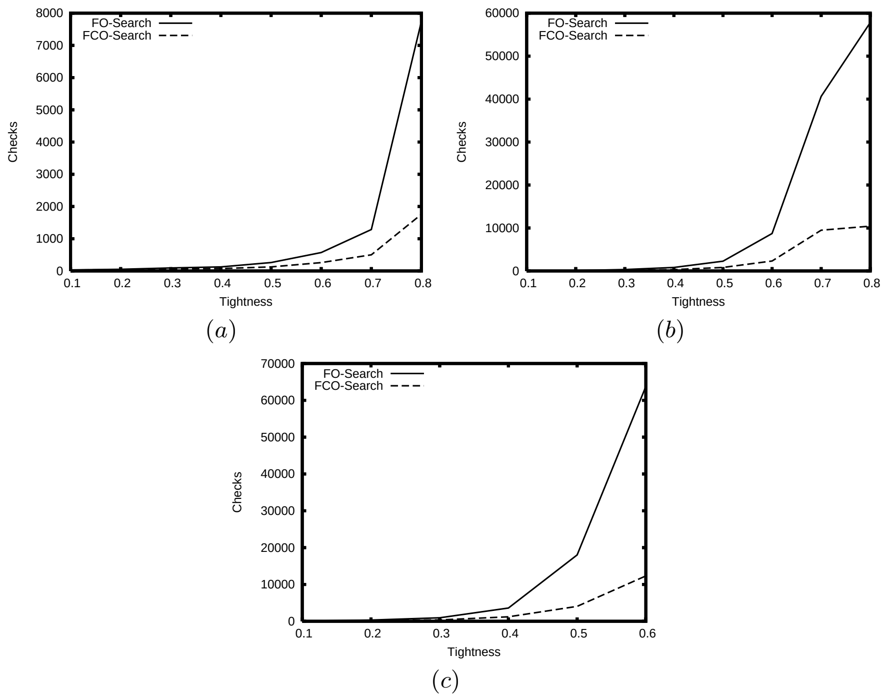


Fig. 5. Comparison of the number of checks when tightness increases. Top: (a) when density = 0.2, (b) when density = 0.5. Bottom: (c) when density = 0.8.

for hard problems producing a substantial improvement in the performance of the proposed algorithm.

In the second experiment, we compared the algorithms in two aspects: the number of accesses to information sources and the number of constraint checks until a solution for the OCSP is found [3]. We generated 100000 random OCSPs with between 5 to 17 variables and a domain size of 10 values, forcing the graph to be solvable and at least connected and at most complete with random density and tightness.

Figure 6(a) shows the number of checks against the number of variables, studying the performance of the algorithms when increasing the number of variables. The benefits are very important (it is "likely" to get at least an order of magnitude for bigger instances) because the FCO-Search algorithm incorporates local consistency and avoids redoing the same solving process every time a new value is added (as explained in previous section).

Figure 6(a) compares the number of queries against the number of variables. In this figure, we include the *Classical CSP* approach which decouples information gathering and problem solving. This approach first queries all values for all variables and then solves the problem. We decided to include it because it

[3] These constraint checks are not hidden in the FCO-Search algorithm.

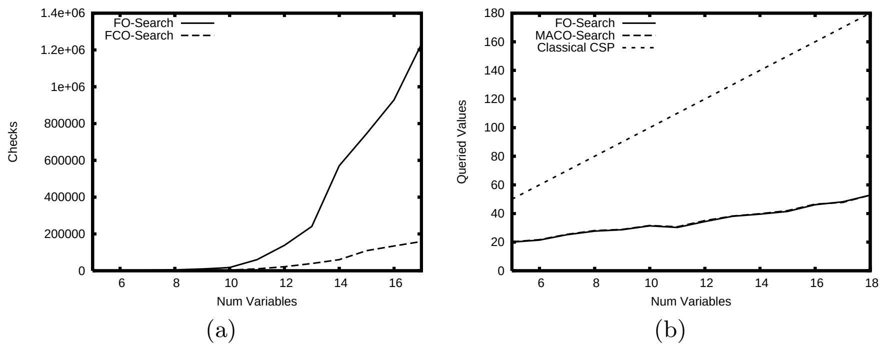


Fig. 6. (a) Comparison of the number of checks vs the number of variables when domain size = 10 (b) Comparison of the number of values queried vs the number of variables when domain size = 10

establishes an upper bound for the performance of the algorithms. As mentioned before, it could be possible that both algorithms FO-Search and FCO-Search do not select the same failed variable in every instance $CSP(i)$, so the number of queried values could vary. Empirical results show that the number of queries is nearly the same for both algorithms, having a slightly better performance the FO-Search algorithm. As explained before, this is due to the property that the FO-Search algorithm finds a failed variable x_k which participates in all unsolvable subproblems $\{x_1, \ldots, x_k\}$, while the FCO-Search algorithm just finds a component that may does not have this property. Thus, the selected variable by the FO-Search algorithm gets a new value that could solve several unsolvable subproblems at the same time, while the selected variable by the FCO-Search can not assure this property. Despite this difference, empirical results show that both algorithms have a similar performance.

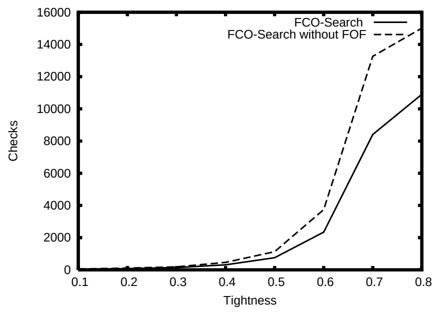
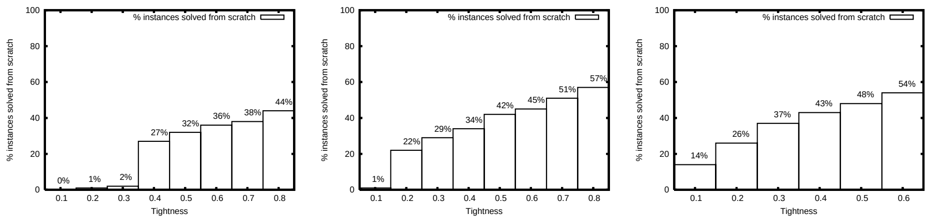


Fig. 7. Comparison of the number of checks when the FCO-Search algorithm uses Factoring out Failure and without it. (density = 0.5)



density = 0.2 density = 0.5 density = 0.8

Fig. 8. Number of times instances are solved from scratch when density and tightness increases

The benefits of our new approach come from a combination of applying local consistency and avoid solving instances from scratch. We are interested in the influence of these improvements separately. Figure 7 compares the number of checks when the FCO-Search algorithm uses the Factoring out Failure method and when it does not use it. For this particular experiment, we generated 1000 random OCSPs with 7 variables with domain size 10 and with a *density* = 0.5. In figure 8, we studied the percentage of instances solved from scratch for every random OCSP. For these instances, solving the obtained decomposition subproblem is not enough for finding the next failed variable. As expected, figure 8 shows that when problems become harder, around 60% of instances are solved from scratch. For these hard problems, if we compare Figure 7 and Figure 8, we can see that local consistency bring us more benefits, although the contribution of the Factoring out Failure method is also significant.

8 Conclusions

The performance of algorithms for solving OCSPs is very poor because they neither use local consistency nor avoid solving subproblems already explored in previous step. We have studied how we can incorporate local consistency while keeping the completeness of the algorithm by finding a failed variable. We also have shown how we can use the idea of the Factoring out Failure method proposed by Freuder [4] to avoid redoing the previous work. Based on these two techniques, we have developed a new algorithm called *FCO-Search* for solving OCSPs. The results described in last section show a significant speed-up in the number of checks compared with the previous *FO-Search* algorithm while the number of queried values remains nearly the same, even when the problem becomes hard to solve.

As future work, we are studying how to incorporate dynamic variable ordering into our algorithm. This new improvement poses a new challenge when calculating the failed variable, but we suspect it will provide a promising improvement for OCSPs solvers.

References

1. Marian Nodine and Jerry Fowler and Tomasz Ksiezyk and Brad Perry and Malcolm Taylor and Amy Unruh : Active Information Gathering in InfoSleuth. International Journal of Cooperative Information Systems **9** (2000) 3-28
2. Genesereth M. and R. Keller and A. M. Duschka: Infomaster: An Information Integration System. Proceedings of 1997 ACM SIGMOD Conference, May 1997
3. Alon Y. Levy and Anand Rajaraman and Joann J. Ordille: Querying Heterogeneous Information Sources Using Source Descriptions. Proceedings of the Twenty-second International Conference on Very Large Databases, VLDB Endowment, Saratoga, Calif, Bombay, India (1996) 251–262
4. Eugene C. Freuder and Paul D. Hubbe: Extracting Constraint Satisfaction Subproblems. IJCAI - 1995, 548-557.
5. S. Macho-Gonzalez and Pedro Meseguer. Open, Interactive and Dynamic CSP. Changes'05 International Workshop on Constraint Solving under Change and Uncertainty (CP'05).
6. Boi Faltings and Santiago Macho-Gonzalez: Open Constraint Programming. Artificial Intelligence 161, (2005) 181–208.
7. Gio Wiederhold and Michael R. Genesereth: The Conceptual Basis for Mediation Services. IEEE Expert volume 12 (5) 38–47 (1997).
8. Yuejun Jiang and Thomas Richards and Barry Richards: No-good backmarking with min-conflicts repair in constraint satisfaction and optimization. Proceedings of Principles and Practice of Constraint Programming 94, (1994) 21–39.
9. Thomas Schiex and Gérard Verfaillie: Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. International Journal on Artificial Intelligence Tools, volume 3 (2) 187–207 (1994).
10. Grzegorz Kondrak and Peter van Beek: A theoretical evaluation of selected backtracking algorithms. Artificial Intelligence 89, (1997) 365–387.
11. Haralick and Elliot: Increasing tree search efficiency for constraint-satisfaction problems. Artificial Intelligence 14 (3) 263-313, (1980).
12. F. Rossi and C. Petrie and V. Dhar: In the equivalence of constraint satisfaction problems. Proc. ECAI-90 550–556 (1990).
13. Kostas Stergiou and Toby Walsh : Encodings of Non-binary Constraint Satisfaction Problems. Proceedings of AAAI/IAAI 163-168 (1999).
14. Evelina Lamma and Paola Mello and Michela Milano and Rita Cucchiara and Marco Gavanelli and Massimo Piccardi: Constraint Propagation and Value Acquisition: Why we should do it Interactively. Proceeding of IJCAI'99 468-477 1999.
15. Thomas Schiex and Gérard Verfaillie: Nogood Recording for Static and Dynamic Constraint Satisfaction Problem, International Journal of Artificial Intelligence Tools (3)2 187–207, (1994).

Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs)

Robert Mateescu and Rina Dechter

Donald Bren School of Information and Computer Science
University of California, Irvine, CA 92697-3425
{mateescu, dechter}@ics.uci.edu

Abstract. Inspired by AND/OR search spaces for graphical models recently introduced, we propose to augment Ordered Decision Diagrams with AND nodes, in order to capture function decomposition structure. This yields *AND/OR multi-valued decision diagram* (AOMDD) which compiles a constraint network into a canonical form that supports polynomial time queries such as solution counting, solution enumeration or equivalence of constraint networks. We provide a compilation algorithm based on Variable Elimination for assembling an AOMDD for a constraint network starting from the AOMDDs for its constraints. The algorithm uses the APPLY operator which combines two AOMDDs by a given operation. This guarantees the complexity upper bound for the compilation time and the size of the AOMDD to be exponential in the treewidth of the constraint graph, rather than pathwidth as is known for ordered binary decision diagrams (OBDDs).

1 Introduction

The work presented in this paper is based on two existing frameworks: (1) AND/OR search spaces for graphical models and (2) decision diagrams (DD). AND/OR search spaces [1,2,3] have proven to be a unifying framework for various classes of search algorithms for graphical models. The main characteristic is the exploitation of independencies between variables during search, which can provide exponential speedups over traditional search methods that can be viewed as traversing an OR structure. The AND nodes capture problem decomposition into **independent subproblems**, and the OR nodes represent branching according to variable values. Backjumping schemes for constraint satisfaction and satisfiability can be shown to explore the AND/OR space automatically if only one solution is sought. However, for counting and other enumeration tasks a deliberate exploration of the AND/OR space is beneficial [4].

Decision diagrams are widely used in many areas of research, especially in software and hardware verification [5,6]. A BDD represents a Boolean function by a directed acyclic graph with two sink nodes (labeled 0 and 1), and every internal node is labeled with a variable and has exactly two children: *low* for 0 and *high* for 1. If isomorphic nodes were not merged, on one extreme we would have the full search *tree*, also called Shannon tree, which is the usual full tree explored by backtracking algorithm. The tree can be ordered if we impose that variables be encountered in the same order along every branch. It can then be compressed by merging isomorphic nodes (i.e., with the same label and identical children), and by eliminating redundant nodes (i.e., whose *low* and *high* children are identical). The result is the celebrated *reduced ordered binary decision diagram*, or OBDD for short, introduced by Bryant [7]. However, the underlying structure is OR, because the initial Shannon tree is an OR tree. If AND/OR search trees are reduced by node merging and redundant nodes elimination we get a compact search graph that can be viewed as a BDD representation augmented with AND nodes.

In this paper we combine the two ideas, in order to create a decision diagram that has an AND/OR structure, thus exploiting problem decomposition. As a detail, the number of values is also increased from two to any constant, but this is less significant for the algorithms. Our proposal is closely related to two earlier research lines within the BDD literature. The first is the work on Disjoint Support Decompositions (DSD) investigated within the area of design automation [8], that were proposed recently as enhancements for BDDs aimed at exploiting function decomposition [9]. The second is the work on BDDs trees [10]. Another related proposal is the recent work by Fargier and Vilarem [11] on compiling CSPs into tree-driven automata.

A decision diagram offers a compilation of a problem. It typically requires an extended offline effort in order to be able to support polynomial (in its size) or constant time online queries. In the context of constraint networks, it could be used to represent the whole set of solutions, to give the solutions count or solution enumeration and to test satisfiability or equivalence of constraint networks. The benefit of moving from OR structure to AND/OR is a lower complexity of the algorithms and size of the compiled structure. It typically moves from being bounded exponentially in *pathwidth* pw^*, which is characteristic to chain decompositions or linear structures, to being exponentially bounded in *treewidth* w^*, which is characteristic of tree structures (it always holds that $w^* \leq pw^*$ and $pw^* \leq w^* \cdot \log n$).

Our contribution consists of: (1) we formally describe the AND/OR multi-valued decision diagram (AOMDD) and prove that it is a canonical representation of a constraint network; (2) we describe the APPLY operator that combines two AOMDDs by an operation and prove its complexity to be linear in the output. We show that the output of apply is bounded by the product of the sizes of the inputs. (3) we give a scheduling of building the AOMDD of a constraint network starting with the AOMDDs of its constraints. It is based on an ordering of variables, which gives rise to a pseudo tree (or bucket tree) according to the execution of Variable Elimination algorithm. This gives the complexity guarantees in terms of the *induced width* along that ordering (equal to the treewidth of the corresponding decomposition).

The structure of the paper is as follows: Sect. 2 provides preliminary definitions, a description of Variable Elimination and AND/OR search spaces; Sect. 3 describes the AOMDD, its graphical representation and properties, and demonstrates its compilation by Variable Elimination; Sect. 5 presents the APPLY operation; Sect. 6 discusses extensions to probabilistic models; Sect. 7 presents related work and Sect. 8 concludes.

2 Preliminaries

A constraint network and its associated graph are defined in the usual way:

Definition 1 (constraint network). *A* constraint network *is a 3-tuple* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$*, where:* $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a set of variables;* $\mathbf{D} = \{D_1, \ldots, D_n\}$ *is the set of* *their finite domains of values, with cardinalities* $k_i = |D_i|$ *and* $k = \max_{i=1}^{n} k_i$; $\mathbf{C} = \{C_1, \ldots, C_r\}$ *is a set of constraints over subsets of* $\mathbf{X}$. *Each constraint is defined as* $C = (S_i, R_i)$*, where* S_i *is the set of variables on which the constraint is defined, called its scope, and* R_i *is the relation defined on* S_i.

Definition 2 (constraint graph). *The* constraint graph *of a constraint network is an undirected graph,* $G = (\mathbf{X}, E)$*, that has variables as its vertices and an edge connecting any two variables that appear in the scope (set of arguments) of the same constraint.*

A pseudo tree resembles the tree rearrangements introduced in [12]:

Definition 3 (pseudo tree). *A* pseudo tree *of a graph* $G = (\mathbf{X}, E)$ *is a rooted tree* $\mathcal{T}$ *having the same set of nodes* $\mathbf{X}$*, such that every arc in* E *is a backarc in* $\mathcal{T}$ *(i.e., it connects nodes on the same path from root).*

Definition 4 (induced graph, induced width, treewidth, pathwidth). *An* ordered graph *is a pair* (G, d)*, where* G *is an undirected graph, and* $d = (X_1, ..., X_n)$ *is an ordering of the nodes. The* width of a node *in an ordered graph is the number of neighbors that precede it in the ordering. The* width of an ordering d*, denoted* $w(d)$*, is the maximum width over all nodes. The* induced width of an ordered graph*,* $w^*(d)$*, is the width of the induced ordered graph obtained as follows: for each node, from last to first in* d*, its preceding neighbors are connected in a clique. The* induced width of a graph*,* w^**, is the minimal induced width over all orderings. The induced width is also equal to the* treewidth *of a graph. The* pathwidth pw^* *of a graph is the treewidth over the restricted class of orderings that correspond to chain decompositions.*

2.1 Variable Elimination (VE)

Variable elimination (**VE**) [13,14] is a well known algorithm for inference in graphical models. We will describe it using the terminology from [14]. Consider a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ and an ordering $d = (X_1, X_2, \ldots, X_n)$. The ordering d dictates an elimination order for **VE**, from last to first. Each constraints from **C** is placed in the bucket of its latest variable in d. Buckets are processed from X_n to X_1 by eliminating the bucket variable (the constraints residing in the bucket are joined together, and the bucket variable is projected out) and placing the resulting constraint (also called *message*) in the bucket of its latest variable in d. After its execution, **VE** renders the network backtrack free, and a solution can be produced by assigning variables along d. **VE** can also produce the solutions count if marginalization is done by summation (rather than projection) and join is substituted with multiplication.

VE also constructs a bucket tree, by linking the bucket of each X_i to the destination bucket of its message (called the parent bucket). A node in the bucket tree typically has a *bucket variable*, a *collection of constraints*, and a *scope* (the union of the scopes of its constraints). If the nodes of the bucket tree are replaced by their respective bucket variables, it is easy to see that we obtain a pseudo tree.

Example 1. Figure 1a shows a network with four constraints. Figure1b shows the execution of Variable Elimination along $d = (A, B, E, C, D)$. The buckets are processed from D to A [1]. Figure 1c shows the bucket tree. The pseudo tree is given in Fig. 2a.

[1] This representation reverses the top down bucket processing described in earlier papers.

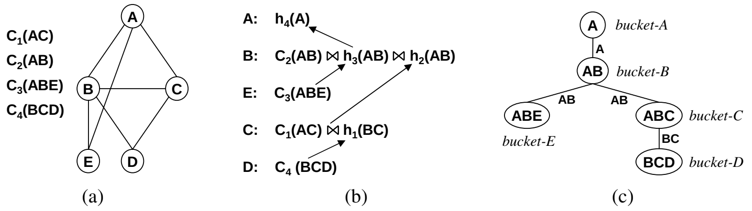


Fig. 1. Execution of Variable Elimination

2.2 AND/OR Search for Constraint Problems

The AND/OR search space is a recently introduced [1,2,3] unifying framework for advanced algorithmic schemes for graphical models. Its main virtue consists in exploiting independencies between variables during search, which can provide exponential speedups over traditional search methods oblivious to problem structure.

Definition 5 (AND/OR search tree of a constraint network). *Given a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, its constraint graph G and a pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree has alternating levels of OR and AND nodes. The OR nodes are labeled X_i and correspond to variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ and correspond to value assignments. The structure of the AND/OR search tree is based on $\mathcal{T}$. The root is an OR node labeled with the root of $\mathcal{T}$. The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$ that are consistent with the assignments along the path from the root. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in the pseudo tree $\mathcal{T}$. The leaves of AND nodes are labeled with "1". There is a one to one correspondence between solution subtrees of the AND/OR search graph and solutions of the constraint network [1].*

The AND/OR search tree can be traversed by a depth first search algorithm, thus using linear space. It was already shown [12,15,16,1] that:

Theorem 1. *Given a constraint network $\mathcal{R}$ and a pseudo tree $\mathcal{T}$ of depth m, the size of the AND/OR search tree based on $\mathcal{T}$ is $O(n\ k^m)$, where k bounds the domains of variables. A constraint network of treewidth w^* has a pseudo tree of depth at most $w^* \log n$, therefore it has an AND/OR search tree of size $O(n\ k^{w^* \log n})$.*

The AND/OR search tree may contain nodes that root identical conditioned subproblems. These nodes are said to be *unifiable*. When unifiable nodes are merged, the search space becomes a graph. Its size becomes smaller at the expense of using additional memory by the search algorithm. The depth first search algorithm can therefore be modified to cache previously computed results, and retrieve them when the same nodes are encountered again. The notion of unifiable nodes is defined formally next.

Definition 6 (isomorphism, minimal AND/OR graph). *Two AND/OR search graphs G and G′ are* isomorphic *if there exists a one to one mapping σ from the vertices of G to the vertices of G′ such that for any vertex v, if $\sigma(v) = v'$, then v and v' root identical*

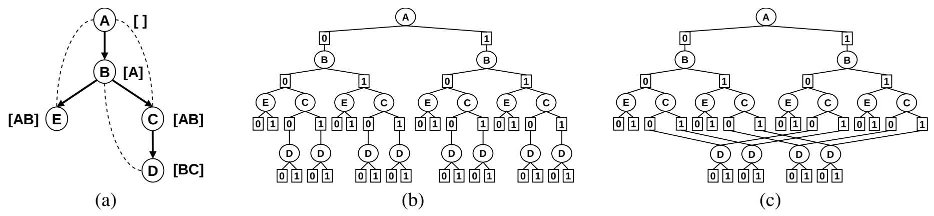


Fig. 2. AND/OR search space

subgraphs relative to σ. The minimal AND/OR graph *is such that all the isomorphic subgraphs are merged. Isomorphic nodes are also called unifiable.*

Some unifiable nodes can be identified based on their *contexts*. We can define graph based contexts for both OR nodes and AND nodes, just by expressing the set of ancestor variables in $\mathcal{T}$ that completely determine a conditioned subproblem. However, it can be shown that using caching based on OR contexts makes caching based on AND contexts redundant and vice versa, so we will only use *OR caching*. Any value assignment to the context of X separates the subproblem below X from the rest of the network.

Definition 7 (OR context). *Given a pseudo tree $\mathcal{T}$ of an AND/OR search space, $context(X) = [X_1 \ldots X_p]$ is the set of ancestors of X in $\mathcal{T}$, ordered descendingly, that are connected in the primal graph to X or to descendants of X.*

Definition 8. *The* context minimal *AND/OR graph is obtained from the AND/OR search tree by merging all the context unifiable OR nodes.*

It was already shown that [15,1]:

Theorem 2. *Given a constraint network $\mathcal{R}$, its primal graph G and a pseudo tree $\mathcal{T}$, the size of the context minimal AND/OR search graph based on $\mathcal{T}$ is $O(n\ k^{w^*_{\mathcal{T}}(G)})$, where $w^*_{\mathcal{T}}(G)$ is the induced width of G over the depth first traversal of $\mathcal{T}$, and k bounds the domain size.*

Example 2. Figure 2 shows AND/OR search spaces for the constraint network from Fig. 1, assuming universal relations (no constraints) and binary valued variables. When constraints are tighter, some of the paths in these graphs do not exists. Figure 2a shows the pseudo tree derived from ordering $d = (A, B, E, C, D)$, having the same structure as the bucket tree for this ordering. The (OR) context of each node appears in square brackets, and the dotted arcs are backarcs. Notice that the context of a node is identical to the message scope from its bucket in Fig. 1. Figure 2b shows the AND/OR search tree, and 2c shows the context minimal AND/OR graph.

3 AND/OR Multi-valued Decision Diagrams (AOMDDs)

The *context minimal* AND/OR graph, described in section 2.2 offers an effective way of identifying unifiable nodes during the execution of the search algorithm. However, merging based on context is not complete, i.e. there may still be unifiable nodes in the search graph that do not have identical contexts. Moreover, some of the nodes in the context minimal AND/OR graph may be redundant, for example when the set of solutions rooted at variable X_i is not dependant on the specific value assigned to X_i (this situation is not detectable based on context).

As overviewed earlier, in [1] we defined the complete *minimal AND/OR graph* which is an AND/OR graph whose all unifiable nodes are merged and we also proved canonicity [4]. Here we propose to augment the minimal AND/OR search graph with removing redundant variables as is common in OBDD representation as well as adopt some notational conventions common in this community. This yields a data structure that we call AND/OR BDDs, that exploits decomposition by using AND nodes. We present this extension over multi-valued variables yielding AND/OR MDD or AOMDD.

3.1 AND/OR Relation Graphs and Canonical AND/OR Decision Diagrams

We first define the AND/OR constraint function graph, which is an AND/OR data structure that defines a relation relative to a tree structure over a set of variables.

Definition 9 (AND/OR constraint function graph). *Given a set of variables* $\mathbf{X} = \{X_1, ..., X_n\}$, *domain of values* $\{D_1, ..., D_n\}$ *and a tree* $\mathcal{T}$ *over* $\mathbf{X}$ *as its nodes, an* AND/OR constraint function graph $\mathcal{G}$ *is a rooted, directed, labeled acyclic graph, having alternating levels of OR and AND nodes and two special terminal nodes labeled "0" and "1". The OR nodes are labeled by variables from* $\mathbf{X}$ *and the AND nodes are labeled by value assignments from respective the domains. There are arcs from nodes to their child nodes defined as follows:*

- *A nonterminal OR vertex v is labeled as $l(v) = X_i$, $X_i \in \mathbf{X}$ and any of its child AND nodes u is labeled $l(u) = \langle X_i, x_i \rangle$, when x_i is a value of X_i. An OR node can also have a single child node which is the terminal "0" node.*
- *An AND node u labeled $l(u) = \langle X_i, x_i \rangle$ has OR child nodes. If an OR child node w of AND node u, is labeled $l(w) = Y$, then Y must be a descendant of X in $\mathcal{T}$. If any two variables Z and Y label two OR child nodes of u where $l(u) = \langle X, x \rangle$, then Z and Y must be on different paths from X down to the leaves in $\mathcal{T}$. An AND node u can also have as a single child node the special node "1".*

The AND/OR constraint function graph defines a relation over the set of variables that are mentioned in $\mathcal{T}$ which can be obtained from the set of all *solution subtrees* of $\mathcal{G}$. A solution subtree of $\mathcal{G}$ contains its root node and if it contains an OR node, then it must contain one of its child nodes, and if it contains an AND node, it must contain all its child nodes. Its only leaf nodes are labeled "1". A solution tree defines a partial assignment that includes all the assignment labeling AND nodes in the solution tree. We say that this partial assignment is generated by $\mathcal{G}$.

Definition 10 (the relation defined by a constraint function graph). *Given an AND/OR constraint function graph $\mathcal{G}$ over variables $\mathbf{X} = \{X_1, ..., X_n\}$ having domain of values $\{D_1, ..., D_n\}$ and a tree $\mathcal{T}$, the relation $rel(\mathcal{G})$ includes all and only full assignments that extend partial assignments generated by $\mathcal{G}$.*

Similar to the case of OBDDs, AND/OR constraint function graphs can be reduced into canonical form by removing *isomorphism* and *redundancy*. The notion of isomorphism was defined earlier for AND/OR graphs. In order to capture the notion of redundancy, it is useful to group every OR node together with its AND children into a *meta-node*. The underlying graph is still an AND/OR graph.

Definition 11 (meta-node). *A nonterminal meta-node v in an AND/OR search graph consists of an OR node labeled $var(v) = X_i$ and its k_i AND children labeled $\langle X_i, x_{i_1} \rangle$, $\ldots$, $\langle X_i, x_{i_{k_i}} \rangle$ that correspond to its value assignments. We will sometimes abbreviate $\langle X_i, x_{i_j} \rangle$, by x_{i_j}. Each AND node labeled x_{i_j} points to a list of child meta-nodes, $u.children_j$. Examples of meta-nodes appear in Fig. 4.*

A variable is considered redundant with respect to the partial assignment of its parents, if assigning it any specific value does not change the value of any possible full assignment. Formally:

Definition 12 (redundant vertex). *Given an AND/OR constraint function graph $\mathcal{G}$, a node v is* redundant *if for all u_1 and u_2 which are AND child nodes of v, u_1 and u_2 have the same child meta nodes.*

If we want to remove a redundant vertex v from an AND/OR constraint function graph $\mathcal{G}$, then we make all of its parent AND nodes v_1, point directly to the common set of its grandchild meta nodes and the meta-node of v is removed from the graph. Finally we define AOMDD:

Definition 13 (AOMDD). *An AND/OR multi-valued decision diagram (AOMDD) is an AND/OR constraint function graph that: (1) contains no redundant vertex and (2) it contains no isomorphic subgraphs.*

Analogously to OBDDs we can show that AOMDDs are a canonical representation of constraint networks, with respect to a given pseudo tree. Thus, AOMDDs can be used as a compiled representation of a constraint network.

Theorem 3 (AOMDDs are canonical). *Given a constraint network, whose constraint set is* $\mathbf{C}$*, having a constraint graph G and given a pseudo tree $\mathcal{T}$ of G, there is a unique (up to isomorphism) reduced AND/OR constraint function graph of* $\mathbf{C}$ *based on $\mathcal{T}$, and any other constraint function graph of* $\mathbf{C}$ *based on $\mathcal{T}$ has more vertices.*

4 Using Variable Elimination to Generate AOMDDs

In this section we propose to use a **VE** type algorithm to guide the compilation of a set of constraints into an AOMDD. Let's look at an example first.

Example 3. Consider the network defined by $\mathbf{X} = \{A, B, \ldots, H\}$, $D_A = \ldots = D_H = \{0, 1\}$ and the constraints ($\oplus$ denotes XOR): $C_1 = F \vee H$, $C_2 = A \vee \neg H$, $C_3 = A \oplus B \oplus G$, $C_4 = F \vee G$, $C_5 = B \vee F$, $C_6 = A \vee E$, $C_7 = C \vee E$, $C_8 = C \oplus D$, $C_9 = B \vee C$. The constraint graph is shown in Figure 3a. Consider the ordering $d = (A, B, C, D, E, F, G, H)$. The pseudo tree (or bucket tree) induced by d is given in Fig. 3b. Figure 4 shows the execution of **VE** with AOMDDs along ordering d. Initially, the constraints C_1 through C_9 are represented as AOMDDs and placed

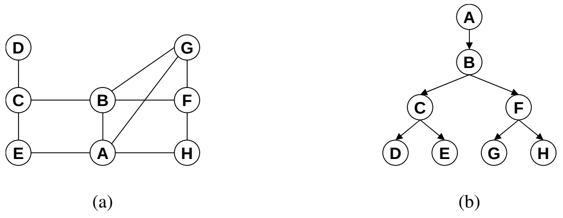


Fig. 3. (a) Constraint graph for $C = \{C_1, \ldots, C_9\}$, where $C_1 = F \vee H$, $C_2 = A \vee \neg H$, $C_3 = A \oplus B \oplus G$, $C_4 = F \vee G$, $C_5 = B \vee F$, $C_6 = A \vee E$, $C_7 = C \vee E$, $C_8 = C \oplus D$, $C_9 = B \vee C$; (b) Pseudo tree (bucket tree) for ordering $d = (A, B, C, D, E, F, G, H)$

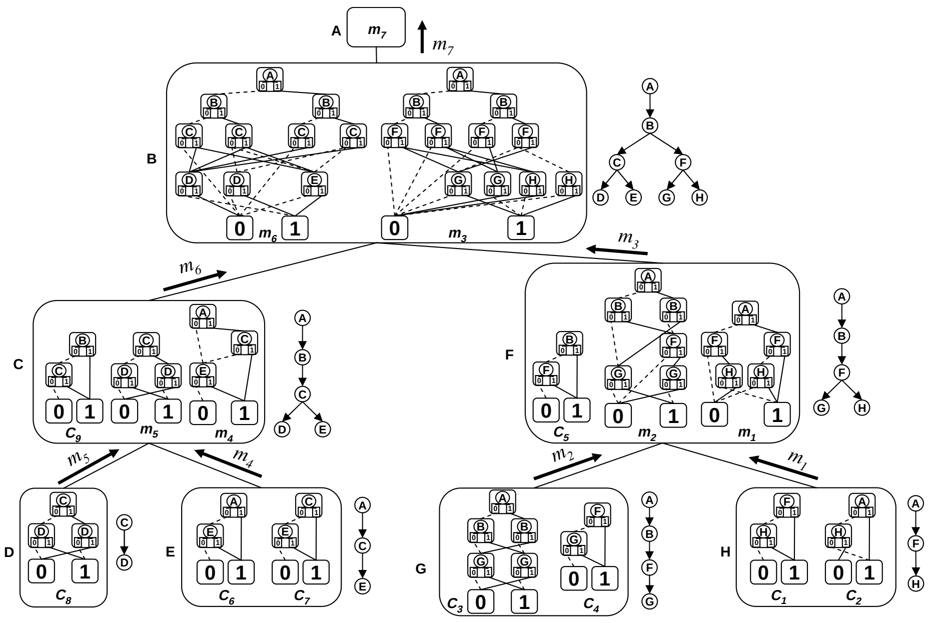
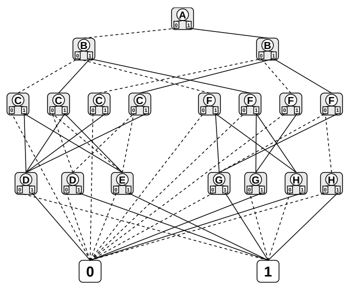


Fig. 4. Execution of VE with AOMDDs

in the bucket of their latest variable in d. Each *original* constraint is represented by an AOMDD based on a chain. For bi-valued variables, they are OBDDs, for multiple-valued they are MDDs. Note that we depict meta-nodes: one OR node and its two AND children, that appear inside each gray node. The dotted edge corresponds to the 0 value (the *low* edge in OBDDs), the solid edge to the 1 value (the *high* edge). We have some redundancy in our notation, keeping both AND value nodes and arc-types (doted arcs from “0” and solid arcs from “1”).

The **VE** scheduling is used to process the buckets in reverse order of d. A bucket is processed by *joining* all the AOMDDs inside it, using the *apply* operator. However, the step of eliminating the bucket variable will be omitted because we want to generate the full AOMDD. In our example, the messages $m_1 = C_1 \bowtie C_2$ and $m_2 = C_3 \bowtie C_4$


(a)

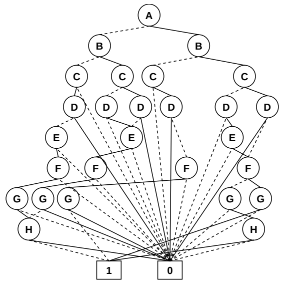

(b)

Fig. 5. (a) The final AOMDD; (b) The OBDD corresponding to d

are still based on chains, so they are still OBDDs. Note that they still contain the variables H and G, which have not been eliminated. However, the message $m_3 = C_5 \bowtie m_1 \bowtie m_2$ is not an OBDD anymore. We can see that it follows the structure of the pseudo tree, where F has two children, G and H. Some of the nodes corresponding to F have two outgoing edges for value 1.

The processing continues in the same manner The final output of the algorithm, which coincides with m_7, is shown in Figure 5a. The OBDD based on the same ordering d is shown in Fig. 5b. Notice that the AOMDD has 18 nonterminal nodes and 47 edges, while the OBDD has 27 nonterminal nodes and 54 edges.

4.1 Algorithm VE-AOMDD

Given an ordering d there is a unique pseudo tree $\mathcal{T}_d$ (or bucket tree) corresponding to it Each constraint C_i is compiled into an AOMDD that is compatible with $\mathcal{T}$ and placed into the appropriate bucket. The buckets are processed from last variable to first as usual. Each bucket contains AOMDDs that are either initial constraints or AOMDDs received from previously processed buckets. The scope of all the variables that are mentioned in a bucket include *relevant* variables, i.e. the ones whose buckets were not yet processed (note that they are identical to the OR context), and *superfluous* variables, the ones whose buckets had been proceessed. The number of relevant variables is bounded by the induced width (because so is the OR context). It is easy to see that all the AOMDDs in a bucket only have in common variables which are relevant, and which reside on the top chain portion of the bucket pseudo tree. The superfluous variables appear in disjoint branches of the bucket pseudo tree.

Consequently combining any two AOMDDs in a bucket amounts to using the regular MDD (OBDD) *apply* operator on their respective common parts that are simple MDDs. The rest of the branches can be attached at the end of the combine operation. Thus, the complexity of processing a bucket of AOMDDs is like the complexity of pair-wise MDD *apply* over constraints restricted to scopes bounded by the induced width.

Proposition 1. *The complexity of processing a bucket by* ***VE-AOMDD*** *is exponential in the number of relevant variables, therefore it is exponential in the induced width.*

In summary, to create the compiled AOMDD, we can use only regular MDD *apply* operators on the common (separator) portion in each bucket. In the next section however we will define a more general AOMDD *apply* operator that can combine any two AOMDDs whose pseudo trees are compatible (to be defined). This can be useful when the two input AOMDDs are created completely independently and we want to still exploit their given structure.

5 The AOMDD APPLY Operation

We will now describe how to combine two general AOMDDs. We focus on combination by join, as usual in constraint processing, but other operations can be equally easily applied. The apply operator takes as an input two AOMDDs representing constraints C_1 and C_2 and returns an AOMDD representing $C_1 \bowtie C_2$, respectively.

In traditional OBDDs the combining *apply* operator assumes that the two input OBDDs have compatible variable ordering. Likewise, in order to combine two AOMDDs, we assume that their backbone pseudo trees are *compatible*. Namely, there should be a pseudo tree in which both can be embedded. In general a pseudo tree induces a strict partial order between the variables where a parent node always precedes its child nodes.

Definition 14 (compatible pseudo tree). *A strict partial order $d_1 = (X, <_1)$ over a set X is consistent with a partial order $d_2 = (Y, <_2)$ over a set Y, if for all $x_1, x_2 \in X \cap Y$ and $x_1 <_2 x_2$ then $x_1 <_1 x_2$. Two partial orders d_1 and d_2 are compatible iff there exists a partial order d that is consistent with both. Two pseudo trees are compatible iff the partial orders induced via the parent-child relationship, are compatible.*

For simplicity, we focus on a more restricted notion of compatibility, which is sufficient when using a VE like schedule for the *apply* operator to combine the input AOMDDs. It is easy to extend this operator to the more general notion of compatibility.

Definition 15 (strictly compatible pseudo trees). *A pseudo tree $\mathcal{T}_1$ having the set of nodes $\mathbf{X}_1$ can be* embedded *in a pseudo tree $\mathcal{T}$ having the set of nodes $\mathbf{X}$ if $\mathbf{X}_1 \subseteq \mathbf{X}$ and $\mathcal{T}_1$ can be obtained from $\mathcal{T}$ by deleting each node in $\mathbf{X} \setminus \mathbf{X}_1$ and connecting its parent to each of its descendents. Two pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are* compatible *if there exists $\mathcal{T}$ such that both $\mathcal{T}_1$ and $\mathcal{T}_2$ can be embedded in $\mathcal{T}$.*

Algorithm 1, called APPLY, takes as input two AOMDDs for two constraints f and g defined along strictly compatible pseudo trees, and a common target pseudo tree $\mathcal{T}$. We will start by describing the intuition behind the algorithm. An AOMDD along a pseudo tree can be regarded as a union of regular MDDs, each restricted to a full path from root to a leaf in the pseudo tree. Let $\pi_{\mathcal{T}}$ be a path in $\mathcal{T}$. Based on the definition of strictly compatible pseudo trees, $\pi_{\mathcal{T}}$ has a corresponding path in $\mathcal{T}_f$ and $\mathcal{T}_g$, which are the pseudo trees for f and g. The MDDs from f and g corresponding to $\pi_{\mathcal{T}_f}$ and $\pi_{\mathcal{T}_g}$ can be combined using the regular MDD *apply*. This process can be repeated for every path $\pi_{\mathcal{T}}$. The resulting MDDs, one for each path in $\mathcal{T}$ need to be synchronized by another MDD *apply* on their common parts (on the intersection of the paths). The algorithm we propose does all this processing at once, in a depth first search traversal over the inputs. Based on our construction it is clear that the complexity of AOMDD-apply is governed by the complexity of MDD-apply.

Algorithm 1. APPLY($v_1; w_1, \ldots, w_m$)

```
input  : AOMDDs $f$ with nodes $v_i$ and $g$ with nodes $w_j$, based on compatible pseudo
         trees $\mathcal{T}_1$, $\mathcal{T}_2$ that can be embedded in $\mathcal{T}$.
         $var(v_1)$ is an ancestor of all $var(w_1), \ldots, var(w_m)$ in $\mathcal{T}$.
         $var(w_i)$ and $var(w_j)$ are not in ancestor-descendant relation in $\mathcal{T}$.
output : AOMDD $v_1 \bowtie (w_1 \wedge \ldots \wedge w_m)$, based on $\mathcal{T}$.
1  if $H_1(v_1, w_1, \ldots, w_m) \neq null$ then return $H_1(v_1, w_1, \ldots, w_m)$;    // is in cache
2  if (any of $v_1, w_1, \ldots, w_m$ is $\mathbf{0}$) then return $\mathbf{0}$
3  if ($v_1 = \mathbf{1}$) then return $\mathbf{1}$
4  if ($m = 0$) then return $v_1$                                   // nothing to join
5  create new nonterminal meta-node $u$
6  $var(u) \leftarrow var(v_1)$ (call it $X_i$, with domain $D_i = \{x_1, \ldots, x_{k_i}\}$ )
7  for $j \leftarrow 1$ to $k_i$ do
8      $u.children_j \leftarrow \phi$          // children of the j-th AND node of $u$
9      if ( ($m = 1$) and ($var(v_1) = var(w_1) = X_i$) ) then
10         $tempChildren \leftarrow w_1.children_j$
11     else
12         $tempChildren \leftarrow \{w_1, \ldots, w_m\}$
13     group nodes from $v_1.children_j \cup tempChildren$ in several $\{v^1; w^1, \ldots, w^r\}$
14     for each $\{v^1; w^1, \ldots, w^r\}$ do
15         $y \leftarrow$ APPLY($v^1; w^1, \ldots, w^r$)
16         if ($y = \mathbf{0}$) then
17             $u.children_j \leftarrow \mathbf{0}$; break
18         else
19             $u.children_j \leftarrow u.children_j \cup \{y\}$
20     if ($u.children_1 = \ldots = u.children_{k_i}$) then                // redundancy
21         return $u.children_1$
22     if ($H_2(var(u), u.children_1, \ldots, u.children_{k_i}) \neq null$) then  // isomorphism
23         return $H_2(var(u), u.children_1, \ldots, u.children_{k_i})$
24 Let $H_1(v_1, w_1, \ldots, w_m) = u$                              // add $u$ to $H_1$
25 Let $H_2(var(u), u.children_1, \ldots, u.children_{k_i}) = u$         // add $u$ to $H_2$
26 return $u$
```

Theorem 4. *Let $\pi_1, \ldots \pi_l$ be the set of path in $\mathcal{T}$ enumerated from left to right and let $\mathcal{G}_f^i$ and $\mathcal{G}_g^i$ be the OBDDs restricted to path π_i, then the size of the output of AOBDD-apply is bounded by $\sum_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i| \leq n \cdot max_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i|$. The time complexity is also bounded by $\sum_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i| \leq n \cdot max_i |\mathcal{G}_f^i| \cdot |\mathcal{G}_g^i|$.*

Algorithm APPLY takes as input one node from f and a list of nodes from g. Initially, the node from f is the root of f, and the list of nodes from g is in fact just one node, the root of g. The list of nodes from g always has a special property: there is no node in it that can be the ancestor in $\mathcal{T}$ of another (we refer to the variable of the meta-node). Therefore, the list $w_1, \ldots, w_m$ from g expresses a decomposition with respect to $\mathcal{T}$, so all those nodes appear on different branches. We will employ the usual techniques from OBDDs to make the operation efficient. First, since join can be viewed as multiplication, if one of the arguments is $\mathbf{0}$, then we can safely return $\mathbf{0}$. Second, a hash table H_1 is used to store the nodes that have already been processed, based on the nodes $(v_1, w_1, \ldots, w_r)$. Therefore, we never need to make multiple recursive calls on the same arguments. Third, a hash table H_2 is used to detect isomorphic nodes. If at the end of the recursion, before returning a value, we discover that a meta-node with the same variable and the same children had already been created, then we don't need to store it and we simply return the existing node. And fourth, if at the end of the recursion we discover we created a redundant node (all children are the same), then we don't store it, and return instead one of its identical lists of children.

Note that v_1 is always an ancestor of all $w_1, \ldots, w_m$ in $\mathcal{T}$. We consider a variable in $\mathcal{T}$ to be an ancestor of itself. A few self explaining checks are performed in lines 1-4. Line 2 is specific for multiplication, and needs to be changed for other operations. The algorithm creates a new meta-node u, whose variable is $var(v_1) = X_i$ - recall that $var(v_1)$ is highest (closest to root) in $\mathcal{T}$ among $v_1, w_1, \ldots, w_m$. Then, for each possible value of X_i, line 7, it starts building its list of children.

One of the important steps happens in line 13. There are two lists of meta-nodes, one from each original AOMDD f and g, and we will refer only to their variables, as they appear in $\mathcal{T}$. Each of these lists has the important property mentioned above, that its nodes are not ancestors of each other. The union of the two lists is grouped into maximal sets of nodes, such that the highest node in each set is an ancestor of all the others. It follows that the root node in each set belongs to one of the original AOMDD, say v^1 is from f, and the others, say $w^1, \ldots, w^r$ are from g. As an example, suppose $\mathcal{T}$ is the pseudo tree from Fig. 3b, and the two lists are $\{C, G, H\}$ from f and $\{E, F\}$ from g. The grouping from line 13 will create $\{C; E\}$ and $\{F; G, H\}$. Sometimes, it may be the case that a newly created group contains only one node. This means there is nothing more to join in recursive calls, so the algorithm will return, via line 4, the single node. From there on, only one of the input AOMDDs is traversed, and this is important for the complexity of APPLY, discussed below.

5.1 Complexity of APPLY

Given AOMDDs f and g, based on compatible pseudo trees $\mathcal{T}_1$ and $\mathcal{T}_2$ and the common pseudo tree $\mathcal{T}$, we define the *intersection pseudo tree* $\mathcal{T}_\cap$ as being obtained from $\mathcal{T}$ by marking all the subtrees whose nodes belong to either $\mathcal{T}_1$ or $\mathcal{T}_2$ but not to both, and removing them simultaneously (not recursively). The part of AOMDD f corresponding to the variables in $\mathcal{T}_\cap$ is denoted by $\mathcal{G}_{f_\cap}$.

Proposition 2. *The time complexity of* APPLY *and the size of the output are* $O(|\mathcal{G}_{f_\cap}| * |\mathcal{G}_{g_\cap}| + |\mathcal{G}_f| + |\mathcal{G}_g|)$.

Proof (sketch). The proof that APPLY makes an effort $O(|\mathcal{G}_{f_\cap}| * |\mathcal{G}_{g_\cap}|)$ when combining nodes from $f_\cap$ and $g_\cap$ is identical to the proof that OBDDs have complexity of the order of the product of the inputs. For the parts of f and g that don't belong to $\mathcal{G}_{f_\cap}$ and $\mathcal{G}_{g_\cap}$, APPLY generates recursive calls that have only one argument, v_1, effectively calling for a simple traversal of just one of the inputs.

6 Compiling Any Probabilistic Model to AOMDD

As we showed in the past, the notion of AND/OR graphs is applicable to any graphical models, such as probabilistic networks, cost networks and influence diagrams. Indeed, the compiled canonical forms of *minimal AND/OR graphs* are well defined and were already introduced [1,4]. Therefore all the ideas we introduced in this paper can be generalized, yielding a canonical representation in the style of AOMDD (i.e., by removing redundant variables), which we can term *weighted AOMDDs*. In particular, weighted AOMDD can be compiled using Variable Elimination schedule that exploits an *apply operator* in each bucket, very similar to the way we carried this task here. The apply operator by itself, combining two weighted AOMDDS should have the same flavor. Compiling graphical models into weighted AOMDDs also extends the work of [17], which defines decision diagrams for the computation of semiring valuations, from linear variable ordering into tree-based partial ordering.

7 Related Work

There are various lines of related research. The formal verification literature, beginning with [7] contains a very large number of papers dedicated to the study of BDDs. However, BDDs are in fact OR structures (the underlying pseudo tree is a chain) and do not take advantage of the problem decomposition in an explicit way. The complexity bounds for OBDDs are based on *pathwidth* rather than *treewidth*.

As noted earlier, the work on Disjoint Support Decomposition (DSD) is related to AND/OR BDDs in various ways [9]. The main common aspect is that both approaches show how structure decomposition can be exploited in a BDD-like representation. DSD is focused on Boolean functions and can exploit more refined structural information that is inherent to Boolean functions. In contrast, AND/OR BDDs assumes only the structure conveyed in the constraint graph, they are therefore more broadly applicable to any constraint expression and also to graphical models in general. They allow a simpler and higher level exposition that yields graph-based bounds on the overall size of the generated AOMDD. The full relationship between these two formalisms should be studied further.

McMillan introduced the BDD trees [10], along with the operations for combining them. For circuits of bounded tree width, BDD trees have linear upper space bound $O(|g|2^{w2^{2w}})$, where $|g|$ is the size of the circuit g (typically linear in the number of variables) and w is the treewidth. This bound hides some very large constants to claim the linear dependence on $|g|$ when w is bounded. However, McMillan maintains that when the input function is a CNF expression BDD-trees have the same bounds as AND/OR BDDs, namely they are exponential in the treewidth only.

Darwiche has done much research on compilation, using insights from the AI community. The AND/OR structure restricted to propositional theories is very similar to deterministic decomposable negation normal form (d-DNNF) [18]. More recently, in [19], the trace of the DPLL algorithm is used to generate an OBDD, and compared with the bottom up approach of combining the OBDDs of the input function according to some schedule (as is typical in formal verification). The structures that are investigated are still OR. The idea can nevertheless be extended to AND/OR search. We could run the depth first AND/OR search with caching, generating the *context minimal* AND/OR graph, which can then be processed bottom up by layers to be reduced even further by eliminating isomorphic subgraphs and redundant nodes.

McAllester [20] introduced the case factor diagrams (CFD) which subsume Markov random fields of bounded tree width and probabilistic context free grammars (PCFG). CFDs are very much related to the AND/OR graphs. The CFDs target the minimal representation, by exploiting decomposition (similar to AND nodes) but also by exploiting context sensitive information and allowing dynamic ordering of variables based on context. CFDs do not eliminate the redundant nodes, and part of the cause is that they use zero suppression. There is no claim about CFDs being a canonical form, and also there is no description of how to combine two CFDs.

More recently, independently and in parallel to our work on AND/OR graphs [1,2], Fargier and Vilarem [11] proposed the compilation of CSPs into tree-driven automata, which have many similarities to our work. In particular, the compiled tree-automata proposed there is essentially the same as what we propose here. Their main focus is the transition from linear automata to tree automata (similar to that from OR to AND/OR), and the possible savings for tree-structured networks and hyper-trees of constraints due to decomposition. Their compilation approach is guided by a tree-decomposition while ours is guided by a variable-elimination based algorithms. And, it is well known that Variable Elimination and cluster-tree decomposition are in principle, the same [21].

We see that our work using AND/OR search graphs has a unifying quality that helps make connections among seemingly different compilation approaches.

8 Conclusion

We propose the AND/OR multi-valued decision diagram (AOMDD), which emerges from the study of AND/OR search for graphical models [1,2,3] and ordered binary decision diagrams (OBDDs) [7]. This data-structure can be used to compile any set of relations over multi-valued variables as well as any CNF Boolean expression.

The approach we take in this paper may seem to go against the current trend in model checking, which moves away from BDD-based algorithms into CSP/SAT based approaches. However, constraint processing algorithms that are search-based and compiled data-structures such as BDDs differ primarily by their choices of time vs memory. When we move from regular OR search space to an AND/OR search space the spectrum of algorithms available is improved for all time vs memory decisions. We believe that the AND/OR search space clarifies the available choices and helps guide the user into making an informed selection of the algorithm that would fit best the particular query asked, the specific input function and the available computational resources.

In summary, the contribution of our work is: (1) We formally describe the AOMDD and prove that it is a canonical representation of a constraint network; (2) We describe the APPLY operator that combines two AOMDDs by an operation and give its complexity bounded by the product of the sizes of the inputs; (3) We give a scheduling of building the AOMDD of a constraint network starting with the AOMDDs of its constraints. It is based on an ordering of variables, which gives rise to a pseudo tree (or bucket tree) according to the execution of Variable Elimination algorithm. This gives the complexity guarantees in terms of the *induced width* along the ordering (equal to the treewidth of the corresponding decomposition); 4) We show how AOMDDs relate to various earlier and recent works, providing a unifying perspective for all these methods.

Acknowledgments

This work was supported in part by the NSF grant IIS-0412854. The initial part of this work was also supported by the Radcliffe fellowship 2005-2006, as part of the partner program, with Harvard undergraduate student John Cobb [22].

References

1. Dechter, R., Mateescu, R.: Mixtures of deterministic-probabilistic networks and their and/or search space. In: UAI'04. (2004) 120–129
2. Dechter, R., Mateescu, R.: The impact of and/or search spaces on constraint satisfaction and counting. In: CP'04. (2004) 731–736
3. Mateescu, R., Dechter, R.: The relationship between and/or search and variable elimination. In: UAI'05. (2005) 380–387
4. Dechter, R., R.Mateescu: And/or search spaces for graphical models. Artificial Intelligence (2006) forthcoming.
5. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
6. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic (1993)
7. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transaction on Computers **35** (1986) 677–691
8. Brayton, R., McMullen, C.: The decomposition and factorization of boolean expressions. In: ISCAS,Proceedings of the International Symposium on Circuits and Systems. (1982) 49–54
9. Bertacco, V., Damiani, M.: The disjunctive decomposition of logic functions. In: ICCAD, International Conference on Computer-Aided Design. (1997) 78–82
10. McMillan, K.L.: Hierarchical representation of discrete functions with application to model checking. In: Computer Aided Verification. (1994) 41–54
11. Fargier, H., Vilarem, M.: Compiling csps into tree-driven automata for interactive solving. Constraints **9**(4) (2004) 263–287
12. Freuder, E.C., Quinn, M.J.: Taking advantage of stable sets of variables in constraint satisfaction problems. In: IJCAI'85. (1985) 1076–1078
13. Bertele, U., Brioschi, F.: Nonserial Dynamic Programming. Academic Press (1972)
14. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113** (1999) 41–85
15. Bayardo, R., Miranker, D.: A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In: AAAI'96. (1996) 298–304
16. Darwiche, A.: Recursive conditioning. Artificial Intelligence **125**(1-2) (2001) 5–41
17. Wilson, N.: Decision diagrams for the computation of semiring valuations. In: IJCAI'05. (2005) 331–336
18. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research (JAIR) **17** (2002) 229–264
19. Huang, J., Darwiche, A.: Dpll with a trace: From sat to knowledge compilation. In: IJCAI'05. (2005) 156–162
20. McAllester, D., Collins, M., Pereira, F.: Case-factor diagrams for structured probabilistic modeling. In: UAI'04. (2004) 382–391
21. Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence **38** (1989) 353–366
22. Cobb, J.: Joining and/or networks. In: Report, Radcliffe fellowship, Cambridge, MA. (2006)

Distributed Constraint-Based Local Search

Laurent Michel[1], Andrew See[1], and Pascal Van Hentenryck[2]

[1] University of Connecticut, Storrs, CT 06269-2155
[2] Brown University, Box 1910, Providence, RI 02912

Abstract. Distributed computing is increasingly important at a time when the doubling of the number of transistors on a processor every 18 months no longer translates in a doubling of *speed* but instead a doubling of the number of cores. Unfortunately, it also places significant conceptual and implementation burden on programmers. This paper aims at addressing this challenge for constraint-based local search (CBLS), whose search procedures typically exhibit inherent parallelism stemming from multistart, restart, or population-based techniques whose benefits have been demonstrated both experimentally and theoretically. The paper presents abstractions that allows distributed CBLS programs to be close to their sequential and parallel counterparts, keeping the conceptual and implementation overhead of distributed computing minimal. A preliminary implementation in Comet exhibits significant speed-ups in constraint satisfaction and optimization applications. The implementation also scales well with the number of machines. Of particular interest is the observation that generic abstractions of CBLS and CP, such as models and solutions, and advanced control structures such as events and closures, play a fundamental role to keep the distance between sequential and distributed CBLS programs small. As a result, the abstractions directly apply to CP programs using multistarts or restarts procedures.

1 Introduction

Moore's law [13], i.e., the prediction that the number of transistors per square inch on integrated circuits would double every 18 months used to translate into a doubling of speed. While it marches on, these additional transistors are now devoted to doubling the number of *cores* and gave rise to commodity multiprocessors. As a result, parallel and distributed computing now offer reasonably cheap alternatives to speed up computationally intense applications. However, parallel and distributed computing also places significant conceptual and implementation burden on programmers. The computational model adds another dimension in conceptual complexity (i.e., the need to handle multiple threads of executions) and programming abstractions are often expressed at a lower level of abstraction than their sequential counterparts. This has slowed the use of distributed computing, even for applications that exhibit natural parallelism as is typically the case in constraint satisfaction and optimization.

The parallelism exhibited in constraint satisfaction and optimization is often coarse-grained, requires minimal synchronization and coordination, and may originate from restart, multistart, and population techniques, whose benefits have been demonstrated both experimentally and theoretically (e.g., [10, 6, 16, 7, 9]). Indeed, task durations are far more uniform and predictable than those associated to search nodes produced by traditional CP solvers. Yet very few implementations actually exploit this inherent potential: it suffices to look over the experimental results published in constraint programming conferences to realize this. The main reason is the absence of high-level abstractions for distributed computing that makes distributed programs substantially different from their sequential counterparts, even for applications that should be naturally amenable to distributed implementations.

This paper originates as an attempt to address this challenge for constraint-based local search (CBLS) and constraint programming (CP) applications which use multistart, restart, or population-based techniques. It presents abstractions that allows distributed CBLS or CP programs to be close to their sequential and parallel counterparts, keeping the conceptual and implementation overhead of distributed computing minimal. The abstractions naturally generalize their parallel counterparts [12] to a distributed setting: they include distributed loops, interruptions, and model pools, as well as shared objects. The resulting distributed programs closely resemble their parallel counterparts which are themselves close to the sequential implementations.

A preliminary implementation of the abstractions in COMET (using, among others, sockets, forks, and TCP) exhibits significant speedups on constraint satisfaction (e.g., Golomb rulers) and optimization (e.g., graph coloring) applications when parallelizing effective sequential programs. The implementation is shown to scale well with the number of machines, even when the pool of machines is heterogeneous (e.g., the machines have different processor frequencies and cache sizes). Together with the simplicity of the resulting CBLS programs, these results indicate that the distributed abstractions offer significant benefits for practitioners at a time when the need for large-scale constraint satisfaction and optimization or fast response time is steadily increasing.

It is also important to emphasize that the abstractions result from the synergy between recent modeling abstractions from CBLS and CP, such as the concepts of models and solutions [11,8], the novel distributed abstractions presented herein, and advanced control structures such as events and closures [19].

The rest of the paper is organized as follows. Section 2 presents the novel abstractions. Section 3 introduces two language extensions, processes and shared objects, that are fundamental in implementing the abstractions. Section 4 sketches the implementation. Section 5 discusses related work, while Section 6 reports the experimental results and concludes the paper.

2 Distributed Constraint-Based Local Search

This section reviews the distributed abstractions of COMET. The main theme is to show that the distance between sequential and distributed COMET programs is small, making distributed computing far more accessible for CBLS than existing

```
                                    0. string[] macs = ["m1","m2","m3"];
1. ThreadPool tp(3);                1. MachinePool tp(macs);
2. SolutionPool S();                2. shared{SolutionPool} S();
3. parall<tp>(i in 1..nbStarts) {   3. parall<tp>(i in 1..nbStarts) {
4.   WarehouseLocation location();  4.   WarehouseLocation location();
5.   location.state();              5.   location.state();
6.   S.add(location.search());      6.   S.add(location.search());
7. }                                7. }
8. tp.close();                      8. tp.close();
9. cout << S.getBest().getValue();  9. cout << S.getBest().getValue();
```

Fig. 1. From Parallel to Distributed Multistart Constraint-Based Local Search

libraries such as PVM and MPI. To demonstrate this significant benefit, the paper contrasts the parallel and distributed applications in Comet, since the parallel abstractions designed for shared memory multi-processors were shown to allow a small distance between sequential and parallel code [12].

Parallel Iterations. The main parallel and distributed abstraction of Comet is the concept of parallel loops. Figure 1 depicts how to move from a parallel to a distributed implementation of a mutistart CBLS for warehouse location. The CBLS for warehouse location was described in [18]: it is organized here as a *model* providing methods to state its constraints and objectives, and to search for a (near-optimal) *solution*. The left part of the figure depicts the parallel implementation, while the right part of the figure exhibits the distributed version.

The parallel implementation declares a thread pool consisting of 3 threads (line 1) and a solution pool to collect the solutions of multiple runs (line 2). The parallel loop is shown in line 3: it is the equivalent of a for-loop but it uses the thread pool to dispatch the iterations of the body to the threads in the pool. The body creates the warehouse model, states its constraints and objectives, searches for a solution, and adds the solution to the solution pool. The various iterations are synchronized in line 8, which then closes the thread pool. The objective value of the best found solution is displayed in line 9.

The distributed implementation is almost identical to the parallel implementation. Instead of thread pools, it uses a machine pool specifying which machines to use for the parallel loop (line 0–1, where the names of the machines are “m1”,“m2”, and “m3”). The solution pool is now *shared*, meaning that processes on the different machines may access it in a synchronized and distributed fashion. The rest of the code is identical to the parallel code, although the loop iterations will now execute on machines “m1”,“m2”, and “m3”.

The simplicity of the implementation is partly due to the advanced control structures of Comet and partly to optimization concepts such as models and solutions. These concepts are recent innovations in CBLS [11] and CP [8] and are fundamental in that they allow for a clean separation between the specificities of the models and parallel and distributed abstractions. Note also that the warehouse model could be implemented as a CBLS algorithm or a randomized branch

```
1.   Boolean found(false);            1.   Boolean found(false);
2.   parall<p>(i in 1..nbStarts) {    2.   parever<p> {
3.     ProgressiveParty pp();         3.     ProgressiveParty pp();
4.     pp.state();                    4.     pp.state();
5.     Solution s = pp.search();      5.     Solution s = pp.search();
6.     found := (s.getValue() == 0);  6.     found := (s.getValue() == 0);
7.   } until found;                   7.   } until found;
```

Fig. 2. Distributed Interruptions of Constraint-Based Local Search

```
                                     0.   string[] macs = ["m1","m2","m3"];
1.   WarehouseLocationFactory f();   1.   WarehouseLocationFactory f();
2.   ModelPool mp(3,f);              2.   DistrModelPool mp(macs,f);
3.   SolutionPool S();               3.   shared{SolutionPool} S();
4.   parall<mp>(i in 1..nbStarts)    4.   parall<mp>(i in 1..nbStarts)
5.     S.add(mp.search());           5.     S.add(mp.search());
6.   mp.close();                     6.   mp.close();
```

Fig. 3. From Parallel to Distributed Model Pools

and bound algorithm with a computation limit. The distributed abstractions are independent of the underlying optimization technology.

Interruptions. In constraint satisfaction, the goal consists in finding a feasible solution. Random restarts or multiple random searches have been shown to be a fundamental ingredient of CBLS and CP algorithms [10, 6]. Such algorithms are inherently parallel but an implementation must stop as soon as a solution has been found. Figure 2 depicts part of the code for a parallel and a distributed multistart algorithm for the progressive party problem.

Consider first the distributed implementation depicted in the left part of the figure. Line 1 declares a Boolean that specifies whether a solution was found and the parallel loop now terminates as soon as the Boolean becomes true (line 6) or when the fixed number of iterations is exhausted. The code (line 1–6) executes with either a thread or a machine pool. In the distributed implementation, different machines execute the model and are interrupted as soon as a feasible solution is found on another machine. Observe once again how easy it is to move from a parallel to a distributed implementation: it suffices to replace a thread pool by a machine pool. Consider now the right part of Figure 2. Here the code uses the `parever` instruction which iterates its body until a solution is found. Once again, the code is identical for the parallel and distributed implementation.

It is important to emphasize that the constraint satisfaction was not modified at all to be amenable to a distributed implementation or to allow for interruptions. This clean separation of concerns is one of the main advantages of the distributed abstractions presented herein. They naturally leverage sequential models and automate tedious aspects of distributed computing. Once again, the progressive party implementation can be either a CBLS algorithm or a randomized constraint program with a computation limit.

```
interface Model {
  void state();
  Solution search();
  Solution search(Solution s);
  Solution search(Solution s1,Solution s2);
  Solution search(Solution[] s);
}
interface ModelFactory { Model create(); }
```

Fig. 4. The Model and Model Factory Interfaces

```
found := false;
while (!found) {
  Solution op[k in RP] = pop[k];
  parall<mp>(k in RP) {
    select(i in RP, j in RP: i != j) {
      pop[k] = mp.search(op[i],op[j]);
      if (pop[k].getValue() == 0)
        found := true;
  } until found;
}
```

Fig. 5. The Core of the Distributed Implementation for Finding Golomb Rulers

Model Pools. One of the limitations of thread and machine pools is the necessity of creating and stating models multiple times. The concept of model pool was introduced in [12] to remedy this limitation by leveraging the concept of models and it naturally generalizes to a distributed setting. A model pool receives, as parameters, a factory to create models and the number of models to create for parallel execution. When a solution is required, the model pool retrieves, or waits for, an idle model and searches for a solution inside a new thread. Distributed model pools simply receive the names of the machines instead of the number of models. They also retrieve an idle model and search for a solution on one of the idle machines.

Figure 3 illustrates model pools on the warehouse location problem again. Consider first the left part of the figure that describes the parallel implementation. The COMET code first declares a factory to create the warehouse location models (line 1), a model pool (line 2), and a solution pool (line 3). The parallel loop simply asks the model pool for solutions (method `mp.search()`) and stores them in the solution pool. Consider now the right part of the figure: it is almost identical but uses distributed model pools.

For completeness, Figure 4 specifies the interface for models and model factories. A model provides methods to state its constraints and objectives and to search for solutions. The search for solutions may receive zero, one, or more solutions as starting points, allowing for a variety of search algorithms. A model factory simply creates models, which may or may not use different search procedures. Once again, the concept of models and solutions, fundamental abstractions in CBLS and CP, are critical in moving naturally from a sequential to a distributed implementation.

Finally, Figure 5 depicts the core of an evolutionary CBLS for finding Golomb rulers. The algorithm, used in our experimental results, integrates many of the abstractions just presented: parallel loops, interruptions, and model pools. The figure depicts the part of the COMET program searching for a Golomb whose length is smaller than a given number. In the full program, this core is embedded in an outer loop that searches for rulers of smaller and smaller lengths. The core of the algorithm maintain a population `pop` of (infeasible) rulers and generates new generations of the population until a solution is found (lines 4–9). To generate a new ruler, the algorithm selects two individuals in the population (line 5), crosses them, and applies a CBLS minimizing the number of violations to generate a new ruler (line 6). The distributed implementation uses a distributed model pool and generates the new population in parallel on different machines. Each such distributed computation is interrupted as soon as a feasible ruler is found, i.e., a ruler with no violations (lines 7–8). Observe once again the simplicity of the distributed implementation that closely resembles its sequential counterpart and is almost identical to the parallel implementation, since it only replaces model pools by distributed model pools.

3 Enabling Technology

To support the abstractions presented above, only two language extensions are required: processes and shared objects. This section briefly reviews them.

Processes. COMET features a `process` construct to fork a new process on a specific machine. For instance, the code

```
Queens q(1024);
SolutionPool S();
process("bohr") {
  S.add(q.search());
}
```

forks a new COMET process on machine `bohr` to find a solution to the 1024-queens problem. When a COMET process is created, it is initialized with a copy of its parent's runtime. The child process starts executing the body of the `process` instruction, while the parent continues in sequence. There is an implicit rendezvous when the parent is about to terminate. Observe that, since the child has its own copy of the runtime, operations performed in the child are only visible on its own runtime and do not affect the parent's objects and data structures. In particular, in the above code, the solution to the queens problem is only available in the child, the parent's model and its solution pool `S` being left unchanged.

Shared Annotations. The second abstraction was already mentioned earlier. To allow distributed algorithms to communicate naturally, COMET features the concept of shared objects that are visible across processes. Consider the code.

```
1.  Queens q(1024);
2.  shared{SolutionPool} S();
3.  process("bohr") {
4.    S.add(q.search());
5.  }
```

The solution pool S is now a shared object that lives in the runtime of the parent but is visible in the runtime of the child. As a result, when line 4 is executed, the solution is added into the parent's pool. The child in fact does not have a solution pool, simply a reference to the parent's pool since the object is shared. Shared objects allow processes to communicate naturally by (remote) method invocations very much like systems such as CORBA and MPI. The linguistic overhead in Comet is really minimal however, keeping the distance between sequential and distributed programs small.

It is important to connect shared objects and process creation. When a process is created, the parent's runtime is cloned, except for shared objects that are replaced by proxy objects with the same interface. These proxy objects encapsulate a remote reference to the original object and transform method invocations into remote method invocations using traditional serialization techniques.

Events on shared objects are also supported. A child process may subscribe to events of shared objects and be notified when these events occur. This functionality, which is highly convenient for implementing interruptions, allows this fundamental control abstraction of Comet [19] to be used transparently across threads and processes.

4 Implementation

This section sketches the implementation of the distributed abstractions which consists of three parts:

1. source to source transformations of the Comet programs to replace parallel loops by traditional loops, closures, and barriers;
2. machine and model pools which receive closures in input and creates processes to execute them on remote machines;
3. processes and shared objects which are now integral parts of the Comet runtime system.

4.1 Source to Source Transformation

The first step of the implementation consists of replacing parallel loops by sequential loops and barriers. This step is parameterized by the pool and thus identical in the parallel and distributed implementations. It is illustrated in Figure 6 for warehouse location. The left of the figure shows the original Comet program, while the right part depicts the transformed program. The transformed program declares a shared barrier (line 1) to synchronize the loop executions. The barrier is initialized to zero, incremented for each iteration (line 3), and decremented when an iteration is completed (line 8). The barrier is used in line 12 to ensure that the main thread continues in sequence only when all the

```
                                       1.  ZeroWait b(0);
1.  parall<mp>(i in 1..nbStarts) {     2.  forall(i in 1..nbStarts) {
                                       3.    b.incr();
                                       4.    closure C {
2.    WarehouseLocation location();    5.      WarehouseLocation location();
3.    location.state();                6.      location.state();
4.    S.add(location.search());        7.      S.add(location.search());
                                       8.      b.decr();
                                       9.    }
                                       10.   mp.submit(C);
5.  }                                  11. }
                                       12. b.wait();
6.  mp.close();                        13. mp.close();
```

Fig. 6. Source to Source Transformation for Parallel Loops

iterations are completed. The parallel loop is replaced by a sequential loop (line 2), which creates a closure `C` which is then submitted to the pool. The thread executing the code thus simply forwards the closures, one for each iteration, to the pool where the parallel or distributed execution takes place.

Figure 7 generalizes the implementation to support interruptions. It shows the source to source transformation for the progressive party problem, the only differences being in line 5–8. The transformation creates an event-handler that calls method `terminate` on the pool whenever the Boolean `found` changes value (and becomes true), interrupting all the threads or processes in the pool. In addition, the pool now tests whether the Boolean is false before executing the body of the iteration. Once again, this transformation is valid for all parallel pools. This genericity is possible because all pools must implements the same `ParallelPool` interface depicted in Figure 8.

4.2 Machine and Distributed Model Pools

As mentioned earlier, the main thread only dispatches the closures, one for each iteration of the loop, to the parallel pool. It is thus the role of the parallel pools to execute these closures in parallel or in a distributed fashion. Figure 9 depicts the implementation of the machine pool.

Machine Pools. The machine pool uses two producer/consumer buffers: one for the machines and one for the closures to execute (line 2). It uses a Boolean `cont` to determine if the pool should continue execution (line 2), and an object `interrupt` to handle interruption (line 3). The closures are produced by the execution of the parallel loop as shown in the source to source transformation. The resulting code (see Figure 6) calls method `submit` which produces a closure.

The core of the machine pool is in its constructor (lines 4–22). It creates the buffers (line 6–7), the interrupt handler (line 8), and produces all the machines in line 9. Observe that the machine buffer is shared, since upon termination processes must release their host and thus must access the buffer `macs` remotely.

```
                                      1.  ZeroWait b(0);
1.  parall<mp>(i in 1..nbStarts) {    2.  forall(i in 1..nbStarts) {
                                      3.    b.incr();
                                      4.    closure C {
                                      5.      when found@changes()
                                      6.        mp.terminate();
                                      7.      in {
                                      8.        if (!found) {
2.    ProgressiveParty pp();          9.          ProgressiveParty pp();
3.    pp.state();                     10.         pp.state();
4.    Solution s = pp.search();       11.         Solution s = pp.search();
5.    found := (s.getValue() == 0);   12.         found:=(s.getValue()==0);
                                      13.       }
                                      14.     }
                                      15.     b.decr();
                                      16.   }
                                      17.   mp.submit(C);
6.  } until found;                    18. }
                                      19. b.wait();
7.  mp.close();                       19. mp.close();
```

Fig. 7. Source to Source Transformation for Parallel Loops with Interruptions

```
interface ParallelPool {
  void submit(Closure c);
  void terminate();
  void close();
  int getSize();
}
```

Fig. 8. `ParallelPool` interface

The distributed computing code lies in line 10–21. It creates a thread responsible for dispatching closures to the various machines as long as the machine pool must execute. The essence of the implementation is in lines 12–19 that are executed at each iteration. First, the implementation consumes a machine or waits until such a machine is available (i.e., the buffer `macs` contains an available host) (line 12). Once a machine is obtained, the implementation consumes a closure to execute or waits for a closure to become available (line 13). If the machine or the closure are `null`, then the model pool has terminated and execution completes. Otherwise, the implementation creates a process on the available machine, which executes the closure (line 17) and then returns the machine to the buffer. The closure execution may be interrupted, which happens if the closure calls method `terminate` (see line 6 in the right part of Figure 7). The `break` implementation was discussed in [12] and uses events with either exceptions or continuations. Finally, the `Interrupt` class is depicted in lines 27-31 and simply encapsulates an event. When method `terminate` on the machine pool is called, the event `raised` is notified and the closure call in line 17 is interrupted.

```
1.  class MachinePool {
2.    StringBuffer macs;ClosureBuffer closures;Boolean cont;
3.    Interrupt interrupt;
4.    MachinePool(string[] mac) {
5.      cont = new Boolean(true);
6.      closures = new ClosureBuffer(mac.rng().sz());
7.      macs = new shared{StringBuffer}(mac.rng().sz());
8.      interrupt = new shared{Interrupt}();
9.      forall(i in mac.rng()) macs.produce(mac[i]);
10.     thread {
11.       while (cont) {
12.         string m = macs.consume();
13.         Closure v = closures.consume();
14.         if (m != null && v!=null)
15.           process(m) {
16.             break when interrupt@raised()
17.               call(v);
18.             macs.produce(m);
19.           }
20.       }
21.     }
22.   }
23.   void submit(Closure v) { closures.produce(v); }
24.   void close() {cont := false;macs.terminate();closures.terminate();}
25.   void terminate() { interrupt.raise(); close(); }
26. }
27. class Interrupt {
28.   Event raised();
29.   Interrupt() {}
30.   void raise() { notify raised();}
31  }
```

Fig. 9. The Machine Pool Implementation

It is interesting to trace the steps involved in executing the progressive party code in Figure 7. First, observe that the closure `v` in line 13 consists of the body of the loop and resides in the parent process. When the child is created, it inherits a copy of that closure and executes it locally. It thus creates the progressive party object (line 9), states the constraints (line 10), and searches for a solution on the remote machine (line 11). Second, if the search finds a solution (line 12), it assigns the Boolean to true. This Boolean also resides in the parent and was copied into the child. Since it is not shared, only the child copy is affected. The code in line 6 of the right-hand side Figure 7 executes and calls `terminate` on the machine pool to raise an exception caught in line 16 of the other processes interrupting their execution of line 17 in Figure 9. This is possible as `interrupt` is shared and all processes subscribe to its `raised` event.

Distributed Model Pools. Distributed model pools are modeled after model pools. The implementation creates one process per machine and associates a unique model to it. The process then becomes a server, waiting for service requests.

4.3 The Runtime System

It remains to discuss how to implement the extensions to the runtime system.

Distributed Forks COMET processes have the same semantics as traditional `fork` system calls available in any modern operating system: they simply add the ability to spawn the child on a different host. The COMET implementation uses a small daemon process on all the machines allowed to host COMET processes. The daemon listens on a TCP port for process requests. When the COMET virtual machine creates a process, it contacts the daemon on the target machine. In response, the daemon forks itself and loads the COMET executable in the new child which uses the dynamic TCP port number for further exchanges. The parent then ships its runtime data structures (including the stack and the heap) to the child over the TCP connection, using traditional serialization and de-serialization. The runtime data structures are re-created at the exact same virtual memory addresses on the child process. The final step consists of replacing the shared objects by proxies. Each proxy then holds a reference to the network connection and the address of the parent process.

Shared Objects. Method invocation on shared objects is completely transparent. It is the role of the proxy to contact the owner process, serialize the arguments, and de-serialize the results. Shared objects may be used as arguments and are replaced by a proxy. The remaining parameters must be serializable, which is the case for fundamental abstractions such as solutions. On the receiver side, the remote method invocation is de-serialized. Since it contains the actual address of the receiver, the implementation performs a standard method dispatch on the de-serialized arguments. Since this receiver is shared, it is also a monitor, automatically synchronizing remote and local invocations.

Remote Events. Events can be handled with the same techniques. Since events are managed through a publish-subscribe model [19], a subscription on a remote object results in a message to the true receiver to notify the subscriber. When the event takes place, the remote object remotely publishes the event, inducing the subscriber to execute its closure locally.

5 Related Work

This section briefly reviews other initiatives in distributed computing.

Languages. OZ is a concurrent language that supports parallel and distributed computing. In [17], it has been used to implement a distributed search engine that implements a protocol and a search node distribution strategy for distributed DFS. Like COMET, it argues in favor of a strong separation of search from concurrency and distribution. In contrast to COMET, search spaces are not distributable structures and the communication protocol relies on a combination of search path and recomputation to ship search nodes to workers.

openMP [3, 2] is a preprocessor for parallel loops in C and Fortran. The parallel abstractions of Comet and openMP share the same motivations as both type of systems aim at making parallel computing widely accessible by reducing the distance between sequential and parallel code. However, as discussed in [12], the parallel abstractions of Comet are simpler and richer, primarily because of its advanced control abstractions. This paper goes one step further: it shows that the abstractions naturally generalize to distributed computing, opening a new realm of possibilities. FORTRESS [1] is a new language aimed at supporting high performance applications. To our knowledge, it is at the specification stage.

Libraries. Libraries for distributed computing focuses on various forms of message passing. The actual implementations can be realized at different levels of abstraction: sockets or messaging. Sockets are very low-level and are best viewed as an implementation technology. `MPI` and `PVM` impose a significant burden for optimization software that must be explicitly reorganized to match the client-server architecture. Parallel Solver [14] is a domain-specific solution to parallelize the exploration of complete search tree but it focuses on SMP systems only.

Object Models. `DCOM` and `CORBA` introduce an object model for distributed computing where remote method calls are performed transparently. However, both impose significant burden on programmers as discussed in [15]. Applications must be redesigned to fit a client-server model, object interfaces must be specified in a separate language (IDL) to generate proxies, and programmers are exposed to low-level threading and memory management issues. All these limitations are addressed by Comet's abstractions for distributed CBLS.

Distributed CSPs. distributed CSPs are formalized and the first distributed asynchronous backtracking search algorithm is introduced in [20]. DiCSPs take a fundamentally different approach as the set of variables and constraints are themselves distributed and is more directly related to agent-based searching.

6 Experimental Results

The Benchmarks. The benchmarks consist of a multistart version of the tabu-search algorithm for graph-coloring from [4] and the hybrid evolutionary algorithm for Golomb rulers mentioned earlier [5]. The coloring algorithm is an optimization application minimizing the number of colors. It was evaluated on the benchmarks `R250.5` (250 vertices, 50% density and best coloring=65) and `R250.1c` (250 vertices, 90% density and best coloring=64). Note that these problems have 250 variables and thousands of constraints, which makes them interesting since distributed implementations must copy the entire address space for each process. The Comet program for Golomb rulers is particularly interesting. At each iteration, it generates a new population of rulers, unless it finds a feasible ruler in which case the computation terminates. There is significant variance in how fast a feasible ruler is found (within and across generations) and hence it

Names	Proc.	Cache	Freq.	bogoMIPS	T(`R250.1c`)	T(`R250.5`)
m1	P4	1M	3.0	5898	95.2	96.33
m2	P4	1M	2.8	5570	109.7	
m3	P4	512K	2.4	4771	**194.3**	**150.59**
m4	Xeon	512K	2.4	4784	202.6	
m5	P4	512K	2.4	4757	210.4	
m6	P4	512K	2.0	3932	248.6	

Fig. 10. Specifications of the Machines

is interesting to assess the performance of the distributed implementation. The program was run until the optimum (of length 72) was found.

The Machines. The benchmarks were executed on an heterogeneous pool of machines all running Debian Linux. The machines differ, not only in their clock frequencies but also in the type of processors and the size of their caches. The features of the machines are depicted in Figure 10 which specifies the processor type, the cache size, the clock frequency, the bogoMIPS speed estimate of Linux, and the time to execute `R250.1c` and `R250.5` in deterministic mode. The boldfaced times for coloring on machine `m3` are used as reference in Figure 11. As can be seen, there are significant speed differences between the machines. The results use prefixes of the worst-case ordering for COMET: `m1,m2, ...,` `m6`; for instance, on four machines, the pool consists of machines `m1,...,m4`. Since the machines are increasingly slower and the sequential times are given on the fastest machine, the speed-ups are negatively affected by the heterogeneity. Nevertheless, they also show the flexibility of the abstractions.

B	N	m_S	μ_S	σ_S	μ_T	S_{m3-6}	S_{m3}
R250.1c	1	64	64.50	0.50	193.16	1.11	1
	2	64	64.30	0.46	103.32	2.07	1.87
	3	64	64.40	0.49	76.95	2.78	2.51
	4	64	64.40	0.49	62.69	3.41	3.08

B	N	μ_T	S_{m1}	S_{m3}
R250.5	m1	100.36	0.95	1.50
	m1-2	52.09	1.84	2.89
	m1-3	42.33	2.27	3.55
	m1-4	38.33	2.51	3.92
	m1-5	37.47	2.57	4.01
	m1-6	28.74	3.35	5.23

Fig. 11. Results on Graph Coloring (Heterogeneous mix of machines)

The Graph Coloring Results. Figure 11 depicts the results on graph coloring. Each line reports the average and deviation for 50 runs. The left part of the figure gives the results for the most homegeneous set of machines (m3-m6) on problem `R250.1c`. It reports the best coloring found (m_s), the average coloring (μ_s), and the standard deviation on the quality. It then reports the executing times in seconds (μ_T), and the speed-ups with respect to the average speed S_{m3-6} and with respect to the best machine in the experiments S_{m3}. Observe that, on the first three (roughly similar) machines, the speedups are about 2.78 and 2.51. With four machines, they are about 3.41 and to 3.08, which is still excellent especially with the last machine being slower. The right part of the

N	S^*	μ_T	σ_T	S_{m3}	N	S^*	μ_T	σ_T	S_{m3}
m3	72	42.30	40.21						
1	72	66.51	56.22	0.64	3	72	19.85	19.06	2.13
2	72	31.30	23.24	1.35	4	72	11.02	6.95	3.84

Fig. 12. Results on Golomb Ruler (Homogeneous mix of machines)

figure depicts the benchmarks for `R250.5` using all the machines. It reports the average times in seconds, as well as the speedups with respect to $m1$ (the fastest machine) and $m3$ (about the average machine). Observe first the speedup of 1.84 on the fastest two machines (wrt $m1$) which indicates that the distributed implementation does not lose much compared to the parallel implementation in [12] despite having to copy the runtime data structures. Compared to the average machine $m3$, the results are quite impressive giving a 5.23 speedups with 6 machines. Even compared to the fast $m1$, the results remain convincing.

The Golomb Results. Figure 12 depicts the same results on finding Golomb rulers using the homogeneous set of machines $m3 - 6$. The table reports the CPU time μ_T, its standard deviation σ_T, and the speedups with respect to the best machine in the set. Once again, the speedups are quite impressive. Moreover, the distributed implementation has a very interesting side-effect: it dramatically reduces the standard deviation on the execution times. This comes from the ability to interrupt the search, once a feasible solution has been found and, of course, the concurrent exploration from multiple startpoints.

7 Conclusion

The paper presented abstractions that allows distributed CBLS programs to be close to their sequential and parallel counterparts, keeping the conceptual and implementation overhead of distributed computing minimal. The new abstractions strongly rely on generic abstractions of CBLS and CP, such as models and solutions, and advanced control structures such as events and closures. A preliminary implementation in Comet exhibits significant speed-ups in constraint satisfaction and optimization applications. The implementation also seem to scale well with the number of machines, although more extensive eperimental evaluations are necessary. Overall, the simplicity of the abstractions and the resulting distributed CBLS, together with the excellent performance behavior of our implementation, indicates that distributed computing should become much more mainstream in years to come.

References

1. Allen, Chase, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt. The Fortress Language Specification, V0.866. Sun microsystems, Feb 2006.
2. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP.* Morgan Kaufmann, 2000. ISBN:1558606718.

3. L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5:46–55, Jan-March 1998.
4. R. Dorne and J.K. Hao. *Tabu Search for Graph Coloring, T-Colorings and Set T-Colorings*, chapter Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization, pages 77–92. Kluwer Academic Publishers, 1998.
5. I. Dotu and P. Van Hentenryck. A Simple Hybrid Evolutionary Algorithm for Finding Golomb Rulers. In *Evolutionary Computation, 2005*, pages 2018–2023. IEEE, 2005.
6. C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.
7. G. Harm and P. Van Hentenryck. A MultiStart Variable Neighborhood Search for Uncapacitated Warehouse Location. In *Proceedings of the 6th Metaheuristics International Conference (MIC-2005)*, Vienna, Austria, August 2005.
8. Ilog Solver 6.2. Documentation. Ilog SA, Gentilly, France, 2006.
9. M. Laguna. *Handbook of Applied Optimization*, chapter Scatter Search, pages 183–193. Oxford University Press, 2002.
10. M. Luby, A. Sinclair, and Zuckerman. D. Optimal Speedup of Las Vegas Algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.
11. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 101–110, Seattle, November 2002.
12. L. Michel and P. Van Hentenryck. Parallel Local Search in Comet. In *Eleventh International Conference on Principles and Practice of Constraint Programming*, Stiges, Spain, 2005.
13. G.E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.
14. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Proceedings of the Fifth International Conference on the Principles and Practice of Constraint Programming*, pages 346–360, Alexandria, Virginia, Oct. 1999.
15. M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice & Experience*, 6:109–133, 1995.
16. G. Resende and R. Werneck. A Hybrid Multistart Heuristic for the Uncapacitated Facility Location Problem. Technical Report TD-5RELRR, AT&T Labs Research, 2003. (To appear in the European Journal on Operations Research).
17. Christian Schulte. Parallel search made simple. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, Singapore, September 2000.
18. P. Van Hentenryck. *Constraint-Based Local Search.* The MIT Press, Cambridge, Mass., 2005.
19. P. Van Hentenryck and L. Michel. Control Abstractions for Local Search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 65–80, Cork, Ireland, 2003.
20. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.

High-Level Nondeterministic Abstractions in C++

Laurent Michel[1], Andrew See[1], and Pascal Van Hentenryck[2]

[1] University of Connecticut, Storrs, CT 06269-2155
[2] Brown University, Box 1910, Providence, RI 02912

Abstract. This paper presents high-level abstractions for nondeterministic search in C++ which provide the counterpart to advanced features found in recent constraint languages. The abstractions have several benefits: they explicitly highlight the nondeterministic nature of the code, provide a natural iterative style, simplify debugging, and are efficiently implementable using macros and continuations. Their efficiency is demonstrated by comparing their performance with the C++ library Gecode, both for programming search procedures and search engines.

1 Introduction

The ability to specify search procedures has been a fundamental asset of constraint programming languages since their inception (e.g., [1,3,13]) and a differentiator compared to earlier tools such as Alice [7] and MIP systems where search was hard-coded in the solver. Indeed, by programming the search, users may define problem-specific branching procedures and heuristics, exploit unconventional search strategies, break symmetries dynamically, and specify termination criteria for the problem at hand. The last two decades have also witnessed significant progress in this area (e.g., [6,8,9,12,14,15]): Modern constraint programming languages enable programmers to specify both the search tree and the search strategy, provide high-level nondeterministic abstractions with dynamic filtering and ordering, and support hybrid and heuristic search.

The embedding of constraint programming in mainstream languages such as C++ has also been a fundamental step in its acceptance, especially in industry. With constraint programming libraries, practitioners may use familiar languages and environments, which also simplifies the integration of a constraint programming solution within a larger application. Ilog Solver [10] is the pioneering system in this respect: it showed how the nondeterministic abstractions of constraint logic programming (e.g., goals, disjunction, and conjunction) can be naturally mapped into C++ objects. To specify a search procedure, users thus define C++ objects called goals, and combine them with logical connectives such as `or` and `and`. In recent years, constraint programming libraries have been enhanced to accommodate search strategies [9,4] (originally proposed in Oz [12]) and high-level nondeterministic abstractions [8] (originally from OPL [14]).

However these libraries, while widely successful, still have two inconveniences as far as specifying search procedures. On the one hand, they impose a recursive style for search procedures, which contrasts with the more familiar iterative constructs of OPL as indicated in [2]. Second, these libraries may obscure the natural nondeterministic structure of the program and may produce some non-trivial interleaving of `C++` code and library functions. This complicates the debugging process which alternates between library and user code.

This paper is an attempt to mirror, in constraint programming libraries, the high-level nondeterministic abstractions of modern constraint programming languages. The paper shows that it is indeed possible and practical to design a search component in `C++` that

– promotes an iterative programming style that expresses both sequential composition and nondeterminism naturally;
– simplifies the debugging process, since the `C++` stack now reflects the full control flow of the application;
– is as efficient as existing libraries.

The technical idea underlying the paper is to map the nondeterministic abstractions of Comet [15] into `C++` using macros and continuations. Obviously, since continuations are not primitive in `C++`, it is necessary to show how they can be implemented directly in the language itself. The implementation differs significantly from the OPL implementation in which the abstractions are implemented using Ilog Solver facilities.

The rest of the paper is organized as follows. Section 2 presents the nondeterministic abstractions and their benefits. Section 3 shows how to implement continuations in `C++`. Section 4 shows how to use macros and continuations to implement the nondeterministic abstractions. Section 5 presents the experimental results which shows that the nondeterministic abstractions can be implemented efficiently and compare well with the search implementation of Gecode.

2 The Search Abstractions

This section describes the search abstractions in `C++`. Section 2.1 starts by describing the nondeterministic abstractions used to define the search tree to explore. These abstractions are parameterized by a search controller that specifies how to explore the search tree. Search controllers are briefly discussed in Section 2.2 and are presented in depth in [15].

2.1 Nondeterministic Abstractions

The nondeterministic abstractions are mostly modelled after OPL [14].

Static Choices. The `try` construct creates a binary search node representing the choice between two alternatives. The snippet

```
TRY(sc)
  cout << "yes" <<endl;
OR(sc)
  cout << "no" <<endl;
ENDTRY(sc)
```

```
TRYALL(<sc>, <param>, <low>, <high>, <condition>, <ordering>)
  [<Statement>]*
ENDTRYALL(<sc>)
```

Fig. 1. The Syntax of the TRYALL Construct

nondeterministically produces two lines of output: the first choice displays yes, while the second one displays no. When the search controller sc implements a depth-first strategy, the instruction first executes the first choice, while the second choice is executed upon backtracking.

Note that TRY, OR, and ENDTRY are not extensions to C++: they are simply macros that encapsulate the instructions to create a search node, to implement backtracking, and to close the search node. The above code thus executes on standard C++ compilers.

Dynamic Choices. The TRYALL construct iterates over a range of values, filtering and ordering the candidate values dynamically. Figure 1 depicts the general syntax of the construct. The first parameter <sc> is the search controller. The <param> argument is the local variable used to store the selected value. Parameters <low> and <high> define the range of values, while <condition> holds for those values to consider in the range. Finally, the expression <ordering> specifies the order in which to try values. For instance, the snippet

```
TRYALL(sc, p, 0, 5, (p%2)==0, -p)
  cout << "p = "<< p << endl;
ENDTRYALL(sc)
```

nondeterministically produces three lines of output: p=4, p=2, and p=0. The instruction binds the parameter p to values 0 through 5 in increasing order of -p and skips those violating the condition (p%2)==0.

Encapsulated Search. The EXPLOREALL construct implements an encapsulated search that initializes the search controller and produces all solutions to its body. Figure 2 illustrates an encapsulated search for implementing a simple labeling procedure. The body of the encapsulated search (lines 2–9) iterates over the values 0..2 (line 2) and nondeterministically assigns x[i] to 0 or 1 (lines 3–7). Once all the elements in array x are labeled, the array is displayed in line 9. The right part of Figure 2 depicts the output of the encapsulated search for a depth-first search controller. Other similar constructs implement encapsulated search to find one solution or to find a solution optimizing an objective function.

It is important to emphasize some benefits of the nondeterministic abstractions. First, the code freely interleaves nondeterministic abstractions and arbitrary C++ code: it does not require the definition of classes, objects, or goals. Second, the nondeterministic structure of the program is clearly apparent, simplifying debugging with traditional support from software environments. In particular, C++ debuggers can be used on these nondeterministic programs, enabling users to follow the control flow of their programs at a high level of abstraction.

```
int x[3] = -1, -1, -1;                              0,0,0
EXPLOREALL(sc)                                      0,0,1
  for(int i=0; i<3 ; i++) {                         0,1,0
    TRY(sc)                                         0,1,1
      x[i] = 0;                                     1,0,0
    OR(sc)                                          1,0,1
      x[i] = 1;                                     1,1,0
    ENDTRY(sc)                                      1,1,1
  }
  cout << x[0]<<',' <<x[1]<<','<<x[2]<<endl;
ENDEXPLOREALL(sc)
```

Fig. 2. An Example of Encapsulated Search

```
EXPLOREALL(dfs)
for(int q = 0; q < N-1; q++)
  if (queen[q]->getSize()>1) {
    TRYALL(dfs, p, 0, N-1, queen[q]->hasValue(p),p)
      dfs->label(v,p);
    ENDTRYALL(dfs);
  }
ENDEXPLOREALL(dfs);
```

Fig. 3. A Search Procedure with a Static Variable Ordering

Ordered Iterations. Since they are implemented as macros, the nondeterministic abstractions can be naturally interleaved with C++ code. Figure 3 illustrates a simple search procedure for the n-queens problems. In the figure, the variable-ordering is static, while the value-ordering assigns first the smallest values in the domain. When the variable ordering is dynamic, it is useful to introduce a FORALL abstraction to avoid tedious bookkeeping by programmers. Figure 4 depicts a search procedure where the first-fail principle is used for variable selection (line 1) and where values close to the middle of the board are tried first (line 2). Apart from the syntax which is less elegant, this search procedure is at the same level of expressiveness as the equivalent search procedures in OPL [14].

2.2 Search Controllers

The nondeterministic abstractions define the search tree to explore by creating search nodes as the program executes. They are parameterized by a search controller specifying how to explore this tree. Figure 5 shows part of the interface of search controllers. The primary methods of this interface are `addNode` and `fail`, both of which are pure and virtual. The `addNode` method adds a search node to the controller, while method `fail` is called upon encountering a failure.

Figure 6 shows a specialization of the `SearchController` for depth-first search. The `addNode` and `fail` methods use a stack of search nodes. Method `addNode` pushes a node on the stack, while method `fail` pops a search node

```
EXPLOREALL(dfs)
  FORALL(q,0,N-1,queen[q]->getSize()>1,queen[q]->getSize())
    TRYALL(dfs, p, 0, N-1,queen[q]->hasValue(p),abs(mid-p)){
      dfs->label(v,p);
    ENDTRYALL(dfs);
  ENDFORALL;
ENDEXPLOREALL(dfs);
```

Fig. 4. A Search Procedure with a Dynamic Variable Ordering

```
class SearchController {
protected:
  SearchNode _explore;
public:
  SearchController() {}
  virtual void explore(SearchNode n) { _explore = n; }
  virtual void fail() = 0;
  virtual void addNode(SearchNode n) = 0;
  ...
}
```

Fig. 5. The Interface of Search Controller (Partial Description)

from the stack and executes it. Observe that, when programmers uses predefined search controllers, they never need to manipulate search nodes or even know that they exist: It is the role of the nondeterministic abstractions to create the search nodes and to apply the appropriate methods on the search controllers.

3 Search Nodes as C++ Continuations

As in COMET [15], the nondeterministic abstractions are implemented using continuations. Since C++ does not support continuations natively, this section describes how to implement continuations in the languages itself. Recall that a continuation captures the current state of computation, i.e., the program counter, the stack, and the registers (but not the heap). Once captured, the continuation can be executed at a later time. Figure 7 illustrates continuations on a simple example: The C++ program is shown on the left of the figure and its output is shown on the right. The program computes, using continuations, the factorial of 5, 4, ..., 0. It captures a continuation in line 6 and calls `fact` with the current value of `i` (5). After printing the result (line 7), the program tests whether the `i`'s value is not smaller than 1. In this case, `i`'s value is decremented and the continuation is executed. The execution then restarts in line 7, computes the factorial of 4, and iterates the process again.

We now show how to implement continuations in C++ using `setjmp` and `longjmp`. The interface of a continuation is specified in Figure 8. Its main methods are `restore` (to restore the stack of the continuation) and `execute` to execute

```
class DFS : public SearchController {
  Stack<SearchNode> _stack;
  void addNode(SearchNode n) { _stack.push(n); }
  void fail() {
    if (_stack.empty()) _explore->execute();
    else _stack.pop()->execute(); }
}
```

Fig. 6. The Implementation of a Depth-First Search Controller (Partial Description)

```
int fact(int n){
  if (n==0) return 1; return n*fact(n-1);
}
int main(int argc, char*argv[]){
  initContinuations(&argc);
  int* i = new int(5);
  Continuation* c = captureContinuation();
  cout <<"fact("<<*i<<")="<<fact(*i)<<endl;
  if(*i >=1){ (*i)--; c->execute(); }
  return 0;
}
```

```
fact(5)=120
fact(4)=24
fact(3)=6
fact(2)=2
fact(1)=1
fact(0)=1
```

Fig. 7. A Simple Example Illustrating Continuations in C++

the continuation. Its instance variables are used to save the buffer _target used by longjmp, the stack, and the number of times the continuation has been called (_calls). The constructor in lines 8–12 saves the C++ stack. Note the definition of search nodes in line 19: Search nodes are simply pointers to continuations.

Figure 9 depicts the core of the implementation. Function initContinuation stores the base of the stack (i.e., the address of argc) in static variable baseStack. A correct value for baseStack is critical to save and restore continuations.

Function captureContinuation (line 2–11) captures a continuation. Line 5 creates an instance of class Continuation using, as arguments, the address of k and the size baseStack-(char*)&k to be able to save the stack. Line 6 uses the C++ instruction setjmp to save the program counter and the registers into the field _target of the continuation. After execution of line 6, the continuation has been captured and line 7 is either executed just after the capture (in which case jmpval is 0) or after a call to longjmp on _target (in which case jmpval is the continuation passed to longjmp). In the second case, function captureContinuation must restore the stack (line 8) before returning the continuation in line 9.

Method restore is depicted in lines 11–14. It restores the stack in line 12 and increments instance variable _calls to specify that the continuation has been called one more time. This last operation is important to implement the nondeterministic abstractions. Finally, method execute simply performs a longjmp on instance variable _target, passing the continuation itself. The effect is to restart the execution in line 7 of captureContinuation after having assigned jmpval to the continuation, inducing the stack restoration in line 8.

```
class Continuation {
  jmp_buf _target;      // instruction to return to. used by long_jmp
  int _length;          // size of captured stack
  void* _start;         // location to restore the stack
  char* _data;          // copy of the stack
  int _calls;           // number of calls to this continuation
public:
  Continuation(int len,void* start) {
    _length = len; _start = start;
    _data = new char[len];
    memcpy(_data,_start,_length);
  }
  ~Continuation();
  void restore();
  void execute();
  const int nbCalls() const {return _calls;}
  friend Continuation* captureContinuation();
};
typedef SearchNode Continuation*;
```

Fig. 8. The Interface of Continuations

Continuations only use standard C functions and are thus portable to all architectures with a correct implementations of these functions. In absence of `setjmp` and `longjmp`, `captureContinuation` and `execute` can be implemented using `getContext` and `setContext`, or in assembly. Our implementation based on `setjmp/longjmp` has been successfully tested on three different hardware platforms (Intel x86, PowerPC, and UltraSparc) and four operating systems (Linux, Windows XP, Solaris, and OSX). The only platform where it fails is Itanium because of its implementation of `setjmp` and `longjmp` does not conform to the specifications: it does not allow several `longjmp` calls for the same `setjump`.

It is important to note that the implementation of Ilog Solver [5] also uses `setjmp` and `longjmp` [11]. The novelty here is to save the stack before calling `setjmp` and restoring the stack after calling `longjmp`. The benefits are twofold. On the one hand, it enables the implementation of high-level nondeterministic abstractions such as `tryall` in `C++`. On the other hand, it enables continuations to be called at any time during the execution even if the stack has fundamentally changed. As a result, continuations provide a sound basis for complex search procedures jumping from node to node arbitrarily in the search tree.

Finally, it is worth emphasizing that the nondeterministic abstractions can be used across function/method calls. They can also be encapsulated in methods to provide generic search procedures for various problem classes.

4 Implementation

It remains to show how to implement the nondeterministic abstractions in terms of continuations. As mentioned earlier, the nondeterministic abstractions are

```
static char* baseStack = 0;
void initContinuations(int* base) { baseStack = (char*) base; }
Continuation* captureContinuation() {
  Continuation* jmpval;
  Continuation* k;
  k = new Continuation(baseStack-(char*)&k,&k);
  jmpval = (Continuation*) setjmp(k->_target);
  if (jmpval != 0)
    jmpval->restore();
  return k;
}
void Continuation::restore() {
  memcpy(_start,_data,_length);
  ++_calls;
}
void Continuation::execute() { longjmp(_target,(int)this); }
```

Fig. 9. Functions to create and use Continuations

implemented as macros that capture and call continuations and apply methods of the search controller. Recall that the use of macro expansions does not interfere with the debugger. Breakpoints placed within the body of the search procedure behave as expected. Also, since a macro is expanded to a single line of code, line-by-line stepping skips over the macro, while single stepping enters the body of the macro definition.

Static Choices. The statement

```
TRY(sc) ⟨ A ⟩ OR(sc) ⟨ B ⟩ ENDTRY(sc)
```

is rewritten as

```
0.  Continuation* cont = captureContinuation();
1.  if(cont->nbCalls() == 0) {
2.    sc->addNode(cont);
3.    ⟨ A ⟩
4.  } else {
5.    ⟨ B ⟩
6.  }
```

where lines 1–2 are produced by the macro `TRY(sc)`, line 4 is from the macro `OR(sc)`, and lines 6 is from `ENDTRY(sc)`. The resulting code can be explained as follows. Line 0 captures a continuation. It then tests whether the continuation has not yet been called (line 1), which is always the case the first time the `TRY` instruction is executed. As a result, lines 2–3 are executed: line 2 adds the continuation (or search node) in the search controller while line 3 executes the instructions `A`. When the continuation is called, execution comes back to line 1 but the number of calls to the continuation is not zero. As a result, the instructions `B` in line 5 are executed. Observe that the exploration strategy is left to the controller: the above implementation does not prescribe a depth-first strategy and the continuation can be called at any time during the computation to execute line 5.

Encapsulated Search. The `EXPLOREALL` is an encapsulated search exploring the search tree specified by its body. The code

```
EXPLOREALL(sc) ⟨ A ⟩ ENDEXPLOREALL(sc)
```

is rewritten as:

```
0.  Continuation *cont = captureContinuation();
1.  if(cont->nbCalls()==0) {
2.    sc->explore(cont);
3.    ⟨A⟩
4.    sc->fail();
5.  }
```

where lines 0–2 are produced by `EXPLOREALL(sc)` and lines 4–6 are generated by `ENDEXPLOREALL(sc)`. The key implementation idea is to create a continuation `cont` representing what to do when the search tree defined by `A` is fully explored. The test in line 2 holds for the first execution of `EXPLOREALL`, in which case lines 2–4 are executed. They tell the search controller to start an encapsulated search, execute `A`, and fail (line 4). The failure makes sure that the search tree defined by `A` is fully explored. When this is the case, the continuation `cont` is executed, leading to the execution of line 6 which terminates the encapsulated search.

Dynamic Choices. Consider now the `TRYALL` abstraction. The presentation is in stepwise refinements, starting first with a version with no filtering and ordering and adding these features one at a time. The code

```
TRYALL4(sc, p, low, high) ⟨ A ⟩ ENDTRYALL4(sc)
```

is rewritten as

```
0.  int p=(low);
1.  int* curIndex = new int(p);
2.  Continuation *cont = captureContinuation();
3.  if (*curIndex <= (high) ){
4.    p=(*curIndex);
5.    ++(*curIndex);
6.    (sc)->addNode(cont);
7.    ⟨ A ⟩
8.  } else {
9.    delete curIndex;
10.   sc->fail();
11. }
```

where lines 0–6 are generated by `TRYALL4(sc, p, low, high)` and lines 8–11 by `ENDTRYALL4(sc)`. There are two features to emphasize here. First, `p` is declared as a local variable in line 0 and can then be used naturally in `A`. Second, `curIndex` holds an integer representing the current index in `low..high`. `curIndex` is allocated on the heap since otherwise its value would be restored to `low` when the continuation is called. The rest of the `TRYALL` implementation then follows the `TRY` implementation. The continuation is captured in line 2. As long as the current index is within the range (line 3), a call to the continuation (or the first call to `TRYALL`) assigns the current index to `p` (line 4), increments

```
0.  int p=low;
1.  int* curIndex = new int(p);
2.  Continuation* cont = captureContinuation();
3.  bool found=false;
4.  while ((*curIndex) <= (high)) {
5.    p=(*curIndex)++;
6.    if (cond) { found=true; break; }
7.  }
8.  if (found){
9.    sc->addNode(cont);
10.   ⟨ A ⟩
11. } else {
12.   delete curIndex;
13.   sc->fail();
14. }
```

Fig. 10. The `TRYALL` Implementation with Filtering

the index for subsequent iterations (line 5), adds the continuation to the search controller to allow additional iterations (line 6), and executes `A` (line 7). When all values in the range are explored, the implementation releases the space taken by `curIndex` (line 9) and fails (line 10).

Figure 10 shows how to generalize the implementation when values in the range are filtered as in
`TRYALL5(sc, p, low, high, cond) ⟨ A ⟩ ENDTRYALL5(sc)`
The main novelty in Figure 10 is the addition of a loop (lines 4–6) to find the first element in the range satisfying the condition `cond`.

Finally, Figure 11 depicts the implementation of the `TRYALL` abstraction with filtering and ordering, i.e.,
`TRYALL(sc, p, low, high, cond, ordering) ⟨ A ⟩ ENDTRYALL(sc)`
The key idea is to replace `curIndex` by an array of Booleans that keep track of which values have been tried already. The array is allocated on the heap in line 2 and initialized in line 3. The continuation is captured in line 4 and the rest of the code depicts the treatment performed for all successive calls to the `TRYALL` instruction. Lines 7–8 search for the available element in the range satisfying condition `cond` and minimizing expression `ordering`. If such an element exists (line 13), the continuation is added to the controller, array `avail` is updated, and the instructions `A` are executed. Otherwise, all elements have been tried, which means that array `avail` can be released and the `TRYALL` must fail.

Iterations with Ordering. The implementation of the `FORALL` instruction is simpler since it does not create search nodes: it simply iterates over the range and selects the value satisfying the condition and minimizing the ordering expression. Figure 12 describes the implementation of the code
`FORALL(sc, p, low, high, cond, ordering) ⟨ A ⟩ ENDFORALL(sc)`
Unlike `TRYALL`, the array `avail` must be allocated on the `C++` stack (line 2). Indeed, the available values must be restored on backtracking, which is achieved automatically by continuations when the array is allocated on the `C++` stack.

```
int p = 0;
int l = (low); int h = (high);
bool* avail = new bool[h-l+1];
for(int i=0; i < h-l+1; i++) avail[i]=true;
Continuation *cont = captureContinuation();
bool found=false;
int bestEval = INT_MAX;
for(int k=l; k <= h; k++)
  if( (bestEval > (ordering)) && avail[k-l] && (cond)) {
    found = true;
    bestEval = (ordering);
    p = k;
  }
if (found) {
  avail[p-l]=false;
  sc->addNode(cont);
  ⟨ A ⟩
} else {
  delete[] avail;
  sc->fail();
}
```

Fig. 11. The Implementation of the TRYALL Abtraction with Filtering and Ordering

5 Experimental Results

This section presents the experimental results demonstrating the efficiency of the implementation. It first shows that the cost of using continuations is not prohibitive. Then, it demonstrates that the abstractions are comparable in efficiency to the search procedures of existing constraint libraries. The CPU Times are given on a Pentium IV 2.0 GHz running Linux 2.6.11.

On the Efficiency of Continuations. One possible source of inefficiency for the nondeterministic abstractions is the overhead of capturing and restoring continuations. To quantify this cost, we use a simple backtrack search for the queens problem and we compare a search procedure written in C++ (and thus with a recursive style) with a search procedure using the nondeterministic abstractions (and thus with an iterative style). Figures 13 and 14 depict the two search procedures and their common attack function. Table 1 shows the runtime of the recursive (R) and nondeterministic (N) search procedures and the percentage increase in CPU time. The results show that the percentage increase in CPU time decreases as the problem size grows and goes down to 54% for the 32-queens problem. These results are noteworthy, since they use a mainstream, non-garbage collected, highly efficient language. Moreover, these programs do not involve any constraint propagation and do not need to save and restore the states of domain variables and constraints. As such, these tests represents the pure cost of the abstractions compared to the hand-coded implementation.

```
int p = 0;
int l = (low); int h = (high);
bool* avail = (bool*) alloca(h-l+1);
for(int i=0; i < h-l+1; i++) avail[i]=true;
while(true) {
  bool found=false;
  int bestEval = INT_MAX;
  for(int k=l; k <= h; k++)
    if( (bestEval > (ordering)) && avail[k-l] && (cond)) {
      found = true;
      bestEval = (ordering);
      p = k;
    }
  if (found) {
    avail[p-l]=false;
    ⟨ A ⟩
  } else break;
}
```

Fig. 12. The Implementation of the `FORALL` Abtraction with Filtering and Ordering

Table 1. The Pure Cost of the Nondeterministic Abstractions

n	16	18	20	22	24	26	28	30	32
R	.007	.028	.152	1.486	.405	.443	3.748	78.900	133.86
N	.014	.063	.308	2.878	.731	.762	6.175	125.64	205.82
$(N-R)/R$	1.00	1.25	1.026	0.937	0.805	0.720	0.648	0.592	0.538

Comparison with an Existing Library. We now compare the efficiency of the nondeterministic abstractions with an existing `C++` constraint programming library: GECODE [4]. Figure 15 shows the partial GECODE model used for the queens problem. The depth-first search in GECODE has a *copy distance* parameter specifying the number of branchings to perform using recomputation before cloning the search space. A value of 1 (meaning no recomputation) was used in the experiments as that tends to give the best performance for the queens problem. Observe the call to the built-in search procedure in line 11: It has a static left-to-right ordering and starts first with the smallest values. We compare the performance of GECODE with a similar statement using a constraint programming library using trailing and the search procedure depicted in Figure 3 and the search controller (partially) described in Figure 16. Both implementations use exactly the same search procedure: it is built-in in the case of GECODE but uses our high-level nondeterministic abstractions in our case. Note also that the search controller now calls the CP manager to push and pop choices (in addition to search nodes) to implement trailing. Table 3 depicts the computational results: they indicate that the program with the nondeterministic abstractions is slightly more efficient than GECODE. These results tend to demonstrate the practicability of our approach, since the queens problem has little propagation

```
bool attack(int* queen,int n,int i,int v){
  for(int k=0; k < i; k++) {
    if( queen[k] == v )return true;
    if( queen[k]+k == v+i)return true;
    if( queen[k]-k == v-i)return true;
  }
  return false;
}
bool search(int* queen,int n,int i){
  if(i >= n) return true;
  for(int v = 0; v < n; v++){
    if (attack(queen,n,i,v)) continue;
    queen[i]=v;
    if (search(queen,n,i+1)) return true;
  }
  return false;
}
search(queens,N,0);
```

Fig. 13. The Recursive Version of the Simple Backtrack Search

```
for(int q=0;q<N;q++){
  TRYALL4(sc, v, 0, N-1, !attack(queen,N,q,v))
    queen[q]=v;
  ENDTRYALL4(sc);
}
```

Fig. 14. The Nondeterministic Version of the Simple Backtrack Search

compared with realistic constraint programs and hence the cost of the abstractions will be even more negligible on more complex applications.

Programming Search Engines. The previous comparison used two different solvers and the difference in efficiency may partially be attributed to their respective efficiency. To overcome this limitation, our last experiment only uses Gecode as the underlying solver. It compares the built-in implementation of depth-first search in Gecode with an implementation using our nondeterministic abstractions. Recall that Gecode manipulates computation spaces representing the search tree. The following C++ code

```
0.  EXPLORE(gecode)
1.    while (gecode->needBranching()) {
2.      int alt = gecode->getNbAlternatives();
3.      TRYALL4(gecode, a, 0, alt-1)
4.        gecode->tryCommit(a);
5.      ENDTRYALL4(gecode);
6.  }
7.  ENDEXPLORE(gecode);
```

```
0.  class Queens : public Example {
1.      IntVarArray q; // Position of queens on boards
2.    public:           // The actual problem
3.      Queens(const Options& opt) : q(this,opt.size,0,opt.size-1) {
4.        const int n = q.size();
5.        for (int i = 0; i<n; i++)
6.          for (int j = i+1; j<n; j++) {
7.            post(this, q[i] != q[j]);
8.            post(this, q[i]+i != q[j]+j);
9.            post(this, q[i]-i != q[j]-j);
10.         }
11.       branch(this, q, BVAR_NONE, BVAL_MIN);
12.     }
13.   }
14.   Example::run<Queens,DFS>(opt);
```

Fig. 15. The GECODE Model for the Queens Problem

```
0.  class CPDFS : public SearchController {
1.    Stack<SearchNode> _stack;
2.    CPManager* _mgr;
3.  public:
4.    ...
5.    void addNode(SearchNode n) {
6.      _nodes->push(n);
7.      _mgr.pushChoice(f)
8.    }
9.    void fail() {
10.     _mgr->popChoice();
11.     if (_stack.empty()) _explore->execute();
12.     else _stack.pop()->execute();
13.   }
14. }
```

Fig. 16. A Depth-First Search Controller for a CP Library with Trailing

Table 2. Performance Comparison (in Seconds) with GECODE on the Queens Problem

n	16	18	20	22	24	26	28	30	32
Gecode	.06	.20	.98	7.83	2.08	2.08	16.34	301.27	507.08
ND	.05	.18	.92	7.56	1.93	1.83	14.96	294.20	498.37

is an implementation of depth-first search for GECODE that uses our nondeterministic abstraction. The code iterates branching until the tree is fully explored (line 1). To branch, the code retrieves the number of alternatives (line 2) and performs a TRYALL to try each alternative (line 4). The code uses a gecode controller to clone and restore the spaces appropriately (which is not shown for space reasons). Note also the combination of a C++ while instruction with TRYALL.

Table 3. Performance Comparison in Seconds on Gecode Only

n	16	18	20	22	24	26	28	30	32
Gecode	.06	.20	.98	7.83	2.08	2.08	16.34	301.27	507.08
ND + Gecode	0.05	.19	.97	7.55	2.02	1.96	16.28	292.80	495.69

Table 2 depicts the computational results. This evaluation has the merit of comparing the search procedures with exactly the same constraint solver and the search procedure coded by the designer of the library. Once again, the nondeterministic abstractions are slightly more efficient than the builtin implementation of Gecode, although they perform exactly the same number of clones, failures, and propagation calls. The results thus indicate that the nondeterministic abstractions are not only expressive and natural; they are also very efficient.

6 Conclusion

This paper showed how to use macros and continuations in `C++` to support the high-level nondeterministic abstractions found in recent constraint languages. The resulting design has several benefits. The abstractions promote an iterative style for search procedures, simplify debugging since the `C++` stack now reflects directly the control flow of the program, and allow for natural implementations of search strategies. The implementation, which uses `setjmp/longjmp`, is shown to compare well with the `C++` library Gecode.

Acknowledgments

This work was supported by the National Science Foundation grant DMI-0423607. Special thanks to the third reviewer for some insightful comments.

References

1. A. Colmerauer. Opening the Prolog-III Universe. *BYTE Magazine*, 12(9), 1987.
2. de Givry, S. and Jeannin, L. Tools: A library for partial and hybrid search methods. In *CP-AI-OR'03.*
3. N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and some Electrical Engineering Problems. In *ICLP-87.*
4. http://www.gecode.org/. Generic Constraint Development Environment, 2005.
5. Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.
6. F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *CP'98.*
7. J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.
8. L. Michel and P. Van Hentenryck. Modeler++: A Modeling Layer for Constraint Programming Libraries. In *CP-AI-OR'2001.*
9. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *CP'99.*

10. J-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS'94*.
11. J.-F.. Puget. Personal Communication, March 2006.
12. C. Schulte. Programming Constraint Inference Engines. In *CP'97*.
13. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, Cambridge, MA, 1989.
14. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1999.
15. P. Van Hentenryck and L. Michel. Nondeterministic Control for Hybrid Search. In *CP-AI-OR'05*.

A Structural Characterization of Temporal Dynamic Controllability

Paul Morris

NASA Ames Research Center
Moffett Field, CA 94035, U.S.A.
pmorris@email.arc.nasa.gov

Abstract. An important issue for temporal planners is the ability to handle temporal uncertainty. Recent papers have addressed the question of how to tell whether a temporal network is *Dynamically Controllable*, i.e., whether the temporal requirements are feasible in the light of uncertain durations of some processes. Previous work has presented an $O(N^5)$ algorithm for testing this property. Here, we introduce a new analysis of temporal cycles that leads to an $O(N^4)$ algorithm.

1 Introduction

Many Constraint-Based Planning systems (e.g. [1]) use Simple Temporal Networks (STNs) to test the consistency of partial plans encountered during the search process. These systems produce *flexible* plans where every solution to the final Simple Temporal Network provides an acceptable schedule. The flexibility is useful because it provides scope to respond to unanticipated contingencies during execution, for example where some activity takes longer than expected. However, since the uncertainty is not modelled, there is no guarantee that the flexibility will be sufficient to manage a particular contingency.

Many applications, however, involve a specific type of temporal uncertainty where the duration of certain processes or the timing of exogenous events is not under the control of the agent using the plan. In these cases, the values for the variables that are under the agent's control may need to be chosen so that they do not constrain uncontrollable events whose outcomes are still in the future. This is the *controllability* problem. By formalizing this notion of temporal uncertainty, it is possible to provide guarantees about the sufficiency of the flexibility.

In [2], several notions of controllability are defined, including *Dynamic Controllability* (DC). Roughly speaking, a network is dynamically controllable if there is a strategy for satisfying the constraints that depends only on knowing the outcomes of past uncontrollable events.

In [3] an algorithm is presented that determines DC and runs in polynomial time under the assumption that the maximum size of links in the STN is bounded. Thus, the algorithm is pseudo-polynomial like arc-consistency, rather than being a strongly polynomial algorithm such as, for example, the Bellman-Ford algorithm [4] for determining consistency of a distance graph. What makes the latter algorithm strongly polynomial is the Bellman-Ford *cutoff*, which restricts the number of iterations based on the number of nodes in the network. The first strongly polynomial algorithm for DC is presented in [5]. This introduces an algorithm with an $O(N^3)$ inner-loop and an outer loop with an $O(N^2)$ cutoff. Thus, the entire algorithm runs in $O(N^5)$ time. The paper also simplifies the mathematical formulation of the reduction rules.

In this paper, we further simplify the mathematical formulation and introduce a structural characterization of DC in terms of the absence of a particular type of negative cycle. This is analogous to the result characterizing consistency of ordinary STNs in terms of the absence of negative cycles in the distance graph. This leads to a reformulated algorithm for DC with an $O(N^3)$ inner-loop and an $O(N)$ cutoff for the outer loop. Thus, the entire algorithm runs in $O(N^4)$ time.

2 Background

This background section defines the types of controllability, and outlines the previous DC algorithms, essentially following [3,5].

A Simple Temporal Network (STN) [6] is a graph in which the edges are annotated with upper and lower numerical bounds. The nodes in the graph represent temporal events or *timepoints*, while the edges correspond to constraints on the durations between the events. Each STN is associated with a *distance graph* derived from the upper and lower bound constraints. An STN is consistent if and only if the distance graph does not contain a negative cycle. This can be determined by a single-source shortest path propagation such as in the Bellman-Ford algorithm [4] (faster than Floyd-Warshall for sparse graphs, which are common in practical problems). To avoid confusion with edges in the distance graph, we will refer to edges in the STN as *links.*

A Simple Temporal Network With Uncertainty (STNU) is similar to an STN except the links are divided into two classes, *requirement links* and *contingent links.* Requirement links are temporal constraints that the agent must satisfy, like the links in an ordinary STN. Contingent links may be thought of as representing causal processes of uncertain duration, or periods from a reference time to exogenous events; their finish timepoints, called *contingent timepoints*, are controlled by Nature, subject to the limits imposed by the bounds on the contingent links. All other timepoints, called *executable timepoints*, are controlled by the agent, whose goal is to satisfy the bounds on the requirement links. We assume the durations of contingent links vary independently, so a control procedure must consider every combination of such durations. Each contingent link is required to have positive (finite) upper and lower bounds, with the lower bound strictly less than the upper. Without loss of generality, we assume contingent links do not share finish points. (If desired, they can be constrained to simultaneity by $[0, 0]$ requirement links. It is also known that networks with coincident contingent finishing points cannot be DC.)

Choosing one of the allowed durations for each contingent link may be thought of as reducing the STNU to an ordinary STN. Thus, an STNU determines a family of STNs corresponding to the different allowed durations; these are called *projections* of the STNU.

Given an STNU with N as the set of nodes, a *schedule* T is a mapping

$$T : N \rightarrow \Re$$

where $T(x)$ is called the *time* of timepoint x. A schedule is *consistent* if it satisfies all the link constraints. The *prehistory* of a timepoint x with respect to a schedule T, denoted by $T\{\prec x\}$, specifies the durations of all contingent links that finish prior to x.

An *execution strategy* S is a mapping

$$S : \mathcal{P} \rightarrow \mathcal{T}$$

where $\mathcal{P}$ is the set of projections and $\mathcal{T}$ is the set of schedules. An execution strategy S is *viable* if $S(p)$, henceforth written S_p, is consistent with p for each projection p.

We are now ready to define the various types of controllability, following [7].

An STNU is *Weakly Controllable* if there is a viable execution strategy. This is equivalent to saying that every projection is consistent.

An STNU is *Strongly Controllable* if there is a viable execution strategy S such that

$$S_{p1}(x) = S_{p2}(x)$$

for each executable timepoint x and projections $p1$ and $p2$. In Strong Controllability, a "conformant" strategy (i.e., a fixed assignment of times to the executable timepoints) works for all the projections.

An STNU is *Dynamically Controllable* if there is a viable execution strategy S such that

$$S_{p1}\{\prec x\} = S_{p2}\{\prec x\} \Rightarrow S_{p1}(x) = S_{p2}(x)$$

for each executable timepoint x and projections $p1$ and $p2$. Thus, a Dynamic execution strategy assigns a time to each executable timepoint that may depend on the outcomes of contingent links in the past, but not on those in the future (or present). This corresponds to requiring that only information available from observation may be used in determining the schedule. We will use *dynamic strategy* in the following for a (viable) Dynamic execution strategy.

It is easy to see from the definitions that Strong Controllability implies Dynamic Controllability, which in turn implies Weak Controllability. In this paper, we are primarily concerned with Dynamic Controllability.

2.1 Previous Algorithms

It was shown in [3] that determining Dynamic Controllability is tractable, and an algorithm was presented that ran in pseudo-polynomial time. We will refer to this as the classic algorithm.

The classic algorithm involves repeated checking of a special consistency property called pseudo-controllability. An STNU is *pseudo-controllable* if it is consistent in the STN sense and none of the contingent links are squeezed, where a contingent link is *squeezed* if the other constraints imply a strictly tighter lower bound or upper bound for the link. The pseudo-controllability property is tested by computing the AllPairs Shortest Path graph using Johnson's Algorithm [4]. If the network passes the test, the algorithm then analyzes triangles of links and possibly tightens some constraints in a way that has been shown not to change the status of the network as DC or non-DC, but makes explicit all limitations to the execution strategies due to the presence of contingent links.

Some of the tightenings involve a novel temporal constraint called a *wait.* Given a contingent link AB and another link AC, the $<\mathrm{B}, t>$ annotation on AC indicates that execution of the timepoint C is not allowed to proceed until after either B has occurred or t units of time have elapsed since A occurred. Thus, a wait is a ternary constraint involving A, B, and C. It may be viewed as a lower bound of t on AC that can be discarded if B occurs first. Note that the annotation resembles a binary constraint on AC.

In order to describe the tightenings, the notation $\mathrm{A} \overset{[\mathrm{x,y}]}{\Longrightarrow} \mathrm{B}$ (or $\mathrm{B} \overset{[\mathrm{x,y}]}{\Longleftarrow} \mathrm{A}$) indicates a contingent link with bounds $[x, y]$ between A and B. We use the similar notation of $\mathrm{A} \xrightarrow{[\mathrm{x,y}]} \mathrm{B}$ (or $\mathrm{B} \xleftarrow{[\mathrm{x,y}]} \mathrm{A}$) for ordinary links.

We can summarize the tightenings, called *reductions*, used in the classic algorithm as follows.

(*Precedes Reduction*) If $u \geq 0,\ y' = y - v,\ x' = x - u,$
$\mathrm{A} \overset{[\mathrm{x,y}]}{\Longrightarrow} \mathrm{B} \xleftarrow{[\mathrm{u,v}]} \mathrm{C}$ adds $\mathrm{A} \xrightarrow{[\mathrm{y',x'}]} \mathrm{C}$

(*Unordered Reduction*) If $u < 0, v \geq 0,\ y' = y - v,$
$\mathrm{A} \overset{[\mathrm{x,y}]}{\Longrightarrow} \mathrm{B} \xleftarrow{[\mathrm{u,v}]} \mathrm{C}$ adds $\mathrm{A} \xrightarrow{<\mathrm{B}, y'>} \mathrm{C}$

(*Simple Regression*) If $y' = y - v,$
$\mathrm{A} \xrightarrow{<\mathrm{B}, y>} \mathrm{C} \xleftarrow{[\mathrm{u,v}]} \mathrm{D}$ adds $\mathrm{A} \xrightarrow{<\mathrm{B}, y'>} \mathrm{D}$

(*Contingent Regression*) If $y \geq 0, \mathrm{B} \neq \mathrm{C},$
$\mathrm{A} \xrightarrow{<\mathrm{B}, y>} \mathrm{C} \overset{[\mathrm{u,v}]}{\Longleftarrow} \mathrm{D}$ adds $\mathrm{A} \xrightarrow{<\mathrm{B}, y - u>} \mathrm{D}$

(*"Unconditional" Reduction*) If $u \leq x,$
$\mathrm{B} \overset{[\mathrm{x,y}]}{\Longleftarrow} \mathrm{A} \xrightarrow{<\mathrm{B}, u>} \mathrm{C}$ adds $\mathrm{A} \xrightarrow{[\mathrm{u},\infty]} \mathrm{C}$

(*General Reduction*) If $u > x,$
$\mathrm{B} \overset{[\mathrm{x,y}]}{\Longleftarrow} \mathrm{A} \xrightarrow{<\mathrm{B}, u>} \mathrm{C}$ adds $\mathrm{A} \xrightarrow{[\mathrm{x},\infty]} \mathrm{C}$

The tightenings involve new links that are added when the given pattern is satisfied unless tighter links already exist. The extensive motivation for these in [3] cannot be repeated here due to lack of space. However, some examples may help to give the basic idea.

Example 1. A $\overset{[1,2]}{\Longrightarrow}$ B $\overset{[1,1]}{\longleftarrow}$ C. Here we must schedule C exactly one time unit before B without knowing when B will occur. This requirement cannot be achieved in practical terms, although the network is initially consistent in the STN sense. The *Precedes Reduction* makes the inconsistency explicit. Contrast this with A $\overset{[1,2]}{\Longrightarrow}$ B $\overset{[1,1]}{\longrightarrow}$ C, where B can be observed before executing C, so no addition is needed.

Example 2. A $\overset{[1,2]}{\Longrightarrow}$ B $\overset{[1,2]}{\longleftarrow}$ C. Note that the CB constraint implies C precedes B. This means the agent must decide on a timing for C before information about the timing of B is available, and must do it in a way that the CB constraint is satisfied no matter when B occurs. The only way to accomplish this given our ignorance of B is to constrain C relative to A in such a way that the CB constraint becomes redundant. The *Precedes Reduction* does this by constraining C to happen simultaneously with A.

Example 3. A $\overset{[1,3]}{\Longrightarrow}$ B $\overset{[-1,1]}{\longleftarrow}$ C. Here we cannot safely execute C before B until time 2 after A (otherwise, if B occurs at 3, the [-1,1] constraint would be violated). After that, we can execute C prior to B if we wish, because we know B will finish within one more time unit. Thus, we place a $<\!B, 2\!>$ constraint on AC.

2.2 Labelled Distance Graph and Cutoff Algorithm

We now review the developments in [5], which re-expresses the reductions in a more mathematically concise form.

An ordinary STN has an alternative representation as a *distance graph*, in which a link A $\overset{[x,y]}{\longrightarrow}$ B is replaced by two edges A $\overset{y}{\longrightarrow}$ B and A $\overset{-x}{\longleftarrow}$ B, where the y and $-x$ annotations are called *weights*. Edges with a weight of ∞ are omitted. The distance graph may be viewed as an STN in which there are only upper bounds. This allows shortest path methods to be used to determine consistency, since an STN is consistent if and only if the distance graph does not contain a cycle with negative total distance [6].

Similarly, there is an analogous alternative representation for an STNU called the *labelled distance graph* [5]. This is actually a multigraph (which allows multiple edges between two nodes), but we refer to it as a graph for simplicity. In the labelled distance graph, each requirement link A $\overset{[x,y]}{\longrightarrow}$ B is replaced by two edges A $\overset{y}{\longrightarrow}$ B and A $\overset{-x}{\longleftarrow}$ B, just as in an STN. For a contingent link A $\overset{[x,y]}{\Longrightarrow}$ B, we have the same two edges A $\overset{y}{\longrightarrow}$ B and A $\overset{-x}{\longleftarrow}$ B, but we also have two additional edges of the form A $\overset{b:x}{\longrightarrow}$ B and A $\overset{B:-y}{\longleftarrow}$ B. These are called *labelled edges* because of the additional "b:" and "B:" annotations indicating the contingent timepoint B with which they are associated. Note especially the reversal in the roles of x and y in the labelled edges. We refer to A $\overset{B:-y}{\longleftarrow}$ B and A $\overset{b:x}{\longrightarrow}$ B as *upper-case* and *lower-case* edges, respectively. Observe that the upper-case labelled weight B:-y gives the value the edge would have in a projection where the contingent link takes on its maximum value, whereas the lower-case labelled weight corresponds to the contingent link minimum value.

There is also a representation for a A $\xrightarrow{<\mathrm{B}, t>}$ C wait constraint in the labelled distance graph. This corresponds to a single edge A $\xleftarrow{\mathrm{B:-t}}$ C. Note the analogy to a lower bound. This weight is consistent with the lower bound that would occur in a projection where the contingent link has its maximum value.

We can now represent the tightenings in terms of the labelled distance graph. The first four categories of tightening from the classic algorithm are replaced by what is essentially a single reduction with different flavors. These are:

(Upper-Case Reduction)
A $\xleftarrow{\mathrm{B:x}}$ C $\xleftarrow{\mathrm{y}}$ D adds A $\xleftarrow{\mathrm{B:(x+y)}}$ D

(Lower-Case Reduction) If $x \leq 0$,
A $\xleftarrow{\mathrm{x}}$ C $\xleftarrow{\mathrm{c:y}}$ D adds A $\xleftarrow{\mathrm{x+y}}$ D

(Cross-Case Reduction) If $x \leq 0$, B $\neq$ C,
A $\xleftarrow{\mathrm{B:x}}$ C $\xleftarrow{\mathrm{c:y}}$ D adds A $\xleftarrow{\mathrm{B:(x+y)}}$ D

(No-Case Reduction)
A $\xleftarrow{\mathrm{x}}$ C $\xleftarrow{\mathrm{y}}$ D adds A $\xleftarrow{\mathrm{x+y}}$ D

In place of the *Unconditional* and *General Reductions*, we will have a single reduction:

(Label Removal Reduction) If $z \geq -x$,
B $\xleftarrow{\mathrm{b:x}}$ A $\xleftarrow{\mathrm{B:z}}$ C adds A $\xleftarrow{\mathrm{z}}$ C

It is shown in [5] that the new reductions are sanctioned by the old ones. For example, Upper-Case Reduction follows from a combination of *Unordered Reduction* and *Simple Regression.*

We emphasize that the Cross-Case Reduction does not apply **when the upper and lower labels come from the same contingent link**. (This case violates the B $\neq$ C precondition.) This restriction is crucial; otherwise, the upper-case and lower-case edges of any contingent link could self-interact, immediately producing an inconsistency.

With this reformulation, the "Case" (first four) reductions can all be seen as forms of composition of edges, with the labels being used to modulate when those compositions are allowed to occur. In light of this, the *reduced distance* of a path in the labelled distance graph is defined to be the sum of edge weights in the path, ignoring any labels. Thus, the reductions preserve the reduced distance.

Observe that upper-case labels can apply to new edges as a result of reductions (but the targets of the edges do not change and always point to the start node of the contingent link), whereas the lower-case edges are fixed, i.e., the reductions do not produce new ones.

The approach in [5] also modifies the test that is applied before each iteration. Instead of testing for the complex property of pseudo-controllability, it checks for ordinary consistency of the *AllMax* projection, which is defined to be the projection where all the contingent links take on their maximum values. (Similarly, the AllMin projection is where all the contingent links take on their minimum values.) Observe that the distance graph of the AllMax projection can be obtained from the labelled distance graph by (1) deleting all lower-case edges, and (2) removing the labels from all upper-case edges.

Suppose we now take the classic algorithm for Dynamic Controllability, and modify it by replacing the old reductions/regressions with the new, and replacing the pseudo-controllability test with the AllMax consistency test. This modified algorithm correctly determines DC, and furthermore, if the network is DC, quiescence is reached after at most $O(N^2)$ iterations of the outer loop [5]. Thus, the algorithm can be halted at this cutoff bound. We will refer to this as the Quadratic-Cutoff algorithm.

The algorithm can be summarized as follows.

```
Boolean procedure determineDC()
  loop from 1 to Cutoff Bound do
    if AllMax projection inconsistent
       return false;
    Perform applicable Reductions;
    if no reductions were applicable
       return true;
  end loop;
  return false;
end procedure
```

The overall algorithm runs in $O(N^5)$ time. A more precise $O(N^3K^2)$ bound is given [5] in terms of K, the number of contingent links. Note that $K \leq N$ since the end-points of contingent links are restricted to be distinct. At most KN new (upper-case) labelled edges are added to the multigraph by the reductions.

2.3 Implicit Precondition

The original derivations [3] of the reductions are in terms of a triangular network, which assumes three distinct nodes. Thus, the *Precedes* reduction has an implicit precondition, B $\neq$ C, which is not explicitly stated in [5]. Although the results there are not affected by this omission, we point out for completeness that the precondition is essential for the Precedes reduction. It can easily be seen, for example, that the network A $\overset{[2,4]}{\Longrightarrow}$ B $\xrightarrow{[0,0]}$ B has a dynamic strategy (just execute A at time 0), and hence is DC. However, without the precondition, an application of the *Precedes* reduction would produce an inconsistency. Similar remarks apply to the Lower-Case reduction (which is derived from the *Precedes* reduction) where there is an implicit A $\neq$ C precondition.

Instead of adding this precondition explicitly, we will make a different modification to the Dynamic Controllability formulation that makes it unnecessary, and has other advantages. Recall that a dynamic strategy may depend on the past, but not on the future *or present.* We will change this so that it may depend on the past **or present**. This essentially assumes that observations can be acted upon instantaneously instead of requiring an infinitesimal amount of time. This change is NOT essential to the results in this paper; they could be derived without it. However, the mathematics works out more cleanly with the change. It is also more consistent with the approach used in the dispatchability [8] work.

The effect of this change is that the LOWER-CASE and CROSS-CASE reductions must be modified to read as follows (note the $x \leq 0$ is changed to $x < 0$):

(LOWER-CASE REDUCTION) If $x < 0$,
$\mathrm{A} \xleftarrow{\mathrm{x}} \mathrm{C} \xleftarrow{\mathrm{c:y}} \mathrm{D}$ adds $\mathrm{A} \xleftarrow{\mathrm{x+y}} \mathrm{D}$

(CROSS-CASE REDUCTION) If $x < 0$, $\mathrm{B} \neq \mathrm{C}$,
$\mathrm{A} \xleftarrow{\mathrm{B:x}} \mathrm{C} \xleftarrow{\mathrm{c:y}} \mathrm{D}$ adds $\mathrm{A} \xleftarrow{\mathrm{B:(x+y)}} \mathrm{D}$

We will assume in the remainder of this paper that the LOWER-CASE and CROSS-CASE reductions have been modified in this way. The UPPER-CASE and NO-CASE reductions do not require modification.

3 Structural Characterization

We now proceed to introduce a new analysis of Dynamic Controllability that leads to a faster algorithm.

3.1 Normal Form STNU

In this subsection, we introduce a new way of simplifying the STNU formulation. First, we recall that in the definition of an STNU [3], the bounds on a contingent link $\mathrm{A} \overset{[x,y]}{\Longrightarrow} \mathrm{B}$ are required to satisfy $0 < x < y < \infty$. An analysis of the proof of correctness in [3] shows that the strict $0 < x$ inequality was only needed because of a weakness of the pseudo-controllability test in detecting a deadlock involving a cycle of waits, and the resulting use of the *General Reduction* for this purpose. In [5], the pseudo-controllability test is replaced by a test of the consistency of the AllMax projection. This can detect a cycle of waits even when contingent links are allowed to have lower bounds of zero. Thus, we can relax the contingent link bound requirement to $0 \leq x < y < \infty$.

This provides an opportunity to recognize that we can restrict our attention to a simpler subclass of STNUs without loss of generality. We will say an STNU is in *normal form* if the lower bound of every contingent link is zero. Now consider a general STNU Γ and any contingent link $\mathrm{A} \overset{[x,y]}{\Longrightarrow} \mathrm{B}$ in Γ where $x > 0$. Suppose we create a new STNU Γ' where the $\mathrm{A} \overset{[x,y]}{\Longrightarrow} \mathrm{B}$ contingent link is replaced by $\mathrm{A} \xrightarrow{[\mathrm{x,x}]} \mathrm{C} \overset{[0,\mathrm{y-x}]}{\Longrightarrow} \mathrm{B}$, where C is a new controllable timepoint. It is not difficult to see that any dynamic strategy for Γ can be easily mapped into a dynamic strategy for Γ' (just execute C at x units after A) and vice versa (just drop C).

Thus, Γ is DC if and only if Γ' is DC. The replacement process can be repeated until every contingent link with a non-zero lower-bound has been eliminated. Thus, for any STNU, there is a normal form STNU that is equivalent in terms of the existence of a dynamic strategy. We will assume in our subsequent analysis that the STNUs are in normal form.

Note that the Label Removal reduction assumes a simpler form in a normal form STNU as follows. (This facilitates our subsequent proofs.)

$$\begin{array}{l}(\text{Label Removal}) \text{ If } z \geq 0, \\ \text{A} \xleftarrow{\text{B:z}} \text{C} \quad \text{adds} \quad \text{A} \xleftarrow{\text{z}} \text{C}\end{array}$$

It is also worth commenting that with the normal form assumption, the c:x notation could be "recycled" to mean a c:0 edge followed by a path of ordinary edges of length x. This would allow the Lower-Case and Cross-Case reductions to be rewritten as follows.

$$\begin{array}{l}(\text{Lower-Case Composition}) \\ \text{A} \xleftarrow{\text{x}} \text{E} \xleftarrow{\text{c:y}} \text{D} \quad \text{adds} \quad \text{A} \xleftarrow{\text{c:(x+y)}} \text{D}\end{array}$$

$$\begin{array}{l}(\text{Lower Label Removal}) \text{ If } z < 0, \\ \text{A} \xleftarrow{\text{c:z}} \text{D} \quad \text{adds} \quad \text{A} \xleftarrow{\text{z}} \text{D}\end{array}$$

$$\begin{array}{l}(\text{Cross-Case Composition}) \text{ If B} \neq \text{C}, \\ \text{A} \xleftarrow{\text{B:x}} \text{E} \xleftarrow{\text{c:y}} \text{D} \quad \text{adds} \quad \begin{cases} \text{A} \xleftarrow{\text{B:(x+y)}} \text{D if } (x+y) < 0 \\ \text{A} \xleftarrow{\text{c:(x+y)}} \text{D if } (x+y) \geq 0 \end{cases}\end{array}$$

We will not pursue this notation change here, but it is interesting to note the essential symmetry between lower-case and upper-case behavior.

3.2 Path Transformations

An ordinary STN is consistent if and only if its distance graph does not contain a negative cycle. It is tempting to suppose that Dynamic Controllability might be characterized by the absence of cycles of negative reduced distance in the labelled distance graph. However, this is not true in general. For example, the STNU consisting of the single contingent link $\text{A}\overset{[0,4]}{\Longrightarrow}\text{B}$ is DC, but its distance graph contains the cycle $\text{A} \xrightarrow{\text{b:0}} \text{B} \xrightarrow{\text{B:}-4} \text{A}$, which has negative reduced distance. Nevertheless, as we will see, there is indeed a characterization of DC in terms of negative cycles, but it involves a subclass of such cycles. In order to describe this, we require additional concepts involving a notion of path transformation.

Consider a path $\mathcal{P}$ that contains a subpath $\mathcal{Q}$ between two points A and B and suppose $\mathcal{Q}$ matches the left side of a reduction. Note that applying the reduction to $\mathcal{Q}$ yields a new edge e between A to B. Now consider the path $\mathcal{P}'$ obtained from $\mathcal{P}$ by replacing $\mathcal{Q}$ by e. For convenience, we will abuse language slightly and say $\mathcal{P}$ is *transformed* into $\mathcal{P}'$ by the reduction. (The original path

$\mathcal{P}$ is of course still in the network.) Note that $\mathcal{P}'$ has the same reduced distance as $\mathcal{P}$ since the reductions preserve reduced distance.

Definition 1. *A path is **reducible** if it can be transformed into a single edge by a sequence of reductions. A path is **semi-reducible** if it can be transformed into a path without lower-case edges by a sequence of reductions.*

The second property is more useful for characterizing Dynamic Controllability. Recall that tests of the consistency of the AllMax projection are used to filter non-DC networks in the Quadratic Cutoff algorithm. Note also that the AllMax projection includes edge weights derived from both ordinary edges and upper-case edges, but not from lower-case edges. We may view the reductions as gradually tightening the network by transforming reduced distance in the labelled distance graph into ordinary distance in the AllMax projection. The significant events in this process are the transformations of paths with lower-case edges into paths without lower-case edges. We have the following theorem. (To simplify its statement, we will informally say an STNU has a negative cycle if its labelled distance graph contains a cyclic path with negative reduced distance.)

Theorem 1. *An STNU is Dynamically Controllable if and only if it does not have a semi-reducible negative cycle.*

Proof. If an STNU is not DC, then there is some sequence of reductions that produces a negative cycle in the AllMax projection, i.e., a lower-case-free negative cycle in the labelled distance graph. If we now unwind that sequence of reductions (applying the reverse transformations to the negative cycle), we arrive at a preimage or *precursor* cycle in the original labelled distance graph. Since the reductions preserve reduced distance, this is negative and semi-reducible.

Conversely, if there is a semi-reducible negative cycle, then clearly there is a sequence of reductions that produces an inconsistency in the AllMax projection. Thus, the STNU is not DC. □

Observe that the cycle $\mathrm{A} \xrightarrow{\mathrm{b}:0} \mathrm{B} \xrightarrow{\mathrm{B}:-4} \mathrm{A}$ in our earlier example is not semi-reducible since no reductions are applicable. (The Cross-Case reduction does not apply since the b and B labels are from the same contingent link.)

We now look for ways of identifying semi-reducible paths. The following notation will be useful. Consider a specific path $\mathcal{P}$ in the labelled distance graph of an STNU. We will write $e < e'$ in $\mathcal{P}$ if e is an earlier edge than e' in $\mathcal{P}$. If A and B are nodes in the path, we will write $\mathcal{D}_{\mathcal{P}}(\mathrm{A}, \mathrm{B})$ for the reduced distance from A to B in $\mathcal{P}$. We denote the start and end nodes of an edge e by start(e) and end(e), respectively.

Definition 2. *Suppose e is a lower-case edge in $\mathcal{P}$ and e' is some other edge such that $e < e'$ in $\mathcal{P}$. The edge e' is a **drop edge** for e in $\mathcal{P}$ if $\mathcal{D}_{\mathcal{P}}(end(e), end(e')) < 0$. The edge e' is a **moat edge** for e in $\mathcal{P}$ if it is a* drop edge *and there is no other drop edge e'' such that $e'' < e'$ in $\mathcal{P}$. In this case, we call the subpath of $\mathcal{P}$ from end(e) to end(e′) the **extension** of e in $\mathcal{P}$.*

Thus, a moat edge is a closest drop edge. (The metaphor is of a moat in front of a castle.) Note that a moat edge must be negative, hence not lower-case.

The extension subpath turns out to have a very useful property. We will say a path $\mathcal{P}$ has the *prefix/postfix property* if every nonempty proper prefix of $\mathcal{P}$ has non-negative reduced distance and every nonempty proper postfix of $\mathcal{P}$ has negative reduced distance. We will also refer to such a path as a *prefix/postfix path.* Observe that the extension subpath of a lower-case edge always has the prefix/postfix property. (Otherwise there would be a closer drop edge than the moat edge.) The following lemma will be useful.

Lemma 1 (Nesting Lemma). *If two prefix/postfix paths have a non-empty intersection, then one of the paths is contained in the other.*

Proof. The intersection subpath is a postfix of one path and a prefix of the other. It cannot be proper in both cases; otherwise it would be both non-negative and negative, which is a contradiction. Thus, it must be equal to one of the paths, which must then be a subpath of the other. □

The significance of an extension subpath, as we will see, is that it can eventually be used to "reduce away" the lower-case edge from the path. However, there is an exceptional case where we will show this cannot occur. Suppose e is a lower-case edge in a path and e' is a moat edge for e. We will say e' is *unusable* if e' is the upper-case edge from the same contingent link as e. This prepares the way for the following fundamental theorem.

Theorem 2. *A path $\mathcal{P}$ is semi-reducible if and only if every lower-case edge in $\mathcal{P}$ has a usable moat edge in $\mathcal{P}$.*

Proof. First, suppose $\mathcal{P}$ is semi-reducible. Let e be any lower-case edge in $\mathcal{P}$. Then there must be some sequence of transformations on $\mathcal{P}$ that eliminates e, i.e., e must eventually participate in a lower-case or cross-case reduction with some edge e' of weight $w < 0$ that is derived by a sequence of transformations on $\mathcal{P}$. If we unwind this sequence, we can identify a precursor subpath $\mathcal{Q}$ of $\mathcal{P}$ that will eventually be transformed to e'. Let e'' be the final edge of $\mathcal{Q}$. Since the reductions preserve reduced distance, it follows that $\mathcal{D}_{\mathcal{P}}(\text{end}(e), \text{end}(e'')) = w < 0$. Thus, e'' is a drop edge for e and hence e must have a moat edge e'''.

Next, suppose the moat edge is not usable, i.e., e''' is the upper-case edge that comes from the same contingent link as e. Note that every postfix of the extension (proper or non-proper) of e is negative. It is not hard to see that this rules out any "clearing" of the upper-case label from $\mathcal{Q}$ via the label removal reduction (as modified for a normal form STNU). This implies e' will also have that label. But this prevents application of the cross-case reduction to eliminate e, which is a contradiction. It follows that every lower-case edge in $\mathcal{P}$ has a usable moat edge.

Conversely, suppose that every lower-case edge in $\mathcal{P}$ has a usable moat edge in $\mathcal{P}$. Consider the extension subpaths corresponding to all the lower-case edges in $\mathcal{P}$. By the Nesting Lemma, these are either nested or disjoint, i.e., they fall into nested groups. We will say an extension is *innermost* if it is not contained in another extension. It is enough to show that we can transform $\mathcal{P}$ to eliminate the lower-case edges that determine the innermost extensions (which cannot contain any further lower-case edges); the result will then follow by induction since the other extensions will become innermost after the lower-case edges of the extensions nested within them have been eliminated.

Now consider any innermost extension $\mathcal{E}$ of a lower-case edge e. Since all the proper prefixes of $\mathcal{E}$ are non-negative, it follows that any upper-case labels in the interior of $\mathcal{E}$ can be "cleared" by applying no-case, upper-case and label removal reductions in a left-to-right manner. The only possible upper-case edge remaining will be the moat edge e'. Since this is usable, either e' is an ordinary edge, or e' is an upper-case edge from a different contingent link than e. Thus, $\mathcal{E}$ will eventually reduce to an e'' that is either an ordinary edge or an upper-case edge from a different contingent link than e. Since $\mathcal{E}$ has negative reduced distance, the e'' can then participate in a reduction that eliminates e. □

We can make two important observations from the converse part of the proof of theorem 2. First, the nesting property allows us to place a left-paren before each lower-case edge and a matching right paren after each corresponding moat edge; we call this the *parenthesization* of the path. The second observation is that there is a standard way of performing the transformations, using the parenthesization, that is guaranteed to eliminate the lower-case edges from a semi-reducible path. We call this the *canonical elimination.*

3.3 Complexity of Negative Cycles

Our next task is to analyze the complexity of semi-reducible negative cycles. In the case of an ordinary STN, if it has any negative cycle, then it must have a simple (without any repetitions) negative cycle. This allows the Bellman-Ford algorithm to limit the extent of its propagation. Unfortunately, a similar result does not hold for semi-reducible negative cycles in an STNU. The problem is that (if it is non-simple) there is no guarantee that one of its component cycles will also be both negative and semi-reducible, as seen in the following example. The compound cycle

$$\mathrm{B} \xrightarrow{B:-2} \mathrm{A} \xrightarrow{b:0} \mathrm{B} \xrightarrow{1} \mathrm{D} \xrightarrow{D:-3} \mathrm{C} \xrightarrow{d:0} \mathrm{D} \xrightarrow{3} \mathrm{B} \xrightarrow{B:-2} \mathrm{A} \xrightarrow{b:0} \mathrm{B} \xrightarrow{-2} \mathrm{E} \xrightarrow{4} \mathrm{B}\ ,$$

which is semi-reducible and negative, can be broken into two component cycles $\mathrm{B} \xrightarrow{B:-2} \mathrm{A} \xrightarrow{b:0} \mathrm{B} \xrightarrow{1} \mathrm{D} \xrightarrow{D:-3} \mathrm{C} \xrightarrow{d:0} \mathrm{D} \xrightarrow{3} \mathrm{B}$ and $\mathrm{B} \xrightarrow{B:-2} \mathrm{A} \xrightarrow{b:0} \mathrm{B} \xrightarrow{-2} \mathrm{E} \xrightarrow{4} \mathrm{B}$. However, the first is negative but not semi-reducible, while the second is semi-reducible but not negative. (Note that the CD edge in the first cycle has its moat edge BE in the second cycle.)

The good news is that there are nevertheless some simplifications that we can apply to a semi-reducible negative cycle, and they do lead to a faster DC checking algorithm. We require some additional concepts.

Definition 3. *A lower-case edge e in a semi-reducible path has a* ***breach*** *if its extension contains the upper-case edge from the same contingent link as e.*

Definition 4. *A repetition of a lower-case edge e in a semi-reducible path is* ***nested*** *if the extension from one occurrence contains the other.*

Note that by the Nesting Lemma, the extensions from a repetition (such as AB in the example) must be disjoint (as in the example) if they are not nested.

Theorem 3. *If an STNU has any semi-reducible negative cycle, then it has a breach-free semi-reducible negative cycle in which there are no nested repetitions.*

Proof. First, we will show that breaches can be eliminated. Consider any outermost extension $\mathcal{E}$ associated with a lower-case edge e and its moat edge e' and suppose it has a breach edge e''. Then $e'' \neq e'$. (Otherwise the moat edge would not be usable.) Thus, $\mathcal{D}_{\mathcal{P}}(\text{end}(e), \text{end}(e'')) \geq 0$ by the prefix/postfix property, and so $\mathcal{D}_{\mathcal{P}}(\text{start}(e), \text{end}(e'')) \geq 0$. Since e and e'' are the lower-case and upper-case edges of the same contingent link, start(e)=end(e''). Now observe that if we tighten the cycle by deleting the portion between start(e) and end(e''), we will not affect the moat edges of any remaining lower-case edges. (Since $\mathcal{E}$ does not lie inside any other extension.)

Now suppose by induction that we have eliminated breaches in all extensions that contain a given extension $\mathcal{E}$. We can apply the same breach elimination process as before. This may tighten some extension $\mathcal{E}'$ containing $\mathcal{E}$ such that the former moat edge for $\mathcal{E}'$ is no longer the closest drop edge. However, since $\mathcal{E}'$ has no breaches, the new moat edge will still be usable. Thus, we can eliminate the breach from $\mathcal{E}$, while preserving the property that every lower-case edge has a usable moat edge. By induction, we can eliminate all breaches while preserving this property. This leads to a new tighter (thus, still negative) cycle in which every lower-case edge still has a usable moat edge. Thus, it is still semi-reducible by theorem 2.

Next suppose we have a breach-free semi-reducible negative cycle $\mathcal{P}$, and consider a nested repetition, i.e., lower-case edges e_1 and e_2 with associated extensions $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, such that $\mathcal{E}_1$ contains $\mathcal{E}_2$, and $e_1 = e_2$. By the prefix/postfix property, $\mathcal{D}_{\mathcal{P}}(\text{start}(e_1), \text{start}(e_2)) \geq 0$. In this case, we can tighten the cycle by deleting the subpath between start(e_1) and start(e_2). Since the cycle is breach-free, every lower-case edge still has a usable moat edge (by a similar argument as previously). Thus, the cycle is still semi-reducible. □

The significance of theorem 3 is that if there are no nested repetitions, then the depth of nesting of the extensions cannot be greater than K, where K is the number of contingent links. We now fashion a DC checking algorithm that takes advantage of this. The idea is that each iteration of a propagation phase will decrement the depth of nesting by eliminating the innermost extensions. Thus, at most K iterations will be required to detect some semi-reducible negative cycle if an STNU is not DC. The propagation phase essentially simulates the canonical elimination mentioned earlier: we propagate forward from each lower-case edge over breach-free and lower-case-free paths looking for moat edges. For each one we find, we add a new edge constraint corresponding to the reduction of the lower-case edge and extension to a single edge.

The algorithm can be summarized as follows.

```
Boolean procedure fastDCcheck()
  loop from 1 to K do
    if AllMax projection inconsistent return false;
    loop for each lower-case edge e do
      Propagate forward from end(e) over allowed paths
      loop for each moat edge e' found do
        add reduced edge constraint from start(e) to end(e')
      end loop;
    end loop;
  end loop;
  if AllMax projection inconsistent return false;
  return true;
end procedure
```

We now estimate the complexity of this algorithm. For this, we let N be the number of nodes, E be the number of edges, and K be the number of contingent links. First, we observe that we need only propagate over the shortest paths among the allowed paths. (The only consequence will be possible earlier discovery of tighter reduced edges.) Second, a Bellman-Ford propagation that determines consistency of the AllMax projection can be used to provide a potential function as in Johnson's algorithm [4]. Thus, the shortest path propagations from the lower-case edges can use the $O(E + N \log N)$ Dijkstra algorithm. The overall cost of the algorithm is then $O(K(EN + K(E + N \log N))) = O(KEN + K^2E + K^2N \log N)$. At most KN new labelled edges are added to the multigraph by the reduction rules [3]. Using $K \leq N$, we thus have $E \leq O(N^2)$, and the overall complexity simplifies to $O(N^4)$, compared to the previous $O(N^5)$ algorithm.

The initial filtering of networks with coincident contingent nodes can be done in $O(E)$ time, and the normalization process at worst doubles the number of nodes, so these preprocessing steps do not alter the $O(N^4)$ complexity. Note that DC checking is at least as complex as STN consistency checking, which has an $O(N^3)$ algorithm.

4 Execution

It should be pointed out that fastDCcheck merely determines the status of an STNU. It does not provide a network suitable for the execution algorithm described in [3]. However, once DC has been confirmed, it is an easier matter to prepare the network for execution. Due to space limitations, we can only outline the approach without providing detailed proofs.

Successful execution requires that no contingent link bounds are squeezed due to propagation when a timepoint is executed or a contingent link finishes. To ensure that contingent link upper-bounds are not squeezed, we see from [3] that the key requirement is that waits need to be regressed along both ordinary and lower-case edges as far as they will go. This means that a regressWaits algorithm analogous to fastDCcheck that works backwards from upper-case edges (using Upper and Cross Case reductions), adds new waits and ordinary edges (the latter using Label Removal), and uses an AllMin propagation in each iteration to construct the potential function, can be used to regress the waits. An argument that is symmetrically related to the drop/moat edge analysis can be used to show that quiescence is reached within K iterations.

Once the waits have been regressed, we need to ensure that contingent link lower-bounds are also not squeezed. For this, we observe that propagations during execution are only along ordinary edges. (The waits merely introduce delays.) Thus, we need to ensure that paths that begin with lower-case edges and continue with ordinary edges are transformed to bypass the lower-case edges via the Lower Case reduction. This can be achieved by applying a modified fastDCcheck algorithm where the “allowed” paths are restricted to ordinary edges. With that restriction, the proof methods of this paper can be adapted to show that quiescence is reached within K iterations. It can also be shown that the edges added in this step will not disturb the quiescence of the wait regression.

Both of these post-processing steps run in similar time to fastDCcheck. Thus, the combined algorithm is still $O(N^4)$.

5 Conclusion

We have reformulated Dynamic Controllability testing in a way that provides mathematically simpler operations, characterized DC in terms of the absence of a certain type of negative cycle, and used that to obtain a linear cutoff leading to an $O(N^4)$ algorithm. Previously, only an $O(N^5)$ algorithm was known.

Acknowledgement. We thank Nicola Muscettola for discussions that contributed to these results.

References

1. Muscettola, N., Nayak, P., Pell, B., Williams, B.: Remote agent: to boldly go where no AI system has gone before. Artificial Intelligence **103**(1-2) (1998) 5–48
2. Vidal, T., Fargier, H.: Handling contingency in temporal constraint networks: from consistency to controllabilities. JETAI **11** (1999) 23–45
3. Morris, P., Muscettola, N., Vidal, T.: Dynamic control of plans with temporal uncertainty. In: Proc. of IJCAI-01. (2001)
4. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT press, Cambridge, MA (1990)
5. Morris, P., Muscettola, N.: Dynamic controllability revisited. In: Proc. of AAAI-05. (2005)
6. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial Intelligence **49** (1991) 61–95
7. Vidal, T.: Controllability characterization and checking in contingent temporal constraint networks. In: Proc. of Seventh Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'2000). (2000)
8. Tsamardinos, I., Muscettola, N., Morris, P.: Fast transformation of temporal plans for efficient execution. In: Proc. of Fifteenth Nat. Conf. on Artificial Intelligence (AAAI-98). (1998)

When Interval Analysis Helps Inter-block Backtracking

Bertrand Neveu, Gilles Chabert, and Gilles Trombettoni

COPRIN Project, INRIA, 2004 route des lucioles,
06902 Sophia.Antipolis cedex, B.P. 93, France
{neveu, gchabert, trombe}@sophia.inria.fr

Abstract. Inter-block backtracking (IBB) computes all the solutions of sparse systems of non-linear equations over the reals. This algorithm, introduced in 1998 by Bliek et al., handles a system of equations previously decomposed into a set of (small) $k \times k$ sub-systems, called blocks. Partial solutions are computed in the different blocks and combined together to obtain the set of global solutions.

When solutions inside blocks are computed with interval-based techniques, IBB can be viewed as a new interval-based algorithm for solving decomposed equation systems. Previous implementations used Ilog Solver and its IlcInterval library. The fact that this interval-based solver was more or less a black box implied several strong limitations.

The new results described in this paper come from the integration of IBB with the interval-based library developed by the second author. This new library allows IBB to become reliable (no solution is lost) while still gaining several orders of magnitude w.r.t. solving the whole system. We compare several variants of IBB on a sample of benchmarks, which allows us to better understand the behavior of IBB. The main conclusion is that the use of an interval Newton operator inside blocks has the most positive impact on the robustness and performance of IBB. This modifies the influence of other features, such as intelligent backtracking and filtering strategies.

Keywords: intervals, decomposition, solving sparse systems.

1 Introduction

Interval techniques are promising methods to compute all the solutions of a system of non-linear constraints over the reals. They are general-purpose and become more and more efficient. They have an increasing impact in several domains such as robotics [21] and robust control [12]. However, it is acknowledged that systems with hundreds (sometimes tens) non-linear constraints cannot be tackled in practice.

In several applications made of non linear constraints, systems are sufficiently sparse to be decomposed by equational or geometric techniques. CAD, scene reconstruction with geometric constraints [23], molecular biology and robotics represent such promising application fields. Different techniques can be used to decompose such systems into *$k \times k$ blocks*. Equational decomposition techniques work on the *constraint graph* made of variables and equations [2,15]. The simplest equational decomposition method computes a maximum matching of the constraint graph. The strongly connected components (i.e., the cycles) yield the different blocks, and a kind of triangular form is obtained for the system. When equations model geometric constraints, more sophisticated geometric decomposition techniques generally produce smaller blocks. They work directly on a geometric view of the entities and use a rigidity property [13,10,15].

Once the decomposition has been obtained, the different blocks must be solved in sequence. An original approach of this type has been introduced in 1998 [2] and improved in 2003 [14]. *Inter-Block Backtracking* (IBB) follows the partial order between blocks yielded by the decomposition, and calls a solver to compute the solutions in every block. IBB combines the obtained partial solutions to build the solutions of the problem.

Contributions

The new results described in this paper come from the integration of IBB with our own interval-based library.

– This new library allows IBB to become reliable (no solution is lost) while still gaining several orders of magnitude w.r.t. solving the whole system.
– An extensive comparison on a sample of CSPs allows us to better understand the behavior of IBB and its interaction with interval analysis.
– The use of an interval Newton operator inside blocks has the most positive impact on the robustness and performance of IBB. Interval Newton modifies the influence of other features, such as intelligent backtracking and filtering on the whole system (inter-block interval filtering – IBF).

2 Assumptions

We assume that the systems have a finite set of solutions. This condition also holds on every sub-system (block), which allows IBB to combine together a finite set of partial solutions. Usually, to produce a finite set of solutions, a system must contain as many equations as variables. In practice, the problems that can be decomposed are under-constrained and have more variables than equations. However, in existing applications, the problem is made square by assigning an initial value to a subset of variables called *input parameters*. The values of input parameters may be given by the user (e.g., in robotics, the degrees of freedom, determined during the design of the robot, serve to pilot it), read on a sketch, or are given by a preliminary process (e.g., in scene reconstruction [23]).

3 Description of IBB

IBB works on a **Directed Acyclic Graph** of blocks (in short **DAG**) produced by any decomposition technique. A **block** i is a sub-system containing equations and variables. Some variables in i, called **input variables** (or parameters), are replaced by values when the block is solved. The other variables are called **(output) variables**. There exists an arc from a block i to a block j iff an equation in j involves at least one input variable assigned to a "value" in i. The block i is called parent of j. The DAG implies a partial order in the solving process.

3.1 Example

To illustrate the principle of IBB, we take the 2D mechanical configuration example introduced in [2] (see Fig. 1). Various points (white circles) are connected with rigid rods (lines). Rods impose a distance constraint between two points. Point h (black circle) is attached to the rod $\langle g, i \rangle$. The distance from h to i is one third of the distance from g to i. Finally, point d is constrained to slide on the specified line. The problem is to find a feasible configuration of the points so that all constraints are satisfied. An equational decomposition method produces the DAG shown in Fig. 1-right. Points a, c and j constitute the input parameters (see Section 2).

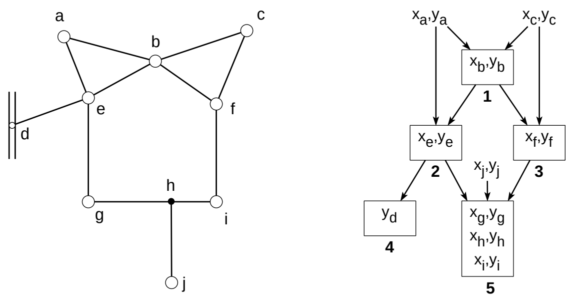


Fig. 1. Didactic problem and its DAG

3.2 Description of IBB[BT]

The algorithm IBB[BT] is a simple version of IBB based on a chronological backtracking (BT). It uses several arrays:

- solutions[i,j] is the j^{th} solution of block i.
- #sols[i] is the number of solutions in block i.
- solIndex[i] is the index of the current solution in block i (between 0 and #sols[i] -1).
- assignment[v] is the current value assigned to variable v.

Respecting the order of the DAG, IBB[BT] follows one of the induced total orders, yielded by the list blocks. The blocks are solved one by one. The procedure BlockSolve computes the solutions of blocks[i]. It stores them in solutions and computes #sols[i], the number of solutions in block i. The found solutions are assigned to block i in a combinatorial way. (The procedure assignBlock instantiates the variables in the block: it updates assignment with the values given by `solutions [i, solIndex[i] ]`.) The process continues recursively in the next block $i + 1$ until a solution for the last block is found: the values in `assignment` are then stored by the procedure `storeTotalSolution`. Of course, when a block has no (more) solution, one has to backtrack, i.e., the next solution of block $i - 1$ is chosen, if any.

The reader should notice a significant difference between `IBB[BT]` and the chronological backtracking schema used in finite CSPs. The domains of variables in a CSP are static, whereas the set of solutions of a given block may change every time it is solved. Indeed, the system of equations itself may change from a call to another because the input variables, i.e., the parameters of the equation system, may change. This explains the use of the variable `recompute` set to *true* when the algorithm goes to a block downstream.

```
Algorithm IBB[BT] (blocks: a list of blocks, #blocks: the number of blocks)
    $i \leftarrow 1$
    recompute $\leftarrow$ true
    while $i \geq 1$ do
        if recompute then
            BlockSolve (blocks, $i$, solutions, #sols)
            solIndex[$i$] $\leftarrow 0$
        end
        if solIndex[i] $\geq$ #sols[i] /* all solutions of block i have been explored */ then
            $i \leftarrow i - 1$
            recompute $\leftarrow$ false
        else
            /* solutions [$i$, solIndex[$i$] ] is assigned to block $i$ */
            assignBlock ($i$, solIndex[$i$], solutions, assignment)
            solIndex[$i$] $\leftarrow$ solIndex[$i$] +1
            if ($i$ == #blocks) /* total solution found */ then
                storeTotalSolution (assignment)
            else
                $i \leftarrow i + 1$
                recompute $\leftarrow$ true
            end
        end
    end
end.
```

Let us emphasize this point on the didactic example. `IBB[BT]` follows one total order, e.g., block 1, then 2, 3, 4, and finally 5. Calling `BlockSolve` on block 1 yields two solutions for x_b. When one replaces x_b by one of its two values in the equations of subsequent blocks (2 and 3), these equations have a different coefficient x_b. Block 2 must thus be solved twice, and with different equations, in case of backtracking, one for each value of x_b.

3.3 `BlockSolve` with Interval-Based Techniques

`IBB` can be used with any type of solver able to compute all the solutions of a system of equations (over the real numbers). In a long term, we intend to use `IBB` for solving systems of geometric constraints in CAD applications. In such applications, certain blocks will be solved by interval techniques while others, corresponding to theorems of geometry, will be solved by parametric hard-coded procedures obtained (off-line) by symbolic computation. In this paper, we consider only interval-based solving techniques, and thus view `IBB` as an interval-based algorithm for solving decomposed systems of equations.

We present here a brief introduction of the most common operators used to solve a system of equations. The underlying principles have been developed in interval analysis and in constraint programming communities.

The whole system of equations, as well as the different blocks in the decomposition, are viewed as numeric CSPs.

Definition 1. *A numeric CSP $P = (X, C, B)$ contains a set of constraints C and a set X of n variables. Every variable $x_i \in X$ can take a real value in the interval $\mathbf{x_i}$ ($B = \mathbf{x_1} \times ... \times \mathbf{x_n}$). A solution of P is an assignment of the variables in V such that all the constraints in C are satisfied.*

The n-set of intervals B can be represented by an n-dimensional parallelepiped called **box**. Because real numbers cannot be represented in computer architectures, the bounds of $\mathbf{x_i}$ are floating-point numbers. A solving process reduces the initial box until a very small box is obtained. Such a box is called an **atomic box** in this paper. In theory, an interval could be composed by two consecutive floats in the end. In practice, the process is interrupted when all the intervals have a width less than w_1, where w_1 is a user-defined parameter. It is worthwhile noting that an atomic box does not necessarily contain a solution. Indeed, evaluating an equation with interval arithmetic may prove that the equation has no solution (when the image of the corresponding box does not contain 0), but cannot assert that there exists a solution in the box. However, several operators from interval analysis can often certify that there exists a solution inside an atomic box.

Our interval-based solver uses interval-based operators to handle the blocks (`BlockSolve`). In the most sophisticated variant of `IBB`, the following three steps are iteratively performed. The process stops when an atomic box of size less than w_1 is obtained.

1. *Bisection:* One variable is chosen and its domain is split into two intervals (the box is split along one of its dimensions). This yields two smaller sub-CSPs which are handled in sequence. This makes the solving process combinatorial.
2. *Filtering/propagation:* Local information is used on constraints handled individually to reduce the current box. If the current box becomes empty, the corresponding branch (with no solution) in the search tree is cut [19,9,17].
3. *Interval analysis/unicity test:* Such operators use the first and/or second derivatives of equations. They produce a "global" filtering on the current box. If additional conditions are fulfilled, they may ensure that a unique solution exists inside the box, thus avoiding further bisection steps.

Filtering/Propagation

Propagation is performed by an AC3-like fix-point algorithm. Several types of filtering operators reduce the bounds of intervals (no gap is created in the current box). The `2B-consistency` (also known as Hull-consistency - HC) and the

`Box-consistency` [9] algorithms both consider one constraint at a time (like AC3) and reduce the bounds of the implied variables. `Box-consistency` uses an iterative process to reduce the bounds while `2B-consistency` uses projection functions. The more expensive job performed by `Box-consistency` may pay when the equations contain several occurrences of a same variable. This is not the case with our benchmarks mostly made of equations modeling distances between 2D or 3D points, and of other geometric constraints. Hence, `Box-consistency` has been discarded. The `3B-consistency` [19] algorithm uses `2B-consistency` as sub-routine and a refutation principle (shaving; similar to the Singleton Arc Consistency [5] in finite domains CSPs) to reduce the bounds of every variable iteratively. On the tested benchmarks our experiments have led us to use the `2B-consistency` operator (and sometimes `3B-consistency`) combined with an interval Newton.

Interval Analysis

We have implemented in our library two operators: the `Krawczyck` operator and the interval Newton (`I-Newton`) operator [22]. Both use an iterative numerical process, based on the first derivatives of equations, and extended to intervals. Without detailing these algorithms, it is worth understanding the output of `I-Newton`. Applied to a box B_0, `I-Newton` provides three possible answers:

1. When the jacobian matrix is not strongly regular, the process is immediatly interrupted and B_0 is not reduced [22]. This necessarily occurs when B_0 contains several solutions. Otherwise, different iterations modifies the current box B_i to B_{i+1}.
2. When B_{i+1} exceeds B_i in at least one dimension, B_{i+1} is intersected with B_i before the next iteration. No existence or unicity property can be guaranteed.
3. When the box B_{i+1} is included or equal to B_i, then B_{i+1} is guaranteed to contain a unique solution (existence and unicity test).

In the last case, when a unique solution has been detected, the convergence onto an atomic box of width w_1 in the subsequent iterations is very fast, i.e., quadratic. Moreover, the width of the obtained atomic box is often very small (even less than w_1), which highlights the significant reduction obtained in the last iteration (see Section 7.2).

Interval Techniques and Block Solving

Let us stress a characteristic of the equation systems corresponding to blocks when they are solved by interval-based techniques: the equations contain coefficients that are not scalar but (small) intervals. Indeed, the solutions obtained in a given block are atomic boxes and become parameters in subsequent blocks. For example, the two possible values for x_b in block 1 are replaced by atomic boxes in block 2. This characterictic has several consequences.

The precision sometimes decreases as long as blocks are solved in sequence. A simple example is the a 1×1 block $x^2 = p$ where the parameter p is $[0, 10^{-10}]$. Due to interval arithmetics, solving the block yields a coarser interval $[-10^{-5}, 10^{-5}]$

for x. Of course, these pathological cases related to the proximity to 0, occur occasionnally and, as discussed above, interval analysis renders the problem more seldom by sometimes producing tiny atomic boxes.

The second consequence is that it has no sense to talk about a unique solution when the parameters are not scalar and can thus take an infinite set of possible real values. Fortunately, the unicity test of `I-Newton` still holds. Generalizing the unicity test to non scalar parameters has the following meaning: if one takes *any* scalar real value in the interval of every parameter, it is ensured that exactly one point inside the atomic box found is a solution. Of course, this point changes according to the chosen scalar values. Although this proposition has not been published (to our knowledge), this is straightforward to extend the "scalar" proof to systems in which the parameters are intervals.

Remark. In the old version using the `IlcInterval` library of Ilog Solver, the unicity test was closely associated to the `Box-consistency`. It made difficult to understand why mixing `Box` and `2B` was fruitful. An interesting consequence of the integration of our interval-based solver in `IBB` is that we now know that it was not due to the `Box-consistency` itself, but related to the unicity test that avoids bisection steps in the bottom of the search tree, and to the use of the *centered form* of the equations that produces additional pruning.

4 Variants of `IBB`

Since 1998, several variants of `IBB` have been tested [2,14]. In particular, sophisticated versions have been designed to exploit the partial order between blocks. Indeed, `IBB[BT]` uses only the total order between blocks and forgets the actual dependencies between them. Figure 1-right shows an example. Suppose block 5 had no solution. Chronological backtracking would go back to block 4, find a different solution for it, and solve block 5 again. Clearly, the same failure will be encountered again in block 5.

It is explained in [2] that the *Conflict-based Backjumping* and *Dynamic backtracking* schemes cannot be used to take into account the structure given by the `DAG`. Therefore, an intelligent backtracking, called `IBB[GPB]`, was introduced, based on the *partial order backtracking* [20,2]. In 2003, we have also proposed a simpler variant `IBB[GBJ]` [14] based on the *Graph-based BackJumping* (GBJ) proposed by Dechter [6].

4.1 The Recompute Condition

In addition to intelligent backtracking schemes mentioned above, there is an even simpler way to exploit the partial order yielded by the DAG of blocks: the **recompute condition**. This condition states that it is useless to recompute the solutions of a block with `BlockSolve` if the parent variables have not changed. In that case, `IBB` can reuse the solutions computed the last time the block has been handled. In other words, when handling the next block $i + 1$, the variable `recompute` is not always set to *true*. This condition has been implemented in `IBB[GBJ]` and in `IBB[BT]`. In the latter case, the variant is named `IBB[BT+]`.

Let us illustrate how `IBB[BT+]` works on the didactic example. Suppose that the first solution of block 3 has been selected, and that the solving of block 4 has led to no solution. `IBB[BT+]` then backtracks on block 3 and the second position of point f is selected. When `IBB[BT+]` goes down again to block 4, that block should normally be recomputed from scratch due to the modification of f. But x_f and y_f are not implied in equations of block 4, so that the two solutions of block 4, which had been previously computed, can be reused. It is easy to avoid this useless computation by using the `DAG`: when `IBB` goes down to block 4, it checks that the parent variables x_e and y_e have not changed.

4.2 Inter-block Filtering (IBF)

Inter-block filtering (in short *IBF*) can be incorporated into any variant of `IBB` using an interval-based solver. The principle of *IBF* is the following. Instead of limiting the filtering process to the current block i, we have extended the scope of filtering to all the variables. More precisely, before solving a block i, one forms a subsystem extracted from the *friend blocks* F_i' of block i:

1. take the set $F_i = \{i...\#blocks\}$ containing the blocks not yet "instantiated",
2. keep in F_i' only the blocks in F_i that are connected to i in the `DAG`[1].

IBF is integrated into `IBB` in the following way. When a bisection is applied in a given block i, the filtering operators described above, i.e., `2B` and `I-Newton`, are first called inside the block. Second, *IBF* is launched on friend blocks of i.

To illustrate *IBF*, let us consider the `DAG` of the didactic example. When block 1 is solved, all the other blocks are considered by *IBF* since they are all connected to block 1. Any interval reduction in block 1 can thus possibly perform a reduction for any variable of the system. When block 2 is solved, a reduction has potentially an influence on blocks 3, 4, 5 for the same reasons. (Notice that block 3 is a friend block of block 2 that is not downstream to block 2 in the DAG.) When block 3 is solved, a reduction can have an influence only on block 5. Indeed, once blocks 1 and 2 have been removed (because they are "instantiated"), block 3 and 4 do not belong anymore to the same connected component. Hence, no propagation can reach block 4 since the parent variables of block 5, which belong to block 2, have an interval of width at most w_1 and thus cannot be reduced further.

IBF implements only a local filtering on the friend blocks, e.g., `2B-consistency` on the tested benchmarks. It turns out that `I-Newton` is counterproductive in *IBF*. First, it is expensive to compute the jacobian matrix of the whole system. More significantly, it is likely that `I-Newton` does not prune at all the search space (except when handling the last block) because it always falls in the singular case. As a rule of thumb, if the domain of one variable x in the last block contains two solutions, then the whole system will contain at least two solutions until x is bisected. This prevents `I-Newton` from pruning the search space.

[1] The orientation of the `DAG` is forgotten at this step, that is, the arcs of the `DAG` are transformed into non-directed edges, so that the filtering can also be applied on friend blocks that are not directly linked to block i.

The experiments confirm that it is always fruitful to perform a sophisticated filtering process inside blocks, whereas *IBF* (on the whole system) produces sometimes, but not always, additional gains in performance.

4.3 Mixing IBF and the Recompute Condition

Incorporating *IBF* into `IBB[BT]` is straightforward. This is not the case for the other variants of `IBB`. Reference [14] gives guidelines for the integration of *IBF* into `IBB[GBJ]`. More generally, *IBF* adds in a sense some edges between blocks. It renders the system less sparse and complexifies the recomputation condition. Indeed, when *IBF* is launched, the parent blocks of a given block i are not the only exterior cause of interval reductions inside i. The friend blocks of i have an influence as well and must be taken into account.

For this reason, when *IBF* is performed, the recompute condition is more often true. Since the causes of interval reductions are more numerous, it is more seldom the case that all of them have not changed. This will explain for instance why the gain in performance of `IBB[BT+]` relatively to `IBB[BT]` is more significant than the gain of `IBB[BT+,IBF]` relatively to `IBB[BT,IBF]`.

4.4 Two Implementations of `IBB`

A simple variant of `IBB`, called `BB`, is directly implemented in our interval-based solver. `BB` handles the entire system of equations as a numeric CSP, and uses a bisection heuristics based on blocks. The heuristics can only choose the next variable to be split inside the current block i. The specific variable inside the block i is chosen with a standard *round robin* strategy. Since a total solution is computed with a depth-first search in a standard interval-based solving, the recompute condition cannot be incorporated. Subsystems corresponding to blocks must be created to prune inside the blocks. Filtering on the whole system implements `IBF` in a simple way and produces the `BB[BT,IBF]` version. Otherwise we get the simple `BB[BT]` version.

It appears that `BB` is not robust against the **multiple solutions** problem that often occurs in practice. With interval solving, multiple solutions occur when several atomic boxes are close to each other: only one of them contains a solution and the others are not discarded by filtering. Even when the number of multiple solutions is small, `BB` explodes because of the multiplicative effect of the blocks (the multiple partial solutions are combined together). In order that this problem occurs more rarely, one can reduce the precision (i.e., enlarge w_1) or mix several filtering techniques together. The use of interval analysis operators like `I-Newton` is also a right way to fix most of the pathological cases (see experiments).

Our second implementation, called `IBB`, does not explode because it takes the union of the multiple solutions (i.e., the hull of the solutions). `IBB` considers the different blocks separetely, so that all the solutions of a block can be computed before solving the next one. This allows `IBB` to merge multiple solutions together. This implementation is completely independent from the interval-based solver and allows more freedom in the creation of new variants (those with the prefix IBB in their name). Intelligent backtracking schemes and the recompute condition can be incorporated. However, as compared to BB, a slight overcost is sometimes caused by the explicit management of the friend blocks. Note that the previous implementations of IBB might lose some solutions because only one of the multiple partial solutions inside blocks was selected (i.e., no hull was performed) and might lead to a failure in the end when the selected atomic box did not contain a solution.

5 New Contributions

The integration of our interval-based solver underlies three types of improvements of IBB. As previously mentioned, using a white box allows us to better understand what happens. Second, IBB is now reliable. The multiple solutions problem has been handled and an ancient **midpoint heuristics** is now abandoned. This heuristics replaced every parameter, i.e., input variable, of a block by a floating-point number in the "middle" of its interval. This heuristics was necessary because the `IlcInterval` library previously used did not allow the use of interval coefficients. Our new solver accepts non scalar coefficients so that no solution is lost anymore, making thus IBB reliable. The midpoint heuristics would however allow the management of sharper boxes, but the gain in running time would be less than 5% in our benchmarks. The price of reliability is not so high!

Finally, as shown in the experiments reported below, the most significant impact on IBB is due to the integration of an interval Newton inside the blocks. `I-Newton` has a good power of filtering, can often reach the finest precision (which is of great interest due to the multiplicative effect of the blocks) and often certifies the solutions. Hence, the combinatorial explosion due to multiple solutions has been drastically limited. Moreover, the use of `I-Newton` alters the comparison between variants. In particular, in the previous versions, we concluded that *IBF* was counterproductive, whereas it is not always true today. Also, the interest of intelligent backtracking algorithms is clearly put into question, which confirms the intuition shared by the constraint programming community that a better filtering removes backtracks (and backjumps). Moreover, since `I-Newton` has a good filtering power, obtaining an atomic box requires less bisections. Hence, the number of calls to *IBF* is reduced in the same proportion.

6 Benchmarks

Exhaustive experiments have been performed on 10 benchmarks made of geometric constraints. They compare different variants of IBB and show a clear improvement w.r.t. solving the whole system.

Some benchmarks are artificial problems, mostly made of quadratic distance constraints. `Mechanism` and `Tangent` have been found in [16] and [3]. `Chair` is a realistic assembly made of 178 equations induced by a large variety of geometric constraints: distances, angles, incidences, parallelisms, orthogonalities.

The DAGs of blocks for the benchmarks have been obtained either with an equational method (abbrev. equ.) or with a geometric one (abbrev. geo.). `Ponts`

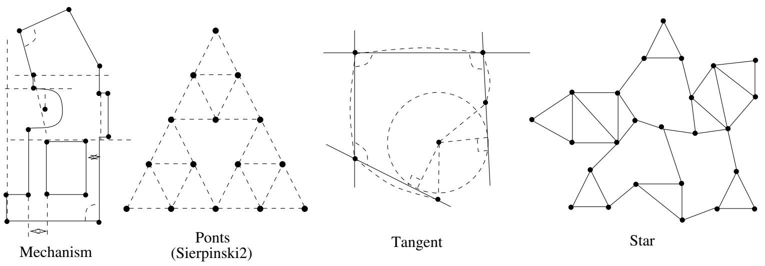


Fig. 2. 2D benchmarks: general view

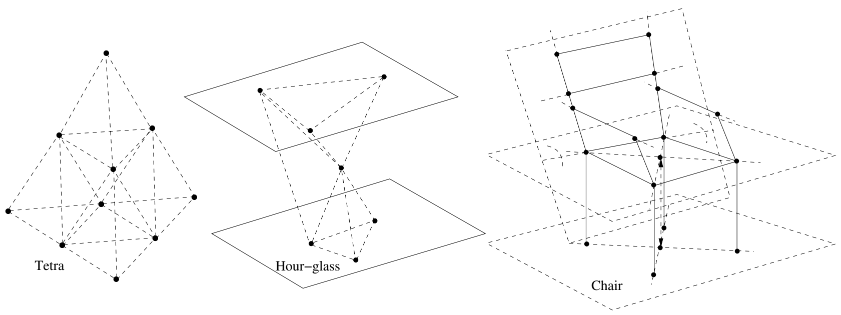


Fig. 3. 3D benchmarks: general view

and `Tangent` have been decomposed by both techniques. A problem defined with a domain of width 100 is generally similar to assigning $]-\infty, +\infty[$ to every variable. The intervals in `Mechanism` and `Sierp3` have been selected around a given solution in order to limit the total number of solutions. In particular, the equation system corresponding to `Sierp3` would have about 2^{40} solutions, so that the initial domains are limited to a width 1. `Sierp3` is the fractal *Sierpinski* at level 3, that is, 3 Sierpinski at level 2 (i.e., `Ponts`) put together. The time

Table 1. Details on the benchmarks. Type of decomposition method (Dec.); number of equations (Size); Size of blocks: NxK means N blocks of size K; Interval widths of variables (Dom.); number of solutions (#sols); bisection precision, i.e., domain width under which bisection does not split intervals (w_1).

GCSP	Dim.	Dec.	Size	Size of blocks	Dom.	#sols	w_1
`Mechanism`	2D	equ.	98	98 = 1x10, 2x4, 27x2, 26x1	10	448	5.10^{-6}
`Sierp3`		geo.	124	124 = 44x2, 36x1	1	198	10^{-8}
`PontsE`		equ.	30	30 = 1x14, 6x2, 4x1	100	128	10^{-8}
`PontsG`		geo.	38	38 = 13x2, 12x1	100	128	10^{-8}
`TangentE`		equ.	28	28 = 1x4, 10x2, 4x1	100	128	10^{-8}
`TangentG`		geo.	42	42 = 2x4, 11x2, 12x1	100	128	10^{-8}
`Star`		equ.	46	46 = 3x6, 3x4, 8x2	100	128	10^{-8}
`Chair`	3D	equ.	178	178 = 1x15,1x13,1x9,5x8,3x6,2x4,14x3,1x2,31x1	100	8	5.10^{-7}
`Tetra`		equ.	30	30 = 1x9, 4x3, 1x2, 7x1	100	256	10^{-8}
`Hourglass`		equ.	29	29 = 1x10, 1x4, 1x3, 10x1	100	8	10^{-8}

spent for the equational and geometric decompositions is always negligible, i.e., a few milliseconds for all the benchmarks.

7 Experiments

We have applied several variants of IBB on the benchmarks described above. All the tests have been conducted using the interval-based library implemented in C++ by the second author [4]. The hull consistency (i.e., 2B-consistency) is implemented with the famous HC4 that builds a syntactic tree for every constraint [1,17]. A "width" parameter w_2 equal to 10^{-4} has been chosen: a constraint is not pushed in the propagation queue if the projection on its variables has reduced the corresponding intervals less than w_2. I-Newton is run when the largest interval in the current box is less than 10^{-2}. Two atomic boxes are merged iff a unique solution has not been certified inside both and the boxes are sufficiently close to each other, that is, for every variable, there is a distance *dist* less than 10^{-2} between the two boxes (10^{-4} for the benchmark Star). Most of the reported CPU times have been obtained on a Pentium IV 3 Ghz.

7.1 Interest of System Decomposition

The first results show the drastic improvement due to IBB as compared to four interval-based solvers tuned to obtain the best results on the benchmarks. All the solvers use a round-robin splitting strategy. Ilog Solver [11] uses a filtering process mixing 2B-consistency and Box-consistency. The relatively bad CPU times simply show that the IlcInterval library has not been improved for several years. They have been obtained on a Pentium IV 2.2 Ghz.

Table 2. Interest of IBB. The columns in the left report the times (in seconds) spent by three interval-based solvers to handle the systems globally. *Home* corresponds to our own library. The column *Best* IBB reports the best time obtained by IBB. Gains of 1, 2, 3 or 4 orders of magnitude are highlighted by the column *Home/(Best* IBB*)*. The last column reports the size of the obtained atomic boxes. The obtained precision is often good (as compared to the specified parameter w_1 – see Table 1) thanks to the use of I-Newton. An entry XXX means that the solver is not able to isolate solutions.

GCSP	Home	RealPaver	Ilog Solver	Best IBB	Home/(Best IBB)	Precision
Mechanism	85	256	> 4000	1.37	62	2.10^{-5}
Sierp3	1426	> 4000	> 4000	1.18	1208	4.10^{-11}
Chair	> 4000	> 4000	$>$ 128 equations	0.5	> 8000	10^{-7}
Tetra	461	98^*	> 4000	0.92	501	2.10^{-14}
PontsE	10.3	9.5	103	0.97	10	7.10^{-14}
PontsG	5.6	15.7	294	0.31	18	10^{-13}
Hourglass	15.3	12.6^*	247	0.4	38	10^{-13}
TangentE	58	25.9^*	191	0.1	580	5.10^{-14}
TangentG	2710	2380	XXX	0.09	30000	4.10^{-14}
Star	2381	237^*	1451	0.08	30000	2.10^{-11}

RealPaver [7,8] and our library use a HC4+Newton filtering and obtain in this case comparable performances. RealPaver uses HC4+Newton+weak3B on several benchmarks (entries with a *) and obtains better results. This highlights the interest of the *weak 3B-consistency* [7]. Our library uses a HC4+Newton+3B filtering on Hourglass.

We have also applied the Quad operator [18] that is sometimes very efficient to solve polynomial equations. This operator appears to be very slow on the tested benchmarks.

7.2 Main Results

Tables 3, 4 and 5 report the main results we have obtained on several variants of IBB. Table 3 first shows that the different variants obtain similar CPU time results provided that HC4+Newton filtering is used. The conclusion is different otherwise, as shown in Table 4.

– BT+ *vs* BT*:* BT+ always reduces the number of bisections and the number of recomputed blocks. The reason why BB[BT,IBF] is better in time than

Table 3. Variants of IBB with HC4 + I-Newton filtering. Every entry contains three values: (top) the CPU time for obtaining all the solutions (in hundredths of second); (middle) the total number of bisections performed by the interval solver; (bottom) the total number of times BlockSolve is called.

	1	2	3	4	5	6
GCSP	IBB[BT]	BB[BT]	IBB[BT+]	IBB[BT,IBF]	BB[BT,IBF]	IBB[BT+,IBF]
Mechanism	137	188	**132**	176	243	172
	6186	6186	6101	6164	6808	6142
	1635	1635	1496	1629	1629	1570
Sierp3	284	228	177	269	**118**	234
	19455	19455	9537	2874	2874	2136
	21045	21045	11564	3671	3671	3035
Chair	50	51	**18**	107	86	76
	3814	3814	1176	3806	3806	2329
	344	344	97	344	344	148
Tetra	100	125	**92**	136	137	123
	3936	3936	3391	3942	3942	3397
	235	235	99	235	235	99
PontsE	**98**	**97**	**97**	121	122	120
	3091	3091	3046	3072	3072	3031
	131	131	86	115	115	74
PontsG	117	96	83	50	**31**	51
	8415	8415	5524	1303	1303	1303
	9283	9283	6825	1011	1011	1011
Hourglass	40	41	40	43	45	42
	995	995	995	994	994	994
	19	19	15	19	19	19
TangentE	**11**	**10**	**10**	19	16	19
	186	186	172	186	186	186
	427	427	405	427	427	427
TangentG	11	13	**9**	20	17	20
	402	402	83	402	402	402
	411	411	238	411	411	396
Star	28	25	17	18	18	**8**
	1420	1420	584	479	479	90
	457	457	324	251	251	31

IBB[BT+,IBF] on Sierp3 and PontsG is the more costly management of friend blocks in the IBB implementation (see Section 4.4).

– IBB *implementation vs* BB*:* Table 4 clearly shows the combinatorial explosion of BB involved by the multiple solution problem. The use of I-Newton limits this problem, except for Mechanism (see Table 3, columns 4 and 5). However, it is important to explain that the problem needed also to be fixed by manually selecting an adequate value for the parameter w_1. In particular, BB undergoes a combinatorial explosion on Chair and Mechanism when the precision is higher (i.e., when w_1 is smaller). On the contrary, the IBB implementation automatically adjusts w_1 in every block according to the width of the largest input variable (parameter) interval. This confirms that the IBB implementation is rarely worse in time than the BB one, is more robust and it can add the recompute condition.

– *Moderate interest of* IBF*:* Table 4 shows that IBF is not useful when only HC4 is used to prune inside blocks. As explained at the end of Section 5, when I-Newton is also called to prune inside blocks, IBF becomes sometimes useful: three instances benefit from the inter-block filtering (see Table 3). Moreover, IBF avoids using sophisticated backtracking schemas, as shown in the last table.

– *No Interest of intelligent backtracking:* Table 5 reports the only two benchmarks for which backjumps actually occur with an intelligent backtracking. It shows that the gain obtained by an intelligent backtracking (IBB[GBJ], IBB[GPB]) is compensated by a gain in filtering with IBF. The number of backjumps is drastically reduced by the use of IBF (6 $\rightarrow$ 0 on Star; 2974 $\rightarrow$ 135 on Sierp3). The times obtained with IBF are better than or equal to those obtained with intelligent

Table 4. Variants of IBB with HC4 filtering. The last column highlights the rather bad precision obtained.

GCSP	IBB[BT]	BB[BT]	IBB[BT+]	IBB[BT,IBF]	BB[BT,IBF]	IBB[BT+,IBF]	Precision
	281		277	361	890	354	
Mechanism	52870	XXX	52711	40696	40696	40650	7.10^{-4}
	1635		1496	1629	1629	1570	
	326		201	306		260	
Sierp3	116490	XXX	64538	15594	XXX	12228	7.10^{-6}
	21045		11564	3671		3035	
	50	10^4	17	110		82	
Chair	4408		1385	4316	XXX	2468	7.10^{-6}
	344		97	344		148	
	347		336	506		484	
Tetra	47406	XXX	42420	28140	XXX	24666	6.10^{-6}
	235		101	235		100	
	843		839	1379		1365	
PontsE	96522	XXX	95949	66033	XXX	65630	3.10^{-7}
	131		86	115		74	
	117		134	69		70	
PontsG	8415	XXX	51750	7910	XXX	7910	7.10^{-7}
	9283		9283	1011		1011	
	130	879	132	221	465	220	
Hourglass	12308		12308	11599	22007	11599	4.10^{-7}
	19		15	19	39	19	

Table 5. No interest of intelligent backtracking

	1	2	3	4	5	6	7	8
GCSP	IBB[BT]	IBB[BT+]	IBB[GBJ]	IBB[GPB]	BB[BT,IBF]	IBB[BT,IBF]	IBB[BT+,IBF]	IBB[GBJ,IBF]
Sierp3	284	177	134	> 64 blocks	**118**	269	234	230
	19455	9537	6357		2874	2874	2136	2088
	21045	11564	7690		3671	3671	3035	2867
			BJ=2974					BJ=135
Star	28	17	15	**8**	18	18	**8**	**8**
	1420	584	536	182	479	479	90	90
	457	324	277	64	251	251	31	31
			BJ=6					BJ=0

backtracking schemas. Only a marginal gain is obtained by IBB[GBJ,IBF] w.r.t. IBB[BT+,IBF] for Sierp3.

Remarks

IBB can often certify solutions of a decomposed system. A straightforward induction ensures that a solution is certified iff all the corresponding partial solutions are certified in every block. Only solutions of Mechanism and Chair have not been certified.

8 Conclusion

IBB[BT+] is often the best tested version of IBB. It is fast, simple to implement and robust (IBB implementation). Its variant IBB[BT+,IBF] can advantageously replace a sophisticated backtracking schema in case some backjumps occur.

Anyway, the three tables above clearly show that the main impact on robustness and performance is due to the mix of local filtering and interval analysis operators inside blocks. To complement the analysis reported in Section 5, a clear indication is the good behavior of simple versions of IBB, i.e., IBB[BT] and BB[BT,IBF], when such filtering operators are used (see Table 3). Tables 4 and 5 show that the influence of IBF or intelligent backtracking is less significant.

Apart from minor improvements, IBB is now mature enough to be used in CAD applications. Promising research directions are the computation of sharper jacobian matrices (because, in CAD, the constraints belong to a specific class) and the design of solving algorithms for equations with non scalar coefficients.

References

1. F. Benhamou, F. Goualard, L. Granvilliers, and J-F. Puget. Revising Hull and Box Consistency. In *ICLP*, pages 230–244, 1999.
2. C. Bliek, B. Neveu, and G. Trombettoni. Using Graph Decomposition for Solving Continuous CSPs. In *Proc. CP'98, LNCS 1520*, pages 102–116, 1998.
3. W. Bouma, I. Fudos, C.M. Hoffmann, J. Cai, and R. Paige. Geometric constraint solver. *Computer Aided Design*, 27(6):487–501, 1995.
4. G. Chabert. *Contributions à la résolution de Contraintes sur intervalles ?* Phd thesis, Université de Nice–Sophia Antipolis, 2006. (to be defended).

5. R. Debruyne and C. Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. of IJCAI*, pages 412–417, 1997.
6. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.
7. L. Granvilliers. `RealPaver` *User's Manual, version 0.3.* University of Nantes, 2003. Available at `www.sciences.univ-nantes.fr/info/perso/permanents/granvil/realpaver`.
8. L. Granvilliers. Realpaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, , Accepted for publication.
9. Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica : A Modeling Language for Global Optimization.* MIT Press, 1997.
10. C. Hoffmann, A. Lomonossov, and M. Sitharam. Finding solvable subsets of constraint graphs. In *Proc. Constraint Programming CP'97*, pages 463–477, 1997.
11. ILOG, Av. Galliéni, Gentilly. *Ilog Solver V. 5, Users' Reference Manual*, 2000.
12. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis.* Springer-Verlag, 2001.
13. C. Jermann, B. Neveu, and G. Trombettoni. Algorithms for Identifying Rigid Subsystems in Geometric Constraint Systems. In *Proc. IJCAI*, pages 233–38, 2003.
14. C. Jermann, B. Neveu, and G. Trombettoni. Inter-Block Backtracking: Exploiting the Structure in Continuous CSPs. In *Proc. of* 2^{nd} *Int. Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS'03)*, 2003.
15. C. Jermann, G. Trombettoni, B. Neveu, and P. Mathis. Decomposition of Geometric Constraint Systems: a Survey. *Int. Journal of Computational Geometry and Applications (IJCGA)*, 16, 2006.
16. R.S. Latham and A.E. Middleditch. Connectivity analysis: A tool for processing geometric constraints. *Computer Aided Design*, 28(11):917–928, 1996.
17. Y. Lebbah. *Contribution à la résolution de Contraintes par Consistance Forte.* Phd thesis, Université de Nantes, 1999.
18. Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J.P. Merlet. Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis*, 42(5):2076–2097, 2005.
19. O. Lhomme. Consistency Tech. for Numeric CSPs. In *IJCAI*, pages 232–238, 1993.
20. D.A. McAllester. Partial order backtracking. Research Note, Artificial Intelligence Laboratory, MIT, 1993. ftp://ftp.ai.mit.edu/people/dam/dynamic.ps.
21. Jean-Pierre Merlet. Optimal design for the micro robot. In *IEEE Int. Conf. on Robotics and Automation*, 2002.
22. A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.
23. M. Wilczkowiak, G. Trombettoni, C. Jermann, P. Sturm, and E. Boyer. Scene Modeling Based on Constraint System Decomposition Tech. In *Proc. ICCV*, 2003.

Randomization in Constraint Programming for Airline Planning

Lars Otten[1], Mattias Grönkvist[1,2], and Devdatt Dubhashi[1]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology, 41296 Gothenburg, Sweden
mail@lotten.net, dubhashi@cs.chalmers.se
[2] Jeppesen (Carmen Systems AB),
Odinsgatan 9, 41103 Gothenburg, Sweden
mattias.gronkvist@jeppesen.com

Abstract. We extend the common depth-first backtrack search for constraint satisfaction problems with randomized variable and value selection. The resulting methods are applied to real-world instances of the tail assignment problem, a certain kind of airline planning problem. We analyze the performance impact of these extensions and, in order to exploit the improvements, add restarts to the search procedure. Finally computational results of the complete approach are discussed.

1 Introduction

Constraint programming has received increasing attention in a multitude of areas and applications and has successfully been incorporated into a number of commercial systems.

Among other things it has been deployed in planning for transportation: Grönkvist [7] uses constraint programming to solve the tail assignment problem, an airline planning variant. Gabteni and Grönkvist [3] combine this with techniques from integer programming, in order to obtain a complete solver and optimizer, which is currently in use at a number of medium-sized airlines.

In Gabteni and Grönkvist's work, special constraints that internally employ the pricing routine of a column generation integer programming solver are used to tackle arising computational issues. In this paper, however, we set out to take an orthogonal approach, which is not dependent on column generation but rather relies on pure constraint programming techniques. Furthermore, in contrast to the usage of specialized propagation algorithms, our approach is generic at heart, hence it is more easily adaptable to problems other than tail assignment.

Recently Gomes et al. [5] have made considerable progress in exploiting the benefits of randomization in backtrack search for constraint programming, they also applied their findings to a number of more or less artificially constructed problem instances.

We pursue this randomized approach with the tail assignment problem, for which we have access to a number of real-world instances. We show how this helps in overcoming the performance bottlenecks mentioned in [3,7].

Our contribution is as follows: We review and present a number of new generic randomized schemes for variable and value selection for backtrack search. We discuss how these techniques enhance performance for the tail assignment problem. Finally, we explain how different restart strategies can systematically make use of the improvements. We then demonstrate that this does in fact work very well for practical purposes in real-life problem instances.

We introduce the tail assignment problem in the general context of airline planning in Sect. 2 and give a formulation as a constraint programming problem. Section 3 describes the randomized extensions to the backtrack search procedure and gives some first computational results. In Sect. 4 we present restart techniques as a way to exploit the benefits of randomization and discuss the respective performance results on our real-world instances. Section 5 concludes and outlines future research directions.

2 Problem Description

The process of airline planning is commonly divided into several isolated steps: During *timetable creation* a schedule of flights (referred to as *legs*) is assembled for a certain time interval. Given such an airline timetable, *fleet assignment* usually means determining which type of aircraft will cover which timetable entry while maximizing revenue. The *aircraft routing* process then assigns individual aircraft to the elements of the schedule for each fleet (or subfleet), mainly with focus on maintenance feasibility. This, in turn, is followed by *crew rostering*, which selects the required personnel for each flight leg.

In practice all these steps will be subject to a number of operational constraints (aircraft range limitations, noise level restrictions at certain airports etc.) and optimization criteria, for example minimizing the resulting costs or keeping the aircraft deployment as evenly distributed as possible.

Since airline planning has received widespread attention in a multitude of respects, there is a lot of related work. For lack of space we refer to Grönkvist [8] for a comprehensive overview and limit ourselves to two references here:

Gopalan and Talluri [6] give a general survey of several of the common problems and methods and Barnhart et al., for instance, discuss combined fleet assignment / aircraft routing in [2].

2.1 The Tail Assignment Problem

The problem of *tail assignment* denotes the process of assigning aircraft (identified by their *tail number*) to each leg of a given airline timetable. As a result one obtains a *route* for each aircraft, consisting of a sequence of legs. Hence tail assignment essentially combines fleet assignment and aircraft routing as describe before. There are several practical, operational advantages inherent to this approach, for details we refer to Gabteni and Grönkvist [3].

In their work they make use of a combined solution approach, employing techniques from both integer programming (in particular column generation) and constraint programming.

The motivation for this is that constraint programming usually finds a feasible but not necessarily optimal solution rather quickly, whereas column generation converges slowly but ensures optimality. We will henceforth focus on the constraint programming component.

2.2 A Constraint Programming Model

We now formulate the tail assignment problem as a constraint satisfaction problem (CSP). First we note that, instead of flight legs and aircraft, we will speak of *activities* and *vehicles*, which is more general and allows us for example to include scheduled maintenance into the problem. We then start by defining the following:

$$F = \{f_1, \ldots, f_n\}\,, \text{ the set of all activities.}$$
$$T = \{t_1, \ldots, t_m\}\,, \text{ the set of all vehicles.}$$

We will think of each activity as a node in an *activity network*: Initially each vehicle is assigned a unique start activity. Each activity will eventually be connected to the two activities pre- and succeeding it. Thus each vehicle's route becomes a path through the network. In fact, since we will connect a route's end activity to its start activity, we obtain exactly m cycles, one for each vehicle.

Now, to capture this within a CSP, we introduce a number of variables: For all $f \in F$ we have $\texttt{successor}_f$ with domain $D(\texttt{successor}_f) \subseteq F$ initially containing all activities possibly succeeding activity f. Equivalently, for all $f \in F$, we have $\texttt{vehicle}_f$ with domain $D(\texttt{vehicle}_f) \subseteq T$ initially containing all the vehicles that are in principle allowed to operate activity f.

Note that with the proper initial domains we cover a lot of the problem already. For example we can be sure that only legally allowed connections between activities will be selected in the final solution. Moreover we can implement a preassigned activity (like maintenance for a specific vehicle) by initializing the respective `vehicle` variable with an unary domain.

Also observe that any solution to the tail assignment problem can be represented by a complete assignment to either all `successor` or `vehicle` variables – we can construct any vehicle's route by following the `successor` links from its start activity, or we can group all activities by their `vehicle` value and order those groups chronologically to obtain the routes.

To obtain stronger propagation behavior later on we also introduce a third group of variables, similar to the `successor` variables: For all $f \in F$ we have $\texttt{predecessor}_f$ with domain $D(\texttt{predecessor}_f) \subseteq F$ initially containing all activities possibly preceding activity f.

Introducing constraints to our model, we first note that we want all routes to be disjoint – for instance two different activities should not have the same succeeding activity. Hence, as a first constraint, we add a global `alldifferent` over all `successor` variables.

Moreover, since the `successor` and `predecessor` are conceptually inverse to each other, we add a global constraint `inverse(successor,predecessor)` to our model, which is in practice implemented by means of the respective number of binary constraints and defined likewise:

$$\forall\, i, j : \quad f_i \in D(\texttt{successor}_j) \Longleftrightarrow f_j \in D(\texttt{predecessor}_i) \ .$$

This also implicitly ensures disjointness with respect to the `predecessor` variables, hence we need not add an `alldifferent` constraint over those.

Finally, to obtain a connection with the `vehicle` variables, we define another global constraint, which we call `tunneling`. It observes all variable domains and, each time a variable gets instantiated, posts other constraints according to the following rules (where $\texttt{element}(a, \texttt{B}, c)$ requires the value of $\texttt{B}_a$ to be equal to c):

$$\begin{array}{rcl}
\texttt{vehicle}_f == t & \Rightarrow & \texttt{POST element}(\texttt{successor}_f, \texttt{vehicle}, t) \\
 & & \texttt{POST element}(\texttt{predecessor}_f, \texttt{vehicle}, t) \\
\texttt{successor}_f == f' & \Rightarrow & \texttt{POST vehicle}_f = \texttt{vehicle}_{f'} \\
\texttt{predecessor}_f == f' & \Rightarrow & \texttt{POST vehicle}_f = \texttt{vehicle}_{f'}
\end{array}$$

These constraints already suffice to model a basic version of the tail assignment problem as described above, with slightly relaxed maintenance constraints. Still we will in practice add a number of constraints to improve propagation and thus computational performance: For example we can add `alldifferent` constraints over `vehicle` variables of activities that overlap in time (for certain cleverly picked times).

When solving this CSP we will only branch on the `successor` variables, meaning only these are instantiated during search. We do this because `successor` modifications are propagated well thanks to the consistency algorithm of the `alldifferent` constraint imposed on them.

2.3 Remarks

The CSP modeled above is essentially what Gabteni and Grönkvist [3] refer to as CSP-TAS^{relax} – "relax" since it does not cover some of the more complicated maintenance constraints. Still this model is used in the final integrated solution presented in [3].

What is of interest to us, however, is that for this model Gabteni and Grönkvist [3] report problems with excessive *thrashing* for problem instances containing several different types of aircraft (and thus more flight restrictions inherent in the initial `vehicle` variable domains). Thrashing means that the search procedure spends large amounts of time in subtrees that do not contain a solution, which results in a lot of backtracking taking place and consequently very long search times.

Gabteni and Grönkvist [3] try to resolve this by extending the model and introducing additional constraints, for which they implement strong propagation algorithms that perform elaborate book-keeping of reachable activities and subroutes and thereby are able to rule out certain parts of the search tree in advance.

It is worth noting that these propagation algorithms make use of the pricing routine of a column generation system from the area of integer programming. The resulting extended model is then referred to as CSP-TAS.

However, with the results of Gomes et al. [5] in mind, we take a different and, as we believe, more general approach to reduce thrashing, modifying the backtrack search procedure itself while leaving the model CSP-TAS^{relax} unchanged. Our solution also supersedes the use of elements from integer programming, which are often not readily available for a CSP but require additional effort.

On another note we should point out that, although in principle tail assignment depicts an optimization problem, in this paper we neglect the aspect of solution quality and focus on finding any one solution. Experience shows that computing any solution is often sufficient, especially if it can be achieved quickly. Moreover, even in situations where one wants a close-to-optimal solution, finding any feasible solution is useful as a starting point for improvement heuristics and as a proof that a solution exists.

3 Randomizing Backtrack Search

Standard backtrack search for constraint programming interleaves search with constraint propagation: The constraints are propagated, one search step is performed, the constraints are propagated and so on. In case of a failure, when the CSP becomes inconsistent, i. e. impossible to solve, we rollback one or several search steps and retry.

Our focus is on the search step: Generally one starts with choosing the next variable to branch on; after that one of the values from the variable's domain is selected, to which the variable is then instantiated. Both these choices will be covered separately. A number of ideas have already been introduced by Gomes et al. [4,5]; we will briefly review their findings and adapt the concept to our problem by introducing enhanced randomized selection schemes.

3.1 Variable Selection

There exist several common, nonrandomized heuristics for finding the most promising variable to branch on: For example we pick the variable with the smallest domain size. We will call this scheme *min-size*, it is sometimes also known as *fail-first*.

Alternatively we choose the variable with the smallest degree, where the degree of a variable is the number of constraints related to it; this heuristic will be referred to as *min-degree*. Grönkvist [7] uses *min-size*, yet we performed our tests with both heuristics.

In case of ties, for instance when two or more variables have the same minimal domain size, these heuristics make a deterministic choice, for example by lexicographical order of the variables. This already suggests a straightforward approach to randomize the algorithm: When we encouter ties we pick one of the candidates at random (uniformly distributed).

With sophisticated heuristics it can happen that only one or very few variables are in fact assigned the optimal heuristic value, meaning that the random tie-breaking will have little or no effect after all. To alleviate this, Gomes et al. suggest a less restrictive selection process, where the random choice is made between all optimal variables and those whose heuristic value is within an $H\,\%$ range of the optimum.

For our problem, however, we found that the said situation rarely occurs, and in fact the less restrictive $H\,\%$ rule had a negative effect on search performance.

Instead we tried another similar but more general variable selection scheme: For a certain base $b \in \mathbb{R}$, we choose any currently unassigned variable x with a probability proportional to the value $b^{-s(x)}$, where $s(x)$ is the current size of the variable's domain. Observe that for increasing values of b we will gradually approach the *min-size* scheme with random tie-breaking as described above.

3.2 Value Selection

Once we have determined which variable to branch on, the simplest way of adding randomization for the value selection is to just pick a value at random (again uniformly distributed), which is also suggested by Gomes et al. [4]. This already works quite well, yet for the problem at hand we developed something more specific.

As a nonrandomized value ordering heuristic for the tail assignment problem, Grönkvist [7] suggests to choose the successor activities by increasing connection time (recall that we only branch on the `successor` variables). Hence in our model we order the activities by increasing start time, so that obtaining the shortest possible connection time is equivalent to selecting the smallest value first.

We then take this idea as an inspiration for the following randomized value selection scheme: Assuming a current domain size of n possible values, we pick the smallest value (representing the shortest possible connection time) with probability $p \in (0,1)$, the second smallest with probability $p \cdot q$, the third smallest with probability $p \cdot q^2$ and so on, for a $q > 0$; in general, we choose the i-th smallest with probability $p \cdot q^{i-1}$.

Having this idea in mind, we note the following: Given n and either p or q, we can compute q or p, respectively. To do so we take the sum over all elements' probabilities, which has to be 1 for a valid probability distribution. The resulting equation can then be solved for either p or q:

$$1 \overset{!}{=} p + pq + pq^2 + \ldots + pq^{n-1} = p \sum_{i=0}^{n-1} q^i = p \cdot \frac{1-q^n}{1-q}$$

Obviously, if we set $q = 1$ we obtain the uniform distribution again. With $q \in (0,1)$ we assign the highest probability to the smallest value in the domain, whereas $p > 1$ gives preference to the hightest value. Trying different values, we eventually settled with $q = 0.3$ and computed p accordingly each time.

Also note that for $n \to \infty$ we obtain $q \to (p-1)$ and thus a standard geometric distribution. For this reason we will refer to this scheme as the "geometric" distribution in general, even though we have in practice only finite values of n.

3.3 Notes on Randomness

As usual the terms “randomness” and “at random” may be a bit misleading – in fact, for our random choices we use the output of a linear congruential random number generator [9], which is purely deterministic at heart. Thus, for any given random seed (the initial configuration of the generator), the sequence of numbers produced is always the same and we can only try to create the illusion of randomness from an external point of view. That’s why these numbers are often referred to as *pseudorandom numbers*.

It has been shown, for instance by Bach [1] and Mulmuley [11], that pseudorandomness still works very well in practice and that the said theoretical shortcoming does not considerably impair the algorithm’s performance.

Now it is clear that, for a given problem instance, each run of the randomized backtrack search algorithm is solely dependent on the random seed[1]. For a certain seed the search will always explore the search space in the same way – in particular the same solution will be produced and an identical number of backtracks will be required.

With this in mind it is understandable why this concept is sometimes also referred to as “deterministic randomness”. From a commercial point of view, however, this is actually a welcome and sometimes even necessary property, since it enables unproblematic reproduction of results, for instance for debugging purposes.

3.4 Computational Results

For our performance tests we obtained several real-world instances of the tail assignment problem from Carmen Systems, varying in size and complexity:

– `1D17V`: 17 vehicles (only one type), 191 activities over one day.
– `1W17V`: 17 vehicles (only one type), 727 activities over one week.
– `1D30V`: 30 vehicles (three different types), 129 activities over one day.
– `3D74V`: 74 vehicles (nine different types), 780 activities over three days.

In the first two instances, `1D17V` and `1W17V`, all aircraft are of the same type, therefore there are almost no operational constraints as to which vehicle may and may not operate a flight. The two instances `1D30V` and `3D74V`, however, comprise several different types of aircraft and hence exhibit numerous operational constraints.

All practical tests were performed using the Gecode constraint programming environment [15]. The package’s C++ source code is freely available, hence it was rather easy for us to implement our custom propagators and add randomization to the search engine.

We first tried to solve all instances with the nonrandomized backtrack search, using different variable selection heuristics and the smallest-value-first value selection heuristic . The results are shown in Table 1. As expected, given the results

[1] Note that the nonrandomized run can be identified with a specific randomized run resulting from a suitable random seed.

Table 1. Backtracks before finding a solution using nonrandomized search

	1D17V	1W17V	1D30V	3D74V
min-size	8	1	> 1000000	5
min-degree	14	1	> 1000000	> 1000000

from Gabteni and Grönkvist [3], the two instances involving just one vehicle type can be solved rather easily, with only a few backtracks. The more constrained instances, however, exhibit more problematic runtime behavior: We aborted the search after running without result for several hours on a 2 GHz CPU. In the light of our further results, we regard the short run using the *min-size* heuristic on the 3D74V instance as a "lucky shot".

To assess the impact of the randomized extensions specified above, we introduce a random variable X denoting the number of backtracks needed by the randomized backtrack search instance to find a solution. We are then interested in the probability of finding a solution within a certain number of backtracks x, formally $P(X \leq x)$.

In Fig. 1 we plot this probability for all four problem instances, using the geometric value distribution and both *min-size* or *min-degree* with random tie-breaking. For each curve we performed 1.000 independent runs, with different random seeds.

Consistent with the previous findings, the two less constrained instances show a rather satisfactory search cost distribution, with 1W17V approaching 100% at around 10 backtracks only; at this point 1D17V is around 60% successful, it reaches 100% after a few thousand backtracks – although the respective CP model has fewer variables, the instances's connection structure appears to be more complex.

However, the situation is not as good for the distributions arising from the more constrained instances: For 1D30V with the randomized *min-degree* heuristic we get close to 50% after only 10 backtracks, but from then on the probability does not increase notably; with *min-size* the success rate is slightly lower.

The biggest and most constrained instance, 3D74V, can be solved within 20 backtracks in roughly 40% of the runs, using the randomized *min-size* heuristic. The success ratio then slowly increases towards 45% as we allow more backtracks. The randomized *min-degree* performs considerably worse, with slightly more than 20% success after 20 backtracks, subseqeuently increasing towards 25% with more backtracks.

In a next step we applied the inversely exponential scheme described previously, where each unassigned variable x is selected with probability proportional to $b^{-s(x)}$ ($s(x)$ being the current domain size of x). We tested this for $b \in \{e, 2.0, 2.5, 3.0, 3.5, 10.0\}$ with 1D30V and 3D74V and compared it to the best solutions from Fig. 1. The results are given in Figs. 2(a) and 2(b), respectively.

For the 1D30V instance the inversely exponential scheme can partly outperform *min-degree* with randomized tie-breaking. Applied to the 3D74V instance, however, it is clearly inferior to *min-size* with randomized tie-breaking, only as

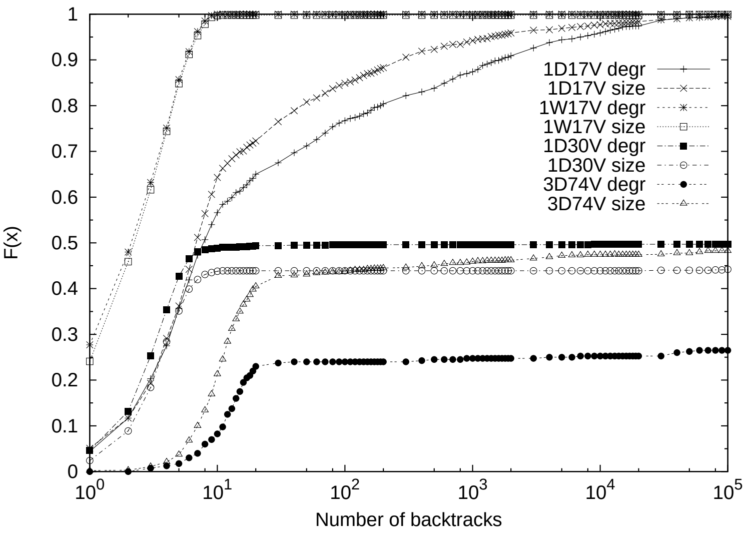


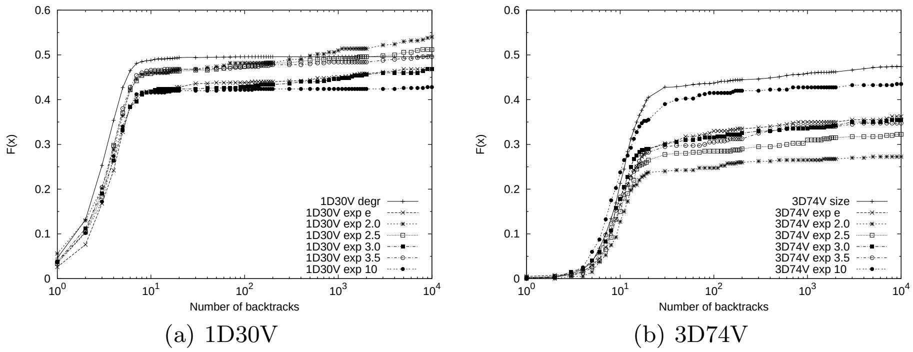
Fig. 1. Plot of the search cost distributions, where $F(x) := P(X \leq x)$

b grows does the success ratio get close (since, as noted earlier, for increasing b we approach the randomized *min-size* again).

All in all we think that the *min-size* heuristic with randomized tie-breaking is the best choice. Although it does not produce the best search cost distributions in some cases, its performance is never far from the respective optimum.

3.5 Analysis of Results

We have noticed before that the two instances with only one vehicle type involved can be solved rather easily by the nonrandomized backtrack search already. This



(a) 1D30V (b) 3D74V

Fig. 2. Results for the inversely exponential variable selection

is also confirmed by the cost distribution for the randomized search and thus not very surprising. The other two instances, however, seem considerably harder.

This behavior can be explained by the presence of *critically constrained variables* in these instances (and the related concept of *backdoors* [13,14]): Once this subset of critical variables has been fixed, the remaining problem is potentially a lot easier and everything else more or less matches up automatically.

Therefore, if the randomized heuristic picks these variables right at the start and "good" values are assigned, a solution will probably be found within a few backtracks. On the other hand, if we start out with noncritical variables or assign "bad" values, the algorithm will explore large portions of the search space to no avail. The former represents the left-hand tail of the distribution, whereas the latter results in the long, rather flat remaining distribution.

4 Restarts

As we saw in Sect. 3, for the more constrained instances the randomized search can quite easily "get stuck" in the right-hand side of the distribution, so that it will take a lot of backtracks to reach a solution. On the other hand we also observed (via the left-hand tail of the distribution) a nonnegligible probability of finding a solution very quickly, with only a few backtracks.

Naturally we want to exploit this fact; a straightforward way to achieve that is the introduction of restarts into the search procedure: We explore the search tree until a certain number of backtracks has been necessary (the *cutoff value*), at which point we assume that we have descended into one of the "bad" subtrees. Thus we abort and restart the search from the initial configuration – this time hoping to make better randomized choices along the way.

4.1 Restart Strategies

The crucial question is then how many backtracks we allow before restarting. A number of such *restart strategies* have previously been proposed.

Gomes et al. [4] propose a fixed cutoff value, meaning we restart the search every $c \in \mathbb{N}$ backtracks; they call this the *rapid randomized restart* strategy. In their work the optimal cutoff value is determined by a trial-and-error process.

If one has more detailled knowledge about the specific search cost distribution of a problem, one can mostly avoid the trial-and-error approach – however, this knowledge is not always available. Moreover the optimal cutoff value potentially needs to be redetermined for every problem, which does not make this approach very general.

Therefore Walsh [12] suggests a strategy of *randomization and geometric restarts*, where the cutoff value is increased geometrically after each restart by a constant factor $r > 1$. This is obviously less sensitive to the underlying distribution and is reported to work well by Walsh [12] and Gomes et al. [4].

An alternative general approach is the *universal strategy* introduced by Luby et al. [10]. They show that the expected number of backtracks for this strategy is only a logarithmic factor away from what you can expect from an optimal strategy. The sequence of cutoff values begins with 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,..., it can be computed via the following recursion:

$$c_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \ , \\ c_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 \ . \end{cases}$$

4.2 Computational Results

We henceforth focus on the two instances `1D30V` and `3D74V`, as their constrainedness and the resulting randomized search cost distribution (cf. Sect. 3.4) predestines them for the introduction of restarts.

We solved each of the two instances with different cutoff values (i. e. the constant cutoff value or the initial value in case of the geometric and universal strategy). For each configuration we ran several hundred iterations with differing random seeds and computed the arithmetic mean of the number of backtracks required to find a feasible solution.

The resulting graphs for the constant cutoff and the universal strategy are given in Fig. 3(a), where the x-axis value is used as a constant multiplier for the universal strategy. For the geometrically increasing cutoff we varied the factor $r \in \{1.1, 1.2, 1.3\}$, the plots of the respective averages are shown in Fig. 3(b).

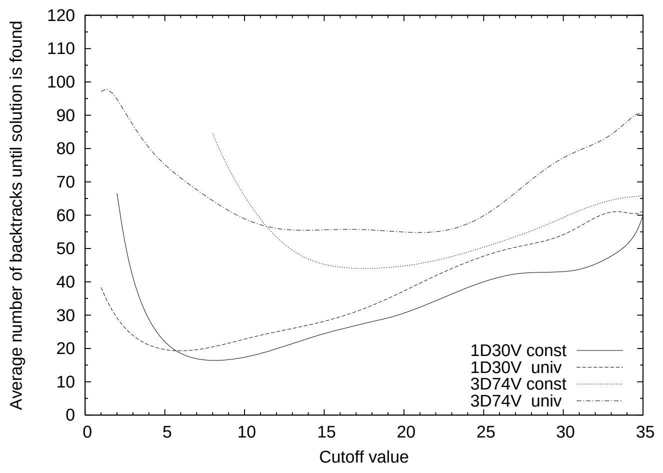
4.3 Analysis of Results

To begin with, we note that the introduction of restarts does exactly what it was intended to do, averting the long tails of the randomized search cost distributions observed in Sect. 3.4. For most configurations a couple of dozen total backtracks suffices to find a solution.

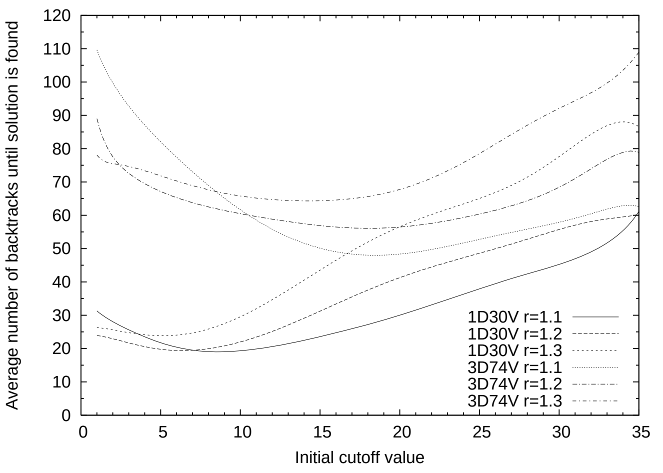
Looking at the left-hand side of the distributions in Fig. 1, it was to be expected that the constant cutoff strategy can deal only poorly (or not at all) with low cutoff values – the chance of a very short run is too small, especially for the `3D74V` instance. Both other restart strategies, with the cutoff increasing over time, can handle these low initial values considerably better, since they will eventually allow a sufficiently high cutoff anyway.

On the other hand, just as noted by Luby et al. in [10], a constant cutoff permits exploiting well-fitting cutoff values more effectively than via the other strategies. This is because, above a certain threshold, increasing the cutoff does not result in a considerably higher probability of finding a solution (cf. the long, almost horizontal tail of the distributions in Fig. 1).

But as we pointed out before, setting an optimal or close-to-optimal cutoff requires knowledge about an instance's search cost distribution, which is mostly not available and may be computationally expensive to obtain. This is the strength of the variable strategies, where the geometric one seems to hold a slight advantage over the universal one, despite the theoretical logarithmic upper bound on the latter's performance (cf. Sect. 4.1) – the universal strategy's intermediate fallbacks to low values do not fit the distributions at hand.



(a) Constant and universal cutoff strategy



(b) Geometric cutoff strategy

Fig. 3. Average number of backtracks after applying restarts to the search procedure

4.4 Search Completeness

One issue with the randomized extensions as described above is that we sacrifice search completeness: Although the random number generator used for the random choices will most probably be in a different state after each restart, one might still end up making the same decisions as before, thereby exploring the same parts of the search space over and over again. Hence, although the probability is evidently low, it is in theory possible to search indefinitely, either missing an existing solution or not establishing the problem's infeasibility.

In principle, with the geometric and universal restart strategy one will eventually have a sufficiently high cutoff value, so that the whole search space will be

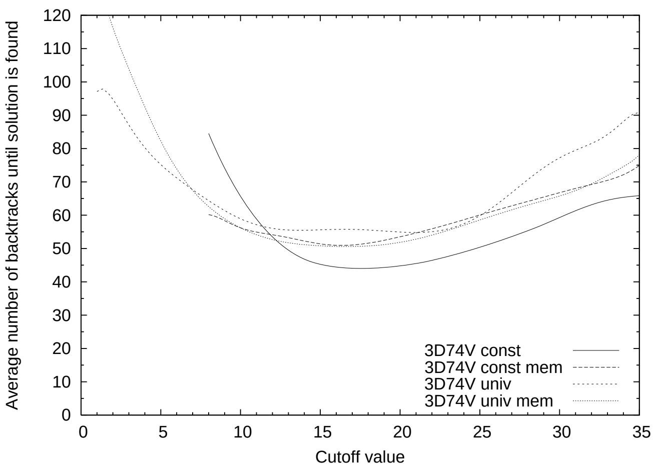


Fig. 4. Impact of memorizing the search history across restarts

explored before restarting and completeness is implicitly ensured. But in practice, given the exponential size of the search space, this will take far too long – also it does not apply to the constant cutoff value strategy.

Therefore we extended the randomized search with a special tree datastructure for the search history, where all visited search tree nodes and "dead ends" (where a backtrack was required) are recorded, thus making sure that the search will not descend into a previously failed subtree.

However, while completeness is a nice theoretic property to attain, we found that in practice it didn't result in enough of a difference to justify the additional processing time and memory consumption, especially since all our instances were known in advance to have at least one solution.

A comparison for the `3D74V` instance, using both the constant cutoff and universal strategy, is plotted in Fig. 4. In some cases the average number of backtracks increases with the introduction of the book-keeping, in some it decreases slightly – probably the differences are partly also due to statistical variations in the relatively low number of runs we performed for each cutoff value.

5 Conclusion and Outlook

Real-world instances of the tail assignment problem impose serious performance problems on standard backtrack search algorithms. Previously this has been solved by the introduction of specialized constraints, that internally make use of the pricing routine of a column generation system.

As an alternative to this we have demonstrated how the use of randomization and restarts can greatly improve the performance of such search algorithms when run on tail assignment instances, reducing the required number of backtracks by several orders of magnitude.

In particular we have shown that with suitable but still generic randomized extensions to the backtrack search procedure we can obtain a substantial probability of finding a solution within just a few backtracks, which we related to the concept of critically constrained variables and backdoors. However, depending on the random choices throughout the search process, we still encounter a lot of very longs runs as well.

Therefore we added restarts to the search engine, which, as we noted, has proven rewarding for other authors before [5,12]. The intention in mind is to exploit the presence of relatively short runs, at the same time avoiding to "get stuck" in the long tail of the search cost distribution. The presented results confirm that this idea works very well for practical purposes.

We have also argued that the randomness is kept "controllable", thereby ensuring reproducibility, which is an important consideration for a potential deployment in a commercial system.

So far we have not been able to experiment with a number of really big problem instances, spanning over a month and comprising well above 2000 activities. This was due to memory limitations on the machines we had at our disposal, in connection with the underlying concept of the Gecode environment, which employs *copying and recomputation* rather than the potentially more memory-efficient *trailing*. However, based on our results we believe that these instances would profit from our approach as well.

To summarize, we feel that randomization and restarts are an effective yet general way to combat computational hardness in constraint satisfaction problems. Consequently, as Gomes et al. note [4], this concept is already being deployed in a number of production systems.

In fact, given that our findings show great potential, the possibility of extending the current Carmen Systems tail assignment optimizer accordingly will be investigated further. In this respect it will certainly be interesting to explore how well randomization and restarts interact with the aforementioned specialized constraints currently used in the system.

Acknowledgments

We would like to thank the developers of Gecode, Christian Schulte, Guido Tack and Mikael Lagerkvist, for their constant willingness to answer our questions and their help with a number of implementation pitfalls.

References

1. E. Bach: Realistic analysis of some randomized algorithms. *Journal of Computer and System Sciences* **42** (1991): 30–53.
2. C. Barnhart, N. L. Boland, L. W. Clarke, E. L. Johnson, G. L. Nemhauser, and R. G. Shenoi: Flight string models for aircraft fleeting and routing. *Transportation Science* **32** (1998): 208–220.
3. S. Gabteni and M. Grönkvist: A hybrid column generation and constraint programming optimizer for the tail assignment problem. In *Proceedings of CPAIOR'06*.

4. C. Gomes, B. Selman, N. Crato, and H. Kautz: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* **24** (2000): 67–100.
5. C. Gomes, B. Selman, and H. Kautz: Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98).*
6. R. Gopalan and K. T. Talluri: Mathematical models in airline schedule planning: A survey. *Annals of Operations Research* **76** (1998): 155–185.
7. M. Grönkvist: A constraint programming model for tail assignment. In *Proceedings of CPAIOR'04*: 142–156.
8. M. Grönkvist: The tail assigment problem. *PhD thesis, Chalmers University of Technology, Gothenburg, Sweden* (2005).
9. D. H. Lehmer: Mathematical methods in large-scale computing units. In *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery (1949)*: 141–146.
10. M. Luby, A. Sinclair, and D. Zuckerman: Optimal speedup of Las Vegas algorithms. *Information Processing Letters, Vol. 47* (1993): 173–180.
11. K. Mulmuley: Randomized Geometric Algorithms and Pseudorandom Generators. *Algorithmica* **16** (1996): 450–463.
12. T. Walsh: Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*: 1172–1177.
13. R. Williams, C. Gomes, and B. Selman: Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03).*
14. R. Williams, C. Gomes, and B. Selman: On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03).*
15. Gecode: Generic constraint development environment. `http://www.gecode.org/`

Towards an Efficient SAT Encoding for Temporal Reasoning

Duc Nghia Pham, John Thornton, and Abdul Sattar

[1] Safeguarding Australia Program, National ICT Australia Ltd., Australia
[2] Institute for Integrated and Intelligent Systems, Griffith University, Australia
{duc-nghia.pham, john.thornton, abdul.sattar}@nicta.com.au

Abstract. In this paper, we investigate how an IA network can be effectively encoded into the SAT domain. We propose two basic approaches to modelling an IA network as a CSP: one represents the relations between intervals as variables and the other represents the relations between end-points of intervals as variables. By combining these two approaches with three different SAT encoding schemes, we produced six encoding schemes for converting IA to SAT. These encodings were empirically studied using randomly generated IA problems of sizes ranging from 20 to 100 nodes. A general conclusion we draw from these experimental results is that encoding IA into SAT produces better results than existing approaches. Further, we observe that the phase transition region maps directly from the IA encoding to each SAT encoding, but, surprisingly, the location of the hard region varies according to the encoding scheme. Our results also show a fixed performance ranking order over the various encoding schemes.

1 Introduction

Representing and reasoning about time dependent information (i.e., *temporal reasoning*), is a central research issue in computer science and artificial intelligence. The basic tasks include the design and development of efficient reasoning methods for finding scenarios that are consistent with the given information, and effectively answering queries. Often, such information is incomplete and uncertain. One of the most expressive formalisms used to represent such qualitative temporal information is the Interval Algebra (IA) proposed by Allen [1].

While IA is an expressively rich framework, the reasoning problem is computationally intractable [21]. Existing reasoning techniques are generally based on the backtracking approach (proposed by Ladkin and Reinefeld [10]), which uses path consistency as forward checking. Although this approach has been further improved [13,20], it and its variants still rely on path consistency checking at each step to prune the search space. This *native* IA approach has the advantage of being fairly compact, but is disadvantaged by the overhead of continually ensuring path-consistency. Additionally, the native IA representation of variables and constraints means that state-of-the-art local search and systematic search heuristics cannot be easily transferred to the temporal domain.

In practice, existing native IA backtracking approaches are only able to find consistent solutions for relatively small general IA instances [18,17]. On the other hand, recent research has shown that modelling and solving hard combinatorial problems (including planning problems) as SAT instances can produce significant performance benefits over solving problems in their original form [9,8,15]. This motivated us to undertake this study.

In this paper we investigate whether the representation of IA problems using specialised models that require specialised algorithms is necessary in the general case. Given that the development of such approaches takes considerable effort, we would expect significant performance benefits to result. To answer this question, we look at expressing IA as a CNF formula using six different SAT encoding schemes. This enables us to apply a range of SAT solvers and to compare the performance of these with the existing native IA approaches. To the best of our knowledge, there is no explicit and thorough work on formulating temporal problems as SAT instances. Nebel and Bürckert [14] pointed out that qualitative temporal instances can be translated to SAT instances but that such a translation causes an exponential blowup in problem size. Hence, no further investigation was provided in their work.[1]

The remainder of the paper is structured as follows: next we review the basic definitions of IA. Then in Section 3 we introduce two models for transforming IA instances into CSP instances. Using these methods, combined with three CSP-to-SAT encodings, six IA-to-SAT encodings are presented in Section 4. Sections 5-7 present an empirical study to investigate the hardness distribution of these SAT encodings and evaluate their performance relative to each other, and in comparison to existing approaches. Finally, Section 8 presents the conclusion and discusses future research directions.

2 Interval Algebra

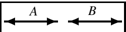
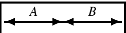
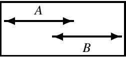
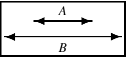
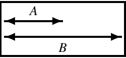
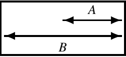
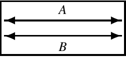
Interval Algebra [1] is the most commonly used formalism to represent temporal interval events. It consists of a set of 13 atomic relations between two time intervals: $\mathcal{I} = \{eq, b, bi, m, mi, o, oi, d, di, s, si, f, fi\}$ (see Table 1). Indefinite information between two time intervals can be expressed as a subset of $\mathcal{I}$ (e.g. a disjunction of atomic relations). For example, the statement *"Event A can happen either before or after event B"* can be expressed as $A\{b, bi\}B$. Hence there are a total of $2^{|\mathcal{I}|}$ = 8,192 possible relations between pairs of temporal intervals.

Let R_1 and R_2 be two IA relations. Then the four operators of IA: *union* ($\cup$), *intersection* ($\cap$), *inversion* ($^{-1}$), and *composition* ($\circ$), can be defined as follows:

$$\begin{gathered}
\forall\, A, B : A(R_1 \cup R_2)B \leftrightarrow (AR_1B \vee AR_2B) \\
\forall\, A, B : A(R_1 \cap R_2)B \leftrightarrow (AR_1B \wedge AR_2B) \\
\forall\, A, B : A(R_1^{-1})B \leftrightarrow BR_1A \\
\forall\, A, B : A(R_1 \circ R_2)B \leftrightarrow \exists\, C : (AR_1C \wedge CR_2B).
\end{gathered}$$

[1] Recent independent work [6] has proposed representing IA as SAT, but the authors do not specify the transformation in detail, and do not provide an adequate empirical evaluation.

Table 1. The 13 IA atomic relations. Note that the endpoint relations $A^- < A^+$ and $B^- < B^+$ have been omitted.

Atomic relation	Symbol	Meaning	Endpoint relations
A before *B*	b		$A^- < B^-, A^- < B^+$
B after *A*	bi		$A^+ < B^-, A^+ < B^+$
A meets *B*	m		$A^- < B^-, A^- < B^+$
B met by *A*	mi		$A^+ = B^-, A^+ < B^+$
A overlaps *B*	o		$A^- < B^-, A^- < B^+$
B overlapped by *A*	oi		$A^+ > B^-, A^+ < B^+$
A during *B*	d		$A^- > B^-, A^- < B^+$
B includes *A*	di		$A^+ > B^-, A^+ < B^+$
A starts *B*	s		$A^- = B^-, A^- < B^+$
B started by *A*	si		$A^+ > B^-, A^+ < B^+$
A finishes *B*	f		$A^- > B^-, A^- < B^+$
B finished by *A*	fi		$A^+ > B^-, A^+ = B^+$
A equals *B*	eq		$A^- = B^-, A^- < B^+$
			$A^+ > B^-, A^+ = B^+$

Hence, the *intersection* and *union* of any two temporal relations (R_1, R_2) are simply the standard set-theoretic intersection and union of the two sets of atomic relations describing R_1 and R_2, respectively. The *inversion* of a temporal relation R is the union of the inversion of each atomic relation $r_i \in R$. The *composition* of any pair of temporal relations (R_1, R_2) is the union of all results of the composition operation on each pair of atomic relations (r_{1i}, r_{2j}), where $r_{1i} \in R_1$ and $r_{2j} \in R_2$. The full composition results of these IA atomic relations can be found in [1].

An IA network can be represented as a *constraint graph* or a *constraint network* where the vertices represent interval events and the arcs are labelled with the possible interval relations between a pair of intervals [13]. Usually, such a constraint graph for n interval events is described by an $n \times n$ matrix M, where each entry M_{ij} is the label of the arc between the i^{th} and j^{th} intervals. An IA *scenario* is a *singleton* IA network where each arc (constraint) is labelled with *exactly one* atomic relation.

An IA network with n intervals is *globally consistent* iff it is strongly n-consistent [11]. Hence, the ISAT problem of determining the satisfiability of a given IA network becomes the problem of determining whether that network is globally consistent [1,13]. ISAT is the *fundamental* reasoning task in the temporal reasoning community because all other interesting reasoning problems can be reduced to it in polynomial time [7] and it is one of the most important tasks in practical applications [20].

It is worth noting that enforcing *path consistency* [11,3] is enough to ensure global consistency for the maximal tractable subclasses of IA, including singleton networks [13]. Allen [1] proposed a path consistency method for an IA network M that repeatedly computes the following *triangle operation*: $M_{ij} \leftarrow M_{ij} \cap M_{ik} \circ M_{kj}$ for all triplets of vertices (i, j, k) until no further change occurs or until $M_{ij} = \emptyset$. These operations remove all the atomic relations that cause an inconsistency between any triple (i, j, k) of intervals. The resulting network is a path consistent IA network. If $M_{ij} = \emptyset$, then the original IA network is path inconsistent. More sophisticated path consistency algorithms have been applied to IA networks that run in $O(n^3)$ time [19,13].

3 Reformulation of IA into CSP

A common approach to encode combinatorial problems into SAT is to divide the task into two steps: (i) modelling the original problem as a CSP; and (ii) mapping the new CSP into SAT. In the next two subsections, we propose two transformation methods to model IA networks as CSPs such that these CSPs can be feasibly translated into SAT. We then discuss three SAT encoding schemes to map the CSP formulations into SAT, producing six different approaches to encode IA networks into SAT. [2]

3.1 The Interval-Based CSP Formulation

A straightforward method to formulate IA networks as CSPs is to represent each arc as a CSP variable. We then limit the domain of each variable to the set of permissible IA atomic relations for that arc, rather than the set of all subsets of $\mathcal{I}$ used in existing IA approaches. This allows us to reduce the domain size of each variable from 2^{13} to a maximum of 13 values. Thus an instantiation of an *interval*-based CSP maps each variable (arc) to *exactly one* atomic relation in its domain. In other words, an instantiation of this new CSP model is actually a singleton network of the original IA network.

Lemma 1. *Let Θ be a singleton IA network with 3 intervals I_1, I_2, and I_3. Then Θ is consistent iff $r_{13} \in r_{12} \circ r_{23}$ where r_{ij} is an arc between any two I_i and I_j intervals.*

Proof. Trivial as there is exactly one mapping of a singleton network onto the time line.

Theorem 1. *Let Θ be a singleton IA network with n intervals and r_{ij} be the label of the arc between I_i and I_j. Then Θ is consistent iff for any triple $(i < k < j)$ of vertices, $r_{ij} \in r_{ik} \circ r_{kj}$.*

Proof. $(\Rightarrow)$ This direction is trivial as Θ is also path consistent.

$(\Leftarrow)$ As $r_{ij} \in r_{ik} \circ r_{kj}$ holds for all triplets $(i < k < j)$ of vertices, Θ is path consistent by Lemma 1. In addition, Θ is singleton. Hence, Θ is globally consistent.

Based on the results of Theorem 1, an *interval*-based CSP representation of a given IA network is defined as follows:

Definition 1. *Given an IA network M with n intervals, $I_1, \ldots, I_n$; the corresponding interval-based CSP is $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where*
$\mathcal{X} = \{X_{ij} \mid i, j \in [1..n], i < j\}$; each variable X_{ij} represents a relation between two intervals I_i and I_j;
$\mathcal{D} = \{D_{ij}\}$, each D_{ij} is a set of domain values for X_{ij}, and $D_{ij} = M_{ij}$ the set of relations between interval I_i and I_j; and
$\mathcal{C}$ consists of the following constraints:

$$\bigwedge_{x \in D_{ik}, y \in D_{kj}} X_{ik} = x \wedge X_{kj} = y \Longrightarrow X_{ij} \in D'_{ij} \tag{1}$$

where $i < k < j$ and $D'_{ij} = D_{ij} \cap (x \circ y)$.

[2] In practice, IA networks can be directly encoded into SAT formulae without being reformulated as CSPs. However, for the sake of clarity we first transform IA into two different CSP formulations and then to SAT.

Theorem 2. *Let Θ be an IA network and Φ be the corresponding interval-based CSP defined by Definition 1. Then Θ is globally consistent or satisfiable iff Φ is satisfiable.*

Proof. We first rewrite the constraint (1) into two equivalent clauses

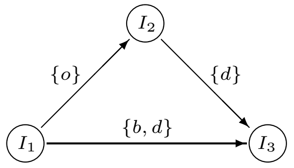
$$\bigwedge_{x \in D_{ik}, y \in D_{kj}} X_{ik} = x \wedge X_{kj} = y \Longrightarrow X_{ij} \in D_{ij} \tag{2}$$

$$\bigwedge_{x \in D_{ik}, y \in D_{kj}} X_{ik} = x \wedge X_{kj} = y \Longrightarrow X_{ij} \in x \circ y \tag{3}$$

($\Rightarrow$) Let Θ' be a consistent scenario of Θ. As Θ' is a singleton network, Θ' is also an instantiation of Φ by Definition 1. Hence clause (2) is satisfied. In addition, as Θ' is globally consistent, clause (3) is also satisfied by Theorem 1. Hence Θ' satisfies all constraints of Φ. As a result, Φ is satisfiable.

($\Leftarrow$) Let Φ' be an instantiation of Φ such that it satisfies all constraints of Φ (i.e. clauses (2) and (3) are satisfied). We construct a singleton network Θ' by labelling each arc (i, j) of Θ' with the atomic relation (value) $\Phi'(i, j)$. As Φ' satisfies clause (2), Θ' is a singleton network of Θ. In addition, as Φ' satisfies clause (3), we have $\Theta'(i, j) \in \Theta'(i, k) \circ \Theta'(k, j)$ for all triples $(i < k < j)$ of vertices. Applying Theorem 1, Θ' is globally consistent. As a result, Θ is satisfiable.

Example. For the sake of clarity, we use the IA network in Figure 1(a) as a running example to illustrate the transformation of IA networks into CSPs and SAT encodings. The example represents the following scenario: *"Anne usually reads her paper (I_1) before or during her breakfast (I_3). In addition, she always drinks a cup of coffee (I_2) during her breakfast. This morning, she started reading her paper before her coffee was served and finished reading before drinking the last of her coffee"*. The corresponding interval-based CSP of this IA network is shown in Figure 1(b), having 3 variables, which represent the temporal relations between each pair of actions. These variables and their corresponding domains are described using the same order in $\mathcal{X}$ and $\mathcal{D}$. Note that as $\{o\} \circ \{d\} = \{o, d, s\}$, the constraint between I_1, I_2 and I_3 further restricts the domain of X_{13} to $\{d\}$ instead of its original $\{b, d\}$, i.e. Anne could not have read her paper before breakfast if she was still reading it while drinking coffee *during* breakfast.



(a) IA network

$$\mathcal{X} = \{\ X_{12},\ X_{13},\ X_{23}\ \}$$
$$\mathcal{D} = \{\ \{o\},\ \{b, d\},\ \{d\}\ \}$$
$$\mathcal{C} = \{\ (X_{12} = o \wedge X_{23} = d \Longrightarrow X_{13} = d)\ \}$$

(b) interval-based CSP

Fig. 1. An interval-based CSP representation of the running example

3.2 The Point-Based CSP Formulation

Vilain and Kautz [21] proposed the Point Algebra (PA) to model qualitative information between time points. PA consists of a set of 3 atomic relations $\mathcal{P} = \{<, =, >\}$ and 4 operators defined in a similar manner to IA. In addition, the concepts of consistency discussed above for IA networks are also applicable to PA networks. Again, we use an $n \times n$ matrix P to represent a PA network with n points where P_{ij} is the relation between two points i and j.

As mentioned in Section 2, IA atomic relations can be uniquely expressed in terms of their endpoint relations. However, representing non-atomic IA relations is more complex, as not all IA relations can be translated into point relations. For example, the following combination of point relations

$$(A^- \neq B^-) \wedge (A^- < B^+) \wedge (A^+ \neq B^-) \wedge (A^+ < B^+)$$

represents not only $A\{b, d\}B$ but also $A\{b, d, o\}B$. This means that PA can only cover 2% of IA [13].

Using the CSP formalism, we can prevent the instantiation of such undesired IA relations by simply introducing new constraints into the CSP model. Let $\mu(r) = (v_{ss}, v_{se}, v_{es}, v_{ee})$ be the PA representation of an IA atomic relation r between two intervals A and B, where v_{se}, for example, is the corresponding PA relation between two endpoints A^- and B^+. We then define the *point*-based CSP model of an IA network as follows:

Definition 2. *Given an IA network M with n intervals and its corresponding PA network P (with $2n$ points, P_1,...,P_{2n}), the point-based CSP of M is $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{X_{ij} \mid i, j \in [1..2n], i < j\}$; each variable X_{ij} represents a relation between two points P_i and P_j of P;*
$\mathcal{D} = \{D_{ij}\}$, each D_{ij} is the set of domain values for X_{ij} and $D_{ij} = P_{ij}$ the set of point relations between P_i and P_j; and
$\mathcal{C}$ consists of the following constraints:

$$\bigwedge_{x \in D_{ik}, y \in D_{kj}} X_{ik} = x \wedge X_{kj} = y \Longrightarrow X_{ij} \in D'_{ij} \tag{4}$$

$$\bigwedge_{r \notin M_{lm}} (X_{l^-m^-}, X_{l^-m^+}, X_{l^+m^-}, X_{l^+m^+}) \neq \mu(r) \tag{5}$$

where $i < k < j$, $D'_{ij} = D_{ij} \cap (x \circ y)$, and $X_{l^*m^*}$ is the CSP variable representing the relation between one endpoint of interval l and one endpoint of interval m.

Theorem 3. *Let Ω be a singleton PA network with n points and r_{ij} be the label of the arc between two points I_i and I_j. Then Ω is consistent iff for any triple $(i < k < j)$ of vertices, $r_{ij} \in r_{ik} \circ r_{kj}$.*

As Theorem 3 is similar to Theorem 1, we can construct its proof in a similar way to the proof of Theorem 1.

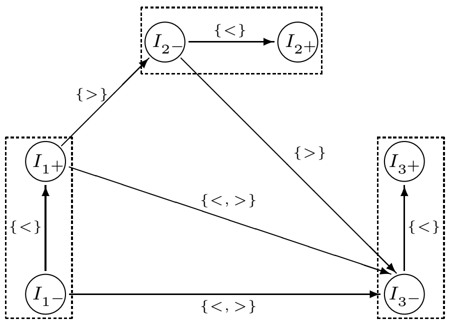
Theorem 4. *Let Θ be an IA network and Ψ be the corresponding point-based CSP defined by Definition 2. Then Θ is globally consistent or satisfiable iff Ψ is satisfiable.*

Proof. ($\Rightarrow$) Let Θ' be a consistent scenario of Θ. As Θ' is a singleton network, its corresponding point-based CSP Ψ', defined by Definition 2, is an instantiation of Ψ. Hence, Ψ' satisfies all constraints (5). In addition, as Θ' is globally consistent, Ψ' satisfies all constraints (4) due to Theorem 3. As a result, Ψ is satisfiable.

($\Leftarrow$) Let Ψ' be an instantiation of Ψ such that all constraints (4) and (5) are satisfied. Let $\mu^{-1}(X_{l^-m^-}, X_{l^-m^+}, X_{l^+m^-}, X_{l^+m^+}) = r$ be the inversion of $\mu(r)$, such that it maps the combination of the PA atomic relations of four endpoints $(X_{l^-m^-}, X_{l^-m^+}, X_{l^+m^-}, X_{l^+m^+})$ to the IA atomic relation r between two intervals l and m. As every variable $X_{l^*m^*}$ of Ψ' is instantiated with exactly one atomic relation, $\mu^{-1}(X_{l^-m^-}, X_{l^-m^+}, X_{l^+m^-}, X_{l^+m^+})$ maps to exactly one interval relation.

We construct a singleton IA network Θ' from Ψ' by labelling each arc (l, m) with the corresponding IA atomic relation $\mu^{-1}(X_{l^-m^-}, X_{l^-m^+}, X_{l^+m^-}, X_{l^+m^+})$. As Ψ' satisfies all constraints (5), Θ' is a scenario of Θ. In addition, Θ' is globally consistent by the application of Theorem 3 as Ψ' satisfies all constraints (4). As a result, Θ is satisfiable.

Example. Figure 2 shows a point-based CSP corresponding to the original IA network from Figure 1(a), including a partial PA graph to assist in understanding the point-based CSP translation. In this graph (Figure 2(a)), each interval I_i has been replaced by its endpoints I_{i^-} (the start point) and I_{i^+} (the finish point) and all temporal relations between pairs of intervals have been replaced by corresponding relations between their endpoints. These endpoint relations are the CSP variables in the new model, which are in turn instantiated with PA atomic relations. For example, the expression $X_{1^-1^+} = <$ means that the arc between the endpoints I_{1^-} and I_{1^+} must be instantiated with the value $<$, thereby expressing the underlying PA constraint $I_{1^-} < I_{1^+}$. The power we obtain from this CSP model is that we can disallow unwanted interpretations that cannot be eliminated from a simple PA network. For example, in Figure 2(a) the PA graph is not a correct alternative representation of the original IA network as it allows interval I_1 to overlap (o) with interval I_3. In the CSP formalism we can disallow this overlapping relation using the third constraint in Figure 2(b): $(X_{1^-3^-} \neq <) \wedge (X_{1^-3^+} \neq <) \wedge (X_{1^+3^-} \neq >) \wedge (X_{1^+3^+} \neq <)$. It should further be noted that the order of domains in $\mathcal{D}$ is preserved exactly with respect to their corresponding variables in $\mathcal{X}$ and that all



(a) PA network

$$
\begin{aligned}
\mathcal{X} = \{ \; & X_{1^-1^+}, X_{2^-2^+}, X_{3^-3^+}, \\
& X_{1^-2^-}, X_{1^-2^+}, X_{1^+2^-}, X_{1^+2^+}, \\
& X_{1^-3^-}, X_{1^-3^+}, X_{1^+3^-}, X_{1^+3^+}, \\
& X_{2^-3^-}, X_{2^-3^+}, X_{2^+3^-}, X_{2^+3^+} \; \} \\
\mathcal{D} = \{ \; & \{<\}, \{<\}, \{<\}, \\
& \{<\}, \{<\}, \{>\}, \{<\}, \\
& \{<,>\}, \{<\}, \{<,>\}, \{<\}, \\
& \{>\}, \{<\}, \{>\}, \{<\} \; \} \\
\mathcal{C} = \{ \; & (X_{1^-1^+} = < \wedge X_{1^+3^-} = < \Rightarrow X_{1^-3^-} = <), \\
& (X_{1^+2^-} = > \wedge X_{2^-3^-} = > \Rightarrow X_{1^+3^-} = >), \\
& (X_{1^-3^-} \neq < \wedge X_{1^-3^+} \neq < \wedge \\
& X_{1^+3^-} \neq > \wedge X_{1^+3^+} \neq <) \; \}
\end{aligned}
$$

(b) point-based CSP

Fig. 2. A point-based CSP representation of the running example

constraints of type (4) that do not further restrict the domain values of a variable have been omitted.

4 Reformulation of IA into SAT

In this section, we describe three different schemes to encode the interval-based or point-based non-binary CSP formulations (as described in the previous section) into SAT, resulting in six different ways of encoding IA into SAT. First, we describe the one-dimensional (1D) support scheme that naturally translates IA CSPs into CNF formulae. We then present extensions of the *direct* and *log* encoding schemes [8,22].

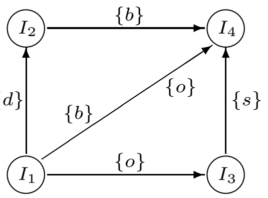
4.1 The SAT 1-D Support Encoding

Using either interval-based or point-based CSP formulations, an IA network can be encoded as a SAT instance, in which each Boolean variable x_{ij}^r represents an assignment of a domain value r to a CSP variable X_{ij}. The Boolean variable x_{ij}^r is true iff the value r is assigned to the CSP variable X_{ij}. For each CSP variable X_{ij} having a domain of values D_{ij}, two sets of at-least-one (ALO) and at-most-one (AMO) clauses are used to ensure that there is exactly one domain value $v \in D_{ij}$ assigned to X_{ij} at any time:

$$ALO: \bigvee_{v \in D_{ij}} x_{ij}^v \tag{6}$$

$$AMO: \bigwedge_{u,v \in D_{ij}} \neg x_{ij}^u \vee \neg x_{ij}^v \tag{7}$$

It is common practice to encode a general CSP into a SAT formula without the AMO clauses, thereby allowing CSP variables to be instantiated with more than one value [22]. A CSP solution can then be extracted by taking any single SAT-assigned value for each CSP variable. However, our two CSP formulation methods strongly depend on the fact that each CSP variable can only be instantiated with exactly one value at any time. This maintains the completeness of our reformulation methods (see the proofs above). A counter-example is shown in Figure 3. I_1 is *before* I_4 because I_1 is *during* I_2 and I_2 is *before* I_4. In addition, as I_1 *overlaps* I_3 and I_3 *starts* I_4, I_1 *overlaps* I_4. As a result, I_1 is either *before* or *overlaps* I_4. However, neither of the scenarios obtained from this network is consistent. Hence, the AMO clauses cannot be removed from our translation.



$$X_{12} = d \wedge X_{24} = b \Rightarrow X_{14} = b$$
$$X_{13} = o \wedge X_{34} = s \Rightarrow X_{14} = o$$

Fig. 3. A counter-example of removing AMO clauses

A natural way to encode the consistency constraints, i.e. constraints (1) and (4) above, is to add the following support (SUP) clauses:

$$SUP: \quad \bigwedge_{u \in D_{ik}, v \in D_{kj}} \neg x_{ik}^{u} \vee \neg x_{kj}^{v} \vee x_{ij}^{w_1} \vee \ldots \vee x_{ij}^{w_m} \tag{8}$$

where $D'_{ij} = D_{ij} \cap (u \circ v) = \{w_1, \ldots, w_m\}$. Note that we use the IA composition table for the interval-based reduction method and the PA composition table for the point-based reduction method.

The constraints (5) in a point-based CSP are translated into a SAT formula using the following *forbidden* (FOR) clauses:

$$FOR: \quad \bigwedge_{r \notin M_{lm}} \neg x_{l^- m^-}^{u} \vee \neg x_{l^- m^+}^{v} \vee \neg x_{l^+ m^-}^{y} \vee \neg x_{l^+ m^+}^{z} \tag{9}$$

where u, v, y, z are PA atomic relations and $\mu(r) = (u, v, y, z)$. For example, given that the PA representation of $X_l\{o\}X_m$ is $\mu(o) = (<, <, >, <)$, the corresponding forbidden clause is $\neg x_{l^- m^-}^{<} \vee \neg x_{l^- m^+}^{<} \vee \neg x_{l^+ m^-}^{>} \vee \neg x_{l^+ m^+}^{<}$.

We refer to this method as the 1-D *support* encoding scheme because it encodes the support values of the original problem. In Gent's support encoding scheme [5], the support clauses are necessary for both implication directions of the CSP constraints. However, in our scheme, only one SUP clause is needed for each triple of intervals $(i < k < j)$, and not for *all* permutation orders of this triple.

$$\begin{array}{ll}
\text{ALO:} & (x_{12}^{o})\ (x_{13}^{b} \vee x_{13}^{d})\ (x_{23}^{d}) \\
\text{AMO:} & (\neg x_{13}^{b} \vee \neg x_{13}^{d}) \\
\text{SUP:} & (\neg x_{12}^{o} \vee \neg x_{23}^{d} \vee x_{13}^{d})
\end{array}$$

Fig. 4. An interval-based 1-D support encoding of the running example

$$\begin{array}{lcccc}
\text{ALO:} & (x_{1^-1^+}^{<}) & (x_{2^-2^+}^{<}) & (x_{3^+3^+}^{<}) & \\
 & (x_{1^-2^-}^{<}) & (x_{1^-2^+}^{<}) & (x_{1^+2^-}^{>}) & (x_{1^+2^+}^{<}) \\
 & (x_{1^-3^-}^{<} \vee x_{1^-3^-}^{>}) & (x_{1^-3^+}^{<}) & (x_{1^+3^-}^{<} \vee x_{1^+3^-}^{>}) & (x_{1^+3^+}^{<}) \\
 & (x_{2^-3^-}^{>}) & (x_{2^-3^+}^{<}) & (x_{2^+3^-}^{>}) & (x_{2^+3^+}^{<}) \\
\text{AMO:} & \multicolumn{2}{c}{(\neg x_{1^-3^-}^{<} \vee \neg x_{1^-3^-}^{>})} & \multicolumn{2}{c}{(\neg x_{1^+3^-}^{<} \vee \neg x_{1^+3^-}^{>})} \\
\text{SUP:} & \multicolumn{2}{c}{(\neg x_{1^-1^+}^{<} \vee \neg x_{1^+3^-}^{<} \vee x_{1^-3^-}^{<})} & \multicolumn{2}{c}{(\neg x_{1^+2^-}^{>} \vee \neg x_{2^-3^-}^{>} \vee x_{1^+3^-}^{>})} \\
\text{FOR:} & \multicolumn{4}{c}{(\neg x_{1^-3^-}^{<} \vee \neg x_{1^-3^+}^{<} \vee \neg x_{1^+3^-}^{>} \vee \neg x_{1^+3^+}^{<})}
\end{array}$$

Fig. 5. A point-based 1-D support encoding of the running example

4.2 The SAT Direct Encoding

Another way of representing CSP constraints as SAT clauses is to encode the conflict values between any pair of CSP variables [8,22]. This *direct* encoding scheme for IA networks can be derived from our 1-D support encoding scheme by replacing the SUP clauses with conflict (CON) clauses. If we represent SUP clauses between a triple of intervals $(i < k < j)$ as a 3D array of allowable values for the CSP variable X_{ij} given the values of X_{ik} and X_{kj}, then the corresponding CON clauses are defined as:

$$CON : \bigwedge_{u \in D_{ik}, v \in D_{kj}, w \in D''_{ij}} \neg x^u_{ik} \vee \neg x^v_{kj} \vee \neg x^w_{ij} \tag{10}$$

where $D''_{ij} = D_{ij} - (u \circ v)$.

The *multivalued* encoding [15] is a variation of the direct encoding, where all AMO clauses are omitted. As discussed earlier, we did not consider such an encoding because in our IA transformations the AMO clauses play a necessary role.

$$\begin{array}{ll}
\text{ALO:} & (x^o_{12})\ (x^b_{13} \vee x^d_{13})\ (x^d_{23}) \\
\text{AMO:} & (\neg x^b_{13} \vee \neg x^d_{13}) \\
\text{CON:} & (\neg x^o_{12} \vee \neg x^d_{23} \vee \neg x^b_{13})
\end{array}$$

Fig. 6. An interval-based direct encoding of the running example

$$\begin{array}{lcccc}
\text{ALO:} & (x^<_{1-1+}) & (x^<_{2-2+}) & (x^<_{3+3+}) & \\
 & (x^<_{1-2-}) & (x^<_{1-2+}) & (x^>_{1+2-}) & (x^<_{1+2+}) \\
 & (x^<_{1-3-} \vee x^>_{1-3-}) & (x^<_{1-3+}) & (x^<_{1+3-} \vee x^>_{1+3-}) & (x^<_{1+3+}) \\
 & (x^>_{2-3-}) & (x^<_{2-3+}) & (x^>_{2+3-}) & (x^<_{2+3+}) \\
\text{AMO:} & \multicolumn{2}{c}{(\neg x^<_{1-3-} \vee \neg x^>_{1-3-})} & \multicolumn{2}{c}{(\neg x^<_{1+3-} \vee \neg x^>_{1+3-})} \\
\text{CON:} & \multicolumn{4}{l}{(\neg x^<_{1-1+} \vee \neg x^<_{1+3-} \vee \neg x^>_{1-3-})\ (\neg x^>_{1+2-} \vee \neg x^>_{2-3-} \vee \neg x^<_{1+3-})} \\
\text{FOR:} & \multicolumn{4}{c}{(\neg x^<_{1-3-} \vee \neg x^<_{1-3+} \vee \neg x^>_{1+3-} \vee \neg x^<_{1+3+})}
\end{array}$$

Fig. 7. A point-based direct encoding of the running example

4.3 The SAT Log Encoding

A compact version of the direct encoding is the *log* encoding [8,22]. Here, a Boolean variable x^l_i is true iff the corresponding CSP variable X_i is assigned a value in which the l-th bit of that value is 1. We can linearly derive log encoded IA instances from direct encoded IA instances by replacing each Boolean variable in the direct encoding with its *bitwise representation*. As a single instantiation of the underlying CSP variable is enforced by the bitwise representation, ALO and AMO clauses are omitted. However, extra prohibited (PRO) clauses are added (if necessary) to prevent undesired bitwise representations from being instantiated. For example, if the domain of variable X has three values then we have to add the clause $\neg x^0_3 \vee \neg x^1_3$ to prevent the fourth value from assigning to X. Another way to handle redundant bitwise representations is to treat them as equivalent to a valid representation. However, this *binary* encoding [4] tends to generate exponentially more conflict clauses than the log encoding and hence is not considered in this study.

$$\begin{array}{ll}
\text{PRO:} & (\neg x^1_{12})\quad (\neg x^1_{23}) \\
\text{CON}_l\text{:} & (x^1_{12} \vee x^1_{23} \vee x^1_{13})
\end{array}$$

Fig. 8. An interval-based log encoding of the running example

$$\begin{array}{lllll}
\text{PRO:} & (\neg x^1_{1-1+}) & (\neg x^1_{2-2+}) & (\neg x^1_{3+3+}) & \\
 & (\neg x^1_{1-2-}) & (\neg x^1_{1-2+}) & (\neg x^1_{1+2-}) & (\neg x^1_{1+2+}) \\
 & & (\neg x^1_{1-3+}) & & (\neg x^1_{1+3+}) \\
 & (\neg x^1_{2-3-}) & (\neg x^1_{2-3+}) & (\neg x^1_{2+3-}) & (\neg x^1_{2+3+}) \\
\text{CON}_l\text{:} & \multicolumn{4}{l}{(x^1_{1-1+} \vee x^1_{1+3-} \vee \neg x^1_{1-3-})\ (x^1_{1+2-} \vee x^1_{2-3-} \vee x^1_{1+3-})} \\
\text{FOR:} & \multicolumn{4}{l}{\qquad (x^1_{1-3-} \vee x^1_{1-3+} \vee \neg x^1_{1+3-} \vee x^1_{1+3+})}
\end{array}$$

Fig. 9. A point-based log encoding of the running example

5 The Phase Transition of SAT-Encoded IA Instances

As our SAT translations were theoretically proved sound and complete, we expected that the following properties would also be true for our SAT-encoded IA instances:

i) The phase transition of SAT-encoded instances happens at the same critical value of the average degree parameter d as for the original IA instances; and
ii) The performance of SAT solvers on SAT-encoded instances is proportionally similar to the performance of temporal backtracking algorithms on the original IA instances.

To verify these properties, we conducted a similar experiment to that reported in Nebel's study [13]. We generated an extensive benchmark test set of $A(n, d, 6.5)$ IA instances by varying the average degree d from 1 to 20 (in steps of 0.5 from 8 to 11 and in steps of 1 otherwise) and n from 20 to 50 (in steps of 5).[3] We generated 500 instances for each n/d data point to obtain a set of $23 \times 7 \times 500 = 80,500$ test instances. We then ran two variants of Nebel's backtracking algorithm [13], NBT$_\mathcal{I}$ and NBT$_\mathcal{H}$, on these instances and zChaff [12] on the corresponding SAT-encoded instances. NBT$_\mathcal{I}$ instantiates each arc with an atomic relation in $\mathcal{I}$, whereas NBT$_\mathcal{H}$ assigns a relation in the set $\mathcal{H}$ of ORD-Horn relations to each arc. The other heuristics used in Nebel's backtracking algorithm were set to default and all solvers were timed out after one hour.

As expected, the probability of satisfiability for our SAT-encoded instances was the same as the probability of satisfiability for the original IA instances, regardless of the SAT translation method. This is illustrated in Figure 10 which shows that the phase transition happens around $d = 9.5$ for $s = 6.5$ regardless of instance size or representation. These results are consistent with the earlier work of Nebel [13].

However, the performance of zChaff on our six different SAT encodings was relatively significantly different from the performance of NBT$_\mathcal{I}$ and NBT$_\mathcal{H}$ on the native IA representations. As graphed in Figure 10, the median runtime of NBT$_\mathcal{I}$ and NBT$_\mathcal{H}$ both peaked where the phase transition happens, i.e. $d = 9.5$. In contrast, the runtime peaks of zChaff on the SAT instances were shifted away from the phase transition. The graphs in the middle row of Figure 10 show that the median CPU time of zChaff on the point-based 1-D support, direct and log instances peaked around $d = 9$, 8 and 6, respectively. In addition, the CPU time of zChaff on instances surrounding these peaks was relatively similar, regardless of which SAT encoding scheme was used.

[3] These instances were generated by Nebel's generator, which is available at ftp://ftp.informatik.uni-freiburg.de/documents/papers/ki/allen-csp-solving.programs.tar.gz

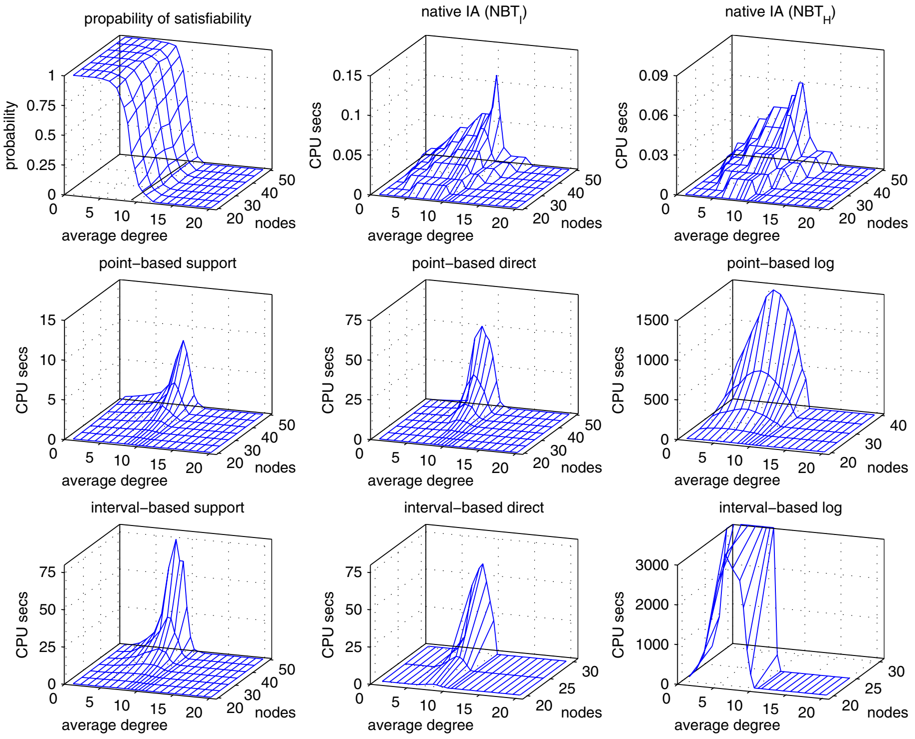


Fig. 10. The phase transition and hardness distribution of NBT$_\mathcal{I}$ and NBT$_\mathcal{H}$ on the native $A(n, d, 6.5)$ IA instances and zChaff on the corresponding SAT-encoded instances (500 instances per data point)

This result is further supported when we take into account the performance of zChaff on the interval-based SAT instances. The graphs in the bottom row of Figure 10 show the median CPU time of zChaff on the corresponding interval-based 1-D support, direct and log instances. Here we can see that the runtime peaks of zChaff are shifted away from the phase transition in exactly the same way as they were on the point-based SAT instances, regardless of which SAT encoding scheme was used. In fact, the CPU time of zChaff on the interval-based direct and log instances peaked at the same points as their corresponding point-based instances, i.e. at $d = 8$ and 6, respectively. The only exception is the runtime of zChaff on the interval-based 1-D support instances which peaked at $d = 8$, i.e. even further away than for the point-based 1-D support instances.

These results are quite surprising and contrast with the results of previous studies on the phase transition behaviour of IA networks [13] and random problems [2,16]. Intuitively, the further left we move from the phase transition, the more solutions an instance has and, as a consequence, the easier this instance should be to solve. However, this conjecture is not true for our SAT encoding schemes. The empirical results clearly show that the hard region, where instances take significantly more time to solve, does not always happen around the phase transition. In contrast, the representation or encoding of the problem instance plays an important role in determining where the hard region will occur.

6 An Empirical Comparison Among SAT Encodings

The graphs in Figure 10 provide strong supporting evidence for the following conjectures:

i) A point-based formulation produces better results than an interval-based formulation, regardless of how IA instances are generated (in terms of the number of nodes n, the average degree d or the average label size s) or the SAT encoding employed.
ii) The 1-D support encoding scheme produces the best results, followed by the direct and log encoding schemes, regardless of how IA instances are generated (in terms of the number of nodes n, the average degree d or the average label size s) or the formulation method employed.
iii) Among the six encoding schemes considered, the point-based 1-D support encoding is the most suitable for translating IA instances into SAT formulae.

The superior performance of the 1-D support encoding can be partly explained by the significantly smaller number of clauses generated (on average about ten times less than for direct or log encoded instances). However, it should be noted that the search space (i.e. the number of variables) of 1-D support and direct encoded instances are the same, whereas the search space of log encoded instances is $O(n \times (|s| - log|s|))$ times smaller [8]. A further possible reason for the superiority of the 1-D support encoding (suggested by Gent [5]) is the reduced bias to falsify clauses, i.e. the numbers of positive and negative literals in the support encoding are more balanced than in a direct encoding and hence this may prevent the search from resetting variables to false shortly after they are set to true.

7 Empirical Evaluation of SAT Versus Existing Approaches

The final question to address in this study is how our SAT approach compares to the existing state-of-the-art specialised approaches. We generated another benchmark test set of $A(n, d, 6.5)$ IA instances by varying the average degree d from 1 to 20 (in steps of 0.5 from 8 to 11 and in steps of 2 otherwise) across nine values of n varied from 60 to 100 (in steps of 5). We generated 100 instances for each n/d data point to obtain a set of $16 \times 9 \times 100 = 14,400$ test instances. This test set allowed us to take a closer look at the performance of different approaches around the phase transition while still providing a general view across the entire distribution. We then ran NBT$_{\mathcal{H}}$ on these instances and zChaff on the corresponding point-based 1-D support SAT instances. All solvers were timed out after one hour for $n < 80$ and four hours for $n \geq 80$.

As shown in Figure 11, the mean CPU time of zChaff was significantly better than the mean CPU time of NBT$_{\mathcal{H}}$ (around 4.35 times at $n = 100$). In addition, when the test instances became bigger (e.g. $n \geq 80$), the time curves of NBT$_{\mathcal{H}}$ were exponentially increased while the time curves of zChaff remained nearly linear. These observations led us to conjecture that zChaff performs better than NBT$_{\mathcal{H}}$ on hard instances and that its performance scales better as the size of the test instances grows. A more thorough analysis of the results produced further evidence to support this conjecture: with a one hour time limit, zChaff was unable to solve 32 of the entire benchmark set of $14,400$ instances, while 323 instances remained unsolved for NBT$_{\mathcal{H}}$ (see Figure 11). When the time limit was raised to four hours, only 2 instances remained unsolvable for zChaff in comparison with 103 for NBT$_{\mathcal{H}}$. This means that the performance of zChaff scaled 51.5 times better than NBT$_{\mathcal{H}}$ on these extremely hard instances.

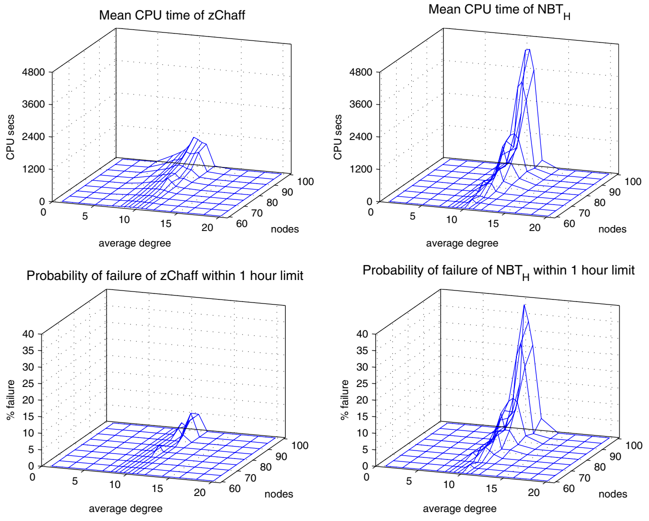


Fig. 11. The CPU time and the probability of failure of zChaff on the point-based 1-D support instances and NBT$_{\mathcal{H}}$ on the native IA instances

8 Summary

In summary, we have proposed six different methods to formulate IA networks into SAT formulae and provided the theoretical proofs of completeness of these transformation techniques. Although our empirical results confirmed that the phase transition of IA networks mapped directly into these SAT encodings, they also showed that the hard regions of these problems were surprisingly shifted away from the phase transition areas after transformation into SAT. Evaluating the effects of these SAT encodings, we found that the point-based 1-D support scheme is the best among the six IA-to-SAT schemes examined. Our results also revealed that zChaff combined with our point-based 1-D support scheme could solve IA instances significantly faster than existing IA solvers working on the equivalent native IA networks.

In future work we anticipate that the performance of our SAT-based approach can be further improved by exploiting the special structure of IA problems in a manner analogous to the work on TSAT [17]. The possibility also opens up of integrating our approach to temporal reasoning into other well known real world problems such as planning. Given the success of SAT solvers in many real world domains, our work promises to expand the reach of temporal reasoning approaches for IA to encompass larger and more practical problems.

Acknowledgments. We thankfully acknowledge the financial support from National ICT Australia (NICTA) and the Queensland government. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative and partly through the Australian Research Council.

References

1. James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.
2. Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *IJCAI-91*, pages 331–337, 1991.
3. Eugene C. Freuder. Synthesizing constraint expressions. *Communication of ACM*, 21(11):958–966, 1978.
4. Alan M. Frisch and Timothy J. Peugniez. Solving non-Boolean satisfiability problems with stochastic local search. In *IJCAI-01*, pages 282–290, 2001.
5. Ian P. Gent. Arc consistency in SAT. In *ECAI-02*, pages 121–125, 2002.
6. K. Ghiathi and G. Ghassem-Sani. Using satisfiability in temporal planning. *WSEAS Transactions on Computers*, 3(4):963–969, 2004.
7. Martin C. Golumbic and Ron Shamir. Complexity and algorithms for reasoning about time: A graph-theoretic approach. *Journal of ACM*, pages 1108–1133, 1993.
8. Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *IJCAI-99*, pages 296–302, 1999.
9. Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *KR-96*, pages 374–384, 1996.
10. Peter Ladkin and Alexander Reinefeld. Effective solution of qualitative interval constraint problems. *Artificial Intelligence*, 57(1):105–124, 1992.
11. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1): 99–118, 1977.
12. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC-01*, pages 530–535, 2001.
13. Bernhard Nebel. Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ORD-Horn class. *Constraints*, 1(3):175–190, 1997.
14. Bernhard Nebel and Hans-Jürgen Bürckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's Interval Algebra. *Journal of ACM*, 42(1):43–66, 1995.
15. Steven Prestwich. Local search on SAT-encoded colouring problems. In *SAT-03*, pages 105–119, 2003.
16. Barbara M. Smith and Martin E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):155–181, 1996.
17. John Thornton, Matthew Beaumont, Abdul Sattar, and Michael Maher. A local search approach to modelling and solving Interval Algebra problems. *Journal of Logic and Computation*, 14(1):93–112, 2004.
18. Peter van Beek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58:297–326, 1992.

19. Peter van Beek and Robin Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–144, 1990.
20. Peter van Beek and Dennis W. Manchak. The design and experimental analysis of algorithms for temporal reasoning. *Journal of Artificial Intelligence Research*, 4:1–18, 1996.
21. Marc Vilain and Henry Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI-86*, pages 377–382, 1986.
22. Toby Walsh. SAT v CSP. In *CP-00*, pages 441–456, 2000.

Decomposition of Multi-operator Queries on Semiring-Based Graphical Models

Cédric Pralet[1,2], Thomas Schiex[2], and Gérard Verfaillie[3]

[1] LAAS-CNRS, Toulouse, France
cpralet@laas.fr
[2] INRA, Castanet Tolosan, France
tschiex@toulouse.inra.fr
[3] ONERA, Centre de Toulouse, France
gerard.verfaillie@onera.fr

Abstract. In the last decades, the Satisfiability and Constraint Satisfaction Problem frameworks were extended to integrate aspects such as uncertainties, partial observabilities, or uncontrollabilities. The resulting formalisms, including Quantified Boolean Formulas (QBF), Quantified CSP (QCSP), Stochastic SAT (SSAT), or Stochastic CSP (SCSP), still rely on networks of local functions defining specific *graphical models*, but they involve queries defined by sequences of distinct *elimination operators* ($\exists$ and $\forall$ for QBF and QCSP, max and $+$ for SSAT and SCSP) preventing variables from being considered in an arbitrary order when the problem is solved (be it by tree search or by variable elimination).

In this paper, we show that it is possible to take advantage of the actual structure of such *multi-operator queries* to bring to light new ordering freedoms. This leads to an improved *constrained induced-width* and doing so to possible exponential gains in complexity. This analysis is performed in a generic semiring-based algebraic framework that makes it applicable to various formalisms. It is related with the *quantifier tree* approach recently proposed for QBF but it is much more general and gives theoretical bases to observed experimental gains.

1 Introduction

Searching for a solution to a Constraint Satisfaction Problem (CSP [1]) is equivalent to searching for an assignment of the problem variables maximizing the quantity given by the constraints conjunction, i.e. to *eliminating* variables using max.[1] As max is the only elimination operator involved in such a *mono-operator* query, variables can be considered in any order. The situation is similar with the Satisfiability problem (SAT) but not with Quantified CSP (QCSP [2]) or Quantified Boolean Formulas (QBF), where min (equivalent to $\forall$) and max (equivalent

[1] Eliminating variables in a set S' with an operator $\oplus$ from a function φ defined on the set $dom(S)$ of assignments of a set of variables S means computing the function $\oplus_{S'} \varphi$ defined by $(\oplus_{S'} \varphi)(A) = \oplus_{A' \in dom(S')} \varphi(A.A')$ for all assignments A of $S - S'$. $\oplus_{S'} \varphi$ synthesizes the information given by φ if we disregard variables in S'.

to $\exists$) operators can alternate, or with Stochastic SAT (SSAT [3]) or Stochastic CSP (SCSP [4]), involving max and + operators: these frameworks define *multi-operator* queries for which the order in which variables can be considered is not free.

To overcome this difficulty, variables are usually considered in an order compatible with the sequence of eliminations (if this sequence is "$\forall x_1, x_2 \, \exists x_3$" for a QCSP, then x_1 and x_2 are considered after x_3 in a variable elimination algorithm). This suffices to obtain the correct result but does not take advantage of all the actual structural features of multi-operator queries. For example, as shown by the *quantifier trees* approach [5] recently introduced for QBF, analyzing hidden structures of "flat" prenex normal form QBF can lead to important gains in terms of solving time.

After the introduction of some notations, we define a generic systematic approach for analyzing the actual *macrostructure* of multi-operator queries by transforming them into a tree of mono-operator ones (Section 3). Being defined in a generic algebraic framework, this approach extends and generalizes the all quantifier tree proposal [5]. It is applicable to multiple formalisms, including QCSP, SSAT, or SCSP. Its efficiency, experienced on QBF with quantifier trees, is interpreted theoretically in terms of a parameter called the *constrained induced-width*. Last, we define on the built macrostructure a generic variable elimination (VE) algorithm exploiting *cluster tree decompositions* [6] (Section 4).

2 Background Notations and Definitions

The domain of values of a variable x is denoted $dom(x)$. By extension, the domain of a set of variables S is $dom(S) = \prod_{x \in S} dom(x)$. A *scoped function* φ on S is a function $dom(S) \to E$. S is called the *scope* of φ and is denoted $sc(\varphi)$.

In order to reason about scoped functions, we need to combine and synthesize the information they express: e.g., to answer a QCSP $\forall x_1, x_2 \, \exists x_3 (\varphi_{x_1,x_3} \wedge \varphi_{x_2,x_3})$, we need to aggregate local constraints using $\wedge$ and to synthesize the result using $\exists$ on x_3 and $\forall$ on x_1, x_2. The operator used to aggregate scoped functions is called a *combination operator* and is denoted $\otimes$. The multiple operators used to synthesize information are called *elimination operators* and are denoted $\oplus$. More precisely, the algebraic structure we consider, defining elimination and combination operators, is a *Multi Commutative Semiring* (MCS).

Definition 1. $(E, \oplus, \otimes)$ *is a* commutative semiring *iff E is a set such that $\oplus$ and $\otimes$ are binary associative, commutative operators on E, $\oplus$ has an identity $0_\oplus \in E$ ($x \oplus 0_\oplus = x$), $\otimes$ has an identity $1_\otimes \in E$ ($x \otimes 1_\otimes = x$), and $\otimes$ distributes over $\oplus$ ($x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$).*[2]

$(E, \{\oplus^i, i \in I\}, \otimes)$ *is a* Multi Commutative Semiring *(MCS) iff for all $i \in I$, $(E, \oplus^i, \otimes)$ is a commutative semiring.*

[2] Compared to other definitions of commutative semirings, $0_\oplus$ is not assumed to be an annihilator for $\otimes$, so that e.g. $(\mathbb{N} \cup \{\infty\}, \max, +)$ is seen as a commutative semiring.

Table 1 shows MCS examples and frameworks in which they are used. There exist many other examples, such as $(E, \{\cap, \cup\}, \cap)$.

Table 1. Examples of MCS (t stands for *true* and f for *false*)

E	$\{\oplus^i, i \in I\}$	$\otimes$	Frameworks
$\mathbb{R}^+ \cup \{\infty\}$	$\{\max, +\}$	$\times$	SSAT [3], SCSP [4], Bayesian networks [7]
$\mathbb{R}^+ \cup \{\infty\}$	$\{\min, \max, +\}$	$\times$	Extended-SSAT [3]
$\mathbb{N} \cup \{\infty\}$	$\{\min, \max\}$	$+$	MDPs based on kappa-rankings [8]
$[0, 1]$	$\{\min, \max\}$	min	possibilistic optimistic MDPs [9]
$\{t, f\}$	$\{\wedge, \vee\}$ (i.e. $\{\forall, \exists\}$)	$\wedge$	QBF, QCSP [2]

Definition 2. *A* graphical model *on a MCS $(E, \{\oplus^i, i \in I\}, \otimes)$ is a pair (V, Φ) where V is a finite set of finite domain variables and Φ is a finite multiset of scoped functions taking values in E and whose scopes are included in V.*

A CSP is a graphical model (V, Φ) where Φ contains constraints on V. We introduce *operator-variables sequences* and *queries* to reason about graphical models.

Definition 3. *Let $\preceq$ be a partial order on V. The set of* linearizations *of $\preceq$, denoted $lin(\preceq)$, is the set of total orders $\preceq'$ on V satisfying $(x \preceq y) \rightarrow (x \preceq' y)$.*

Definition 4. *Let $(E, \{\oplus^i, i \in I\}, \otimes)$ be a MCS. A* sequence of operator-variables *on a set of variables V is defined by $SOV = op1_{S_1} \cdot op2_{S_2} \cdot \ldots \cdot opp_{S_p}$, where $\{S_1, S_2, \ldots, S_p\}$ is a partition of V and $op_j \in \{\oplus^i, i \in I\}$ for all $j \in \{1, \ldots, p\}$. The* partial order $\preceq_{SOV}$ *induced by SOV is given by $S_1 \prec_{SOV} S_2 \prec_{SOV} \ldots \prec_{SOV} S_p$ (it forces variables in S_j to be eliminated before variables in S_i whenever $i < j$). An elimination order $o : x_{o_1} \prec x_{o_2} \prec \ldots \prec x_{o_q}$ on V is a total order on V. It is* compatible with SOV *iff $o \in lin(\preceq_{SOV})$. If $op(x)$ corresponds to the elimination operator of x in SOV, $SOV(o)$ denotes the sequence of operator-variables $op(x_{o_1})_{x_{o_1}} \cdot op(x_{o_2})_{x_{o_2}} \cdot \ldots \cdot op(x_{o_q})_{x_{o_q}}$.*

For the MCS $(\mathbb{R}^+ \cup \{\infty\}, \{\min, \max, +\}, \times)$, a sequence of operator-variables on $V = \{x_1, x_2, x_3, x_4, x_5\}$ is e.g. $SOV = \min_{x_1,x_2} \sum_{x_3,x_4} \max_{x_5}$. The partial order it induces satisfies $\{x_1, x_2\} \prec_{SOV} \{x_3, x_4\} \prec_{SOV} x_5$. The elimination order $o : x_1 \prec x_2 \prec x_4 \prec x_3 \prec x_5$ is compatible with SOV (and $SOV(o) = \min_{x_1} \min_{x_2} \sum_{x_4} \sum_{x_3} \min_{x_5}$), whereas $o' : x_4 \prec x_2 \prec x_1 \prec x_3 \prec x_5$ is not.

Definition 5. *Given a MCS $(E, \{\oplus^i, i \in I\}, \otimes)$, a* query *is a pair $Q = (SOV, \mathcal{N})$ where $\mathcal{N} = (V, \Phi)$ is a graphical model and SOV is a sequence of operator-variables on V. The* answer to a query *is $Ans(Q) = SOV\,(\otimes_{\varphi \in \Phi}\, \varphi)$.*

All the elimination operators considered here being commutative and associative, every elimination order compatible with $\preceq_{SOV}$ can be used to answer a query, i.e. for every $o \in lin(\preceq_{SOV})$, $Ans(Q) = SOV(o)\,(\otimes_{\varphi \in \Phi}\, \varphi)$.

The definition of the answer to a query covers various decision problems raised in many formalisms. Among the multi-operator ones, one can cite:

1. *Quantified Boolean Formulas* in conjunctive prenex normal form and *Quantified CSPs* [2], looking like $\forall x_1, x_2 \exists x_3 \forall x_4\ (\varphi_{x_1,x_3,x_4} \wedge \varphi_{x_2,x_4})$;
2. *Stochastic Satisfaction* problems (SSAT and Extended-SSAT [3]) and some queries on *Stochastic CSPs* [4] looking like $\max_{d_1,d_2} \sum_{s_1} \max_{d_3}(\prod_{\varphi \in \Phi} \varphi)$, where Φ contains both constraints and conditional probability distributions;
3. some types of finite horizon *Markov Decision Processes* (MDPs [10]), on which queries look like $\max_{d_1} \oplus_{s_1} \ldots \max_{d_n} \oplus_{s_n} (\otimes_{\varphi \in \Phi}\, \varphi)$, where $(\oplus, \otimes)$ equals $(+, \times)$ (MDPs optimizing an expected satisfaction), $(\max, \min)$ (optimistic possibilistic MDPs [9]), or $(\max, +)$ (MDPs based on kappa-rankings [8]).

It also covers queries in other frameworks like Bayesian Networks (BN [7]), or in yet unpublished frameworks such as quantified VCSPs (i.e. VCSPs [11] using an alternation of min and max operations on a combination of soft constraints) or semiring CSPs [11] with multiple elimination operators.

As only one combination operator is involved in the definition of the answer to a query, formalisms such as influence diagrams [12], classical probabilistic MDPs [10], or pessimistic possibilistic MDPs [9] are not basically covered but can be if transformed using so-called "potentials" [13]. However, in these cases, more direct efficient approaches can be proposed. See [14] for further details.

3 Macrostructuring a Multi-operator Query

Analyzing the *macrostructure* of queries means bringing to light the actual constraints on the elimination order and the possible decompositions. We first give a parameter, the *constrained induced-width*, for quantifying the complexity of a VE algorithm on multi-operator queries and then show how this complexity can be decreased. This leads us to define a systematic method for structuring an unstructured multi-operator query into a tree of mono-operator ones.

3.1 Constrained Induced-Width

A parameter defining an upper bound on the theoretical complexity of standard VE algorithms on mono-operator queries is the induced-width [15]. In the multi-operator case however, there are constraints on the elimination order because the alternating elimination operators do not generally commute. The complexity can then be quantified using the *constrained induced-width* [16,17] as defined below.

Definition 6. *Let $G = (V_G, H_G)$ be a hypergraph*[3] *and let $\preceq$ be a partial order on V_G. The constrained induced-width $w_G(\preceq)$ of G with constraints on the elimination order given by $\preceq$ ("$x \prec y$" stands for "y must be eliminated before x") is defined by $w_G(\preceq) = \min_{o \in lin(\preceq)} w_G(o)$, $w_G(o)$ being the induced-width of*

[3] V_G is a set of variables and H_G is a set of hyperedges on V_G, i.e. a subset of 2^{V_G}.

G for the elimination order o (i.e. the size of the largest hyperedge created when eliminating variables in the order given by o).[4]

The basic hypergraph associated with a graphical model $\mathcal{N} = (V, \Phi)$ is $G = (V, \{sc(\varphi) \mid \varphi \in \Phi\})$ and the constraints on the elimination order imposed by a query $Q = (SOV, \mathcal{N})$ can be described by $\preceq_{SOV}$ (cf Definition 4). An upper bound on the theoretical complexity of a VE algorithm for answering a query is then $O(|\Phi| \cdot d^{1+w_G(\preceq_{SOV})})$, d being the maximum domain size (for all the complexity results of the paper, we assume that operations like $a \otimes b$ or $a \oplus b$ take a bounded time). Since a linear variation of the constrained induced width yields an exponential variation of the complexity, it is worth working on the two parameters it depends on: the partial order $\preceq_{SOV}$ and the hypergraph G.

Weakening Constraints on the Elimination Order. is known to be useless in contexts like Maximum A Posteriori hypothesis [17], where there is only one alternation of max and sum marginalizations. But it can decrease the constrained induced-width as soon as there are more than two levels of alternation.

Indeed, assume that a Stochastic CSP query is equivalent to computing $\max_{x_1,\ldots,x_q} \sum_y \max_{x_{q+1}} \left(\varphi_y \times \varphi_{y,x_1} \times \prod_{i\in\{1,\ldots,q\}} \varphi_{x_i,x_{q+1}}\right)$ (this may occur if φ_y is a probability distribution on y, the other φ_S model constraints, and the value of y is observed only before making decision x_{q+1}). If one uses $G = (V_G, H_G)$, with $V_G = \{x_1, \ldots, x_{q+1}, y\}$ and $H_G = \{\{y\}, \{y, x_1\}\} \cup \{\{x_i, x_{q+1}\}, i \in \{1, \ldots, q\}\}$, together with $\preceq_1 = \preceq_{SOV}$ ($\{x_1, \ldots, x_q\} \prec_1 y \prec_1 x_{q+1}$), the constrained induced-width is $w_G(\preceq_1) = q$, because x_{q+1} is then necessarily eliminated first (eliminating x_{q+1} from G creates the hyperedge $\{x_1, \ldots, x_q\}$ of size q).

However, the scopes of the functions involved enable us to write the quantity to compute as $\max_{x_1} \left(\left(\sum_y \varphi_y \times \varphi_{y,x_1}\right) \times \left(\max_{x_2,\ldots,x_{q+1}} \left(\prod_{i\in\{1,\ldots,q\}} \varphi_{x_i,x_{q+1}}\right)\right)\right)$. This rewriting shows that the only constraint on the elimination order is that x_1 must be eliminated before y. This constraint, modeled by $\preceq_2$ defined by $x_1 \prec_2 y$, gives $w_G(\preceq_2) = 1$ (e.g. with the elimination order $x_1 \prec x_{q+1} \prec x_2 \prec x_3 \prec \ldots \prec x_q \prec y$). Hence, the complexity decreases from $O((q+2) \cdot d^{1+q})$ to $O((q+2) \cdot d^2)$ (there is a $q+2$ factor because there are $q+2$ scoped functions).

This example shows that defining constraints on the elimination order from the sequence of operator-variables only is uselessly strong and may be exponentially suboptimal compared to a method considering the scopes of the functions involved. It is also obvious that weakening constraints on the elimination order can only decrease the constrained induced-width: if $G = (V_G, H_G)$ is a hypergraph and if $\preceq_1$, $\preceq_2$ are two partial orders on V_G such that $(x \preceq_2 y) \rightarrow (x \preceq_1 y)$ ($\preceq_2$ is weaker than $\preceq_1$), then $w_G(\preceq_1) \geq w_G(\preceq_2)$.

Working on the Hypergraph. There may exist decompositions enabling to use more than just the distributivity of $\otimes$ over $\oplus$.

[4] To be more formal, we should speak of the induced-width of the primal graph of G (the graph containing an edge $\{x, y\}$ iff there exists $h \in H_G$ s.t. $\{x, y\} \subset h$) since the usual definition of the induced-width holds on graphs (and not on hypergraphs).

Indeed, let us consider the QCSP $\exists x_1 \ldots \exists x_q \forall y \left(\varphi_{x_1,y} \wedge \ldots \wedge \varphi_{x_q,y}\right)$. Using $G_1 = (\{x_1, \ldots, x_q, y\}, \{\{x_i, y\}, i \in \{1, \ldots, q\}\})$ and $\preceq_1$ defined by $\{x_1, \ldots, x_q\} \prec_1 y$ gives $w_{G_1}(\preceq_1) = q$ (because y is then necessarily eliminated first). However, it is possible to *duplicate* y and write $\exists x_1 \ldots \exists x_q \forall y \left(\varphi_{x_1,y} \wedge \ldots \wedge \varphi_{x_q,y}\right) = \exists x_1, \ldots, \exists x_q \left((\forall y_1 \varphi_{x_1,y_1}) \wedge \ldots \wedge \left(\forall y_q \varphi_{x_q,y_q}\right)\right)$. The complexity is then given by $G_2 = (\{x_1, \ldots, x_q, y_1, \ldots, y_q\}, \{\{x_i, y_i\}, i \in \{1, \ldots, q\}\})$ and $\preceq_2$ defined by $x_i \prec_2 y_i$, leading to the constrained induced-width $w_{G_2}(\preceq_2) = 1$. Therefore, duplicating y decreases the theoretical complexity from $O(q \cdot d^{q+1})$ to $O(q \cdot d^2)$.

Proposition 1 shows that such a duplication mechanism can be used only in one specific case, applicable for eliminations with $\forall$ on QBF, QCSP, or with min on possibilistic optimistic MDPs. Proposition 2 proves that duplicating is always better than not duplicating.

Proposition 1. *Let $(E, \{\oplus^i, i \in I\}, \otimes)$ be a MCS and let $\oplus \in \{\oplus^i, i \in I\}$. Then, $(\oplus_x (\varphi_1 \otimes \varphi_2) = (\oplus_x \varphi_1) \otimes (\oplus_x \varphi_2)$ for all scoped functions φ_1, $\varphi_2) \leftrightarrow (\oplus = \otimes)$.*

Proof. *If $\oplus = \otimes$, then $\oplus_x (\varphi_1 \oplus \varphi_2) = (\oplus_x \varphi_1) \oplus (\oplus_x \varphi_2)$ by commutativity and associativity of $\oplus$. Conversely, assume that for all scoped functions φ_1, φ_2, $\oplus_x (\varphi_1 \otimes \varphi_2) = (\oplus_x \varphi_1) \otimes (\oplus_x \varphi_2)$. As $(E, \{\oplus^i, i \in I\}, \otimes)$ is a MCS, $\otimes$ has an identity $1_\otimes$ and $\oplus$ has an identity $0_\oplus$. Let us consider a boolean variable x and two scoped functions φ_1, φ_2 of scope x, s.t. $\varphi_1((x,t)) = a$, $\varphi_1((x,f)) = \varphi_2((x,t)) = 1_\otimes$, $\varphi_2((x,f)) = b$. Then, the initial assumption implies that $(a \otimes 1_\otimes) \oplus (1_\otimes \otimes b) = (a \oplus 1_\otimes) \otimes (1_\otimes \oplus b)$, i.e. $a \oplus b = (a \oplus 1_\otimes) \otimes (1_\otimes \oplus b)$. Taking $a = b = 0_\oplus$ gives $0_\oplus = 1_\otimes$. Consequently, for all $a, b \in E$, $a \oplus b = (a \oplus 1_\otimes) \otimes (1_\otimes \oplus b) = (a \oplus 0_\oplus) \otimes (0_\oplus \oplus b) = a \otimes b$, i.e. $\oplus = \otimes$.* $\square$

Note that $\oplus = \otimes$ implies that $\oplus$ is idempotent: indeed, given the properties of a MCS, "$\oplus = \otimes$" implies that $a \oplus a = a \otimes (1_\otimes \oplus 1_\otimes) = a \otimes (1_\otimes \oplus 0_\oplus) = a$.

Proposition 2. *Let $(E, \{\oplus^i, i \in I\}, \otimes)$ be a MCS and let $\oplus \in \{\oplus^i, i \in I\}$. Let φ_{x,S_j} be a scoped function of scope $\{x\} \cup S_j$ for all $j \in \{1, \ldots, m\}$. The direct computation of $\psi = \oplus_x(\varphi_{x,S_1} \otimes \cdots \otimes \varphi_{x,S_m})$ always requires more operations than the one of $(\oplus_x \varphi_{x,S_1}) \otimes \cdots \otimes (\oplus_x \varphi_{x,S_m})$. Moreover, the direct computation of ψ results in a time complexity $O(m \cdot d^{1+|S_1 \cup \ldots \cup S_m|})$, whereas the one of the m quantities in the set $\left\{\oplus_x \varphi_{x,S_j} \,|\, j \in \{1, \ldots, m\}\right\}$ is $O(m \cdot d^{1+\max_{j \in \{1,\ldots,m\}} |S_j|})$.*

Proof. *It can be shown that computing directly $\oplus_x(\varphi_{x,S_1} \otimes \cdots \otimes \varphi_{x,S_m})$ requires $n_1 = |dom(S_1 \cup \ldots \cup S_m)|(m|dom(x)| - 1) = O(md^{1+|S_1 \cup \ldots \cup S_m|})$ operations. Directly computing the quantities in $\left\{\oplus_x \varphi_{x,S_j} | j \in \{1, \ldots, m\}\right\}$ requires $n_2 = (\sum_{j \in \{1,\ldots,m\}} |dom(S_j)|) \cdot (|dom(x)| - 1) = O(m \cdot d^{1+\max_{j \in \{1,\ldots,m\}} |S_j|})$ operations. Directly computing $(\oplus_x \varphi_{x,S_1}) \otimes \cdots \otimes (\oplus_x \varphi_{x,S_m})$ therefore requires $n_3 = n_2 + |dom(S_1 \cup \ldots \cup S_m)|(m-1)$ operations. The result follows from $n_1 - n_3 = (|dom(x)| - 1)(m|dom(S_1 \cup \ldots \cup S_m)| - \sum_{j \in \{1,\ldots,m\}} |dom(S_j)|) \geq 0$.* $\square$

3.2 Towards a Tree of Mono-operator Queries

The constrained induced-width can be decreased and exponential gains in complexity obtained thanks to an accurate multi-operators query analysis. The latter corresponds to determining the actual constraints on the elimination order and the possible additional decompositions using duplication. To systematize it, we introduce rewriting rules transforming an initial unstructured multi-operator query into a tree of mono-operator ones

The basic elements used for such a transformation are computation nodes.

Definition 7. *A computation node n on a MCS $(E, \{\oplus^i, i \in I\}, \otimes)$ is:*

- *either a scoped function φ (*atomic *computation node); the value of n is then $val(n) = \varphi$ and its scope is $sc(n) = sc(\varphi)$;*
- *or a pair (SOV, N) s.t. SOV is a sequence of operator-variables on a set of variables S and N is a set of computation nodes; the value of n is then $val(n) = SOV(\otimes_{n' \in N} val(n'))$, the set of variables it eliminates is $V_e(n) = S$, its scope is $sc(n) = (\cup_{n' \in N} sc(n')) - V_e(n)$, and the set of its sons is $Sons(n) = N$.*

We extend the previous definitions to sets of computation nodes N by $val(N) = \otimes_{n' \in N} val(n')$, $sc(N) = \cup_{n' \in N} sc(n')$, and, if all nodes in N are non-atomic, then $V_e(N) = \cup_{n' \in N} V_e(n')$ and $Sons(N) = \cup_{n' \in N} Sons(n')$. Moreover, for all $\oplus \in \{\oplus^i, i \in I\}$, we define the set of nodes in N performing eliminations with $\oplus$ by $N[\oplus] = \{n \in N \mid n = (\oplus_S, N')\}$.

For example, if $N = \{(\min_{x,y}, N_1), (\sum_z, N_2), (\min_t, N_3)\}$, then $N[\min] = \{(\min_{x,y}, N_1), (\min_t, N_3)\}$ and $N[+] = \{(\sum_z, N_2)\}$. Informally, a computation node (SOV, N) specifies a sequence of eliminations on the combination of its sons and can be seen as the root of a tree of computation nodes. It can be represented as in Figure 1. Given a set of computation nodes N, we define N^{+x} (resp. N^{-x}) as the set of nodes of N whose scope contains x (resp. does not contain x): $N^{+x} = \{n \in N \mid x \in sc(n)\}$ (resp. $N^{-x} = \{n \in N \mid x \notin sc(n)\}$).

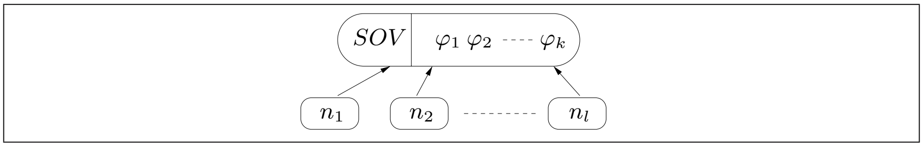


Fig. 1. A computation node (SOV, N). Note that atomic sons (in $N \cap \Phi = \{\varphi_1, \dots, \varphi_k\}$) and non-atomic ones (in $N - \Phi = \{n_1, \dots, n_l\}$) are distinguished.

The value of computation nodes can easily be linked to the answer to a query. Indeed, given a query $Q = (SOV, (V, \Phi))$ defined on a MCS $(E, \{\oplus^i, i \in I\}, \otimes)$, $Ans(Q) = val(n_0)$ where $n_0 = (SOV, \Phi)$. The problem consists in rewriting n_0 so as to exhibit the query structure. To do so, we consider each variable from the right to the left of SOV, using an elimination order o *compatible* with SOV (cf Definition 4), and *simulate* the decomposition induced by the elimination of the $|V|$ variables from the right to the left of $SOV(o)$. More precisely, we start from the initial Computation Nodes Tree (CNT):

$$CNT_0(Q, o) = (SOV(o), \Phi)$$

In the example in Figure 2, this initial CNT corresponds to the first node. For all $k \in \{0, \ldots, |V|-1\}$, the macrostructure at step $k+1$, denoted $CNT_{k+1}(Q,o)$, is obtained from $CNT_k(Q,o)$ by considering the rightmost remaining elimination and by applying two types of rewriting rules:

1. A *decomposition rule DR*, using the distributivity of the elimination operators over $\otimes$ (so that when eliminating a variable x, only scoped functions with x in their scopes are considered) together with possible duplications. Note that DR implements both types of decompositions.

$$\boxed{DR} \quad (sov.\oplus_x, N) \rightsquigarrow \begin{cases} (sov, N^{-x} \cup \{(\oplus_x, \{n\}) \mid n \in N^{+x}\}) \text{ if } \oplus = \otimes \\ (sov, N^{-x} \cup \{(\oplus_x, N^{+x})\}) \text{ otherwise} \end{cases}$$

   In Figure 2, DR transforms the initial structure $CNT_0(Q,o) = (\min_{x_1} \max_{x_2} \max_{x_3} \min_{x_4} \max_{x_5}, \{\varphi_{x_3,x_4}, \varphi_{x_1,x_4}, \varphi_{x_1,x_5}, \varphi_{x_2,x_5}, \varphi_{x_3,x_5}\})$ to $CNT_1(Q,o) = (\min_{x_1} \max_{x_2} \max_{x_3} \min_{x_4}, \{\varphi_{x_3,x_4}, \varphi_{x_1,x_4}, (\max_{x_5}, \{\varphi_{x_1,x_5}, \varphi_{x_2,x_5}, \varphi_{x_3,x_5}\})\})$ (case $\oplus \neq \otimes$). Eliminating x_4 using min $= \otimes$ then transforms $CNT_1(Q,o)$ to $CNT_2(Q,o) = (\min_{x_1} \max_{x_2} \max_{x_3}, \{(\min_{x_4}, \{\varphi_{x_3,x_4}\}), (\min_{x_4}, \{\varphi_{x_1,x_4}\}), (\max_{x_5}, \{\varphi_{x_1,x_5}, \varphi_{x_2,x_5}, \varphi_{x_3,x_5}\})\})$.
2. A *recomposition rule RR* which aims at revealing freedoms in the elimination order for the nodes created by DR.

$$\boxed{RR} \quad (\oplus_x, N) \rightsquigarrow \left(\oplus_{x \cup V_e(N[\oplus])}, (N - N[\oplus]) \cup Sons(N[\oplus])\right)$$

   In Figure 2, RR transforms the computation node $(\min_{x_1} \max_{x_2}, \{(\min_{x_4}, \{\varphi_{x_1,x_4}\}), (\max_{x_3}, \{(\min_{x_4}, \{\varphi_{x_3,x_4}\}), (\max_{x_5}, \{\varphi_{x_1,x_5}, \varphi_{x_2,x_5}, \varphi_{x_3,x_5}\})\})\})$ into $CNT_3(Q,o) = (\min_{x_1} \max_{x_2}, \{(\min_{x_4}, \{\varphi_{x_1,x_4}\}), (\max_{x_3,x_5}, \{(\min_{x_4}, \{\varphi_{x_3,x_4}\}), \varphi_{x_1,x_5}, \varphi_{x_2,x_5}, \varphi_{x_3,x_5}\})\})$, because the structure shows that although $x_3 \prec_{SOV} x_5$, there is actually no need to eliminate x_5 before x_3. RR cannot make one miss a better variable ordering, since what is recomposed will always be decomposable again (using the techniques of Section 4).

More formally, for rewriting rule $RR : n_1 \rightsquigarrow n_2$, let us denote $n_2 = RR(n_1)$. Then, for all $k \in \{0, \ldots, |V|-1\}$, $CNT_{k+1}(Q,o) = rewrite(CNT_k(Q,o))$, where

$$rewrite((sov \cdot \oplus_x, N)) = \begin{cases} (sov, N^{-x} \cup \{RR((\oplus_x, \{n\})), n \in N^{+x}\}) \text{ if } \oplus = \otimes \\ (sov, N^{-x} \cup \{RR((\oplus_x, N^{+x}))\}) \text{ otherwise} \end{cases}$$

This means that when eliminating variable x, we decompose the computations (using duplication if $\oplus = \otimes$), and recompose the created nodes in order to reveal freedoms in the elimination order. At each step, a non-duplicated variable appears *once* in the tree and a duplicated one appears at most *once in each branch* of the tree. The final computation nodes tree, denoted $CNT(Q,o)$, is

$$CNT(Q,o) = CNT_{|V|}(Q,o) = rewrite^{|V|}(CNT_0(Q,o))$$

3.3 Some Good Properties of the Macrostructure Obtained

The Soundness of the Created Macrostructure. is provided by Propositions 3 and 4, which show that the rewriting process preserves nodes value.

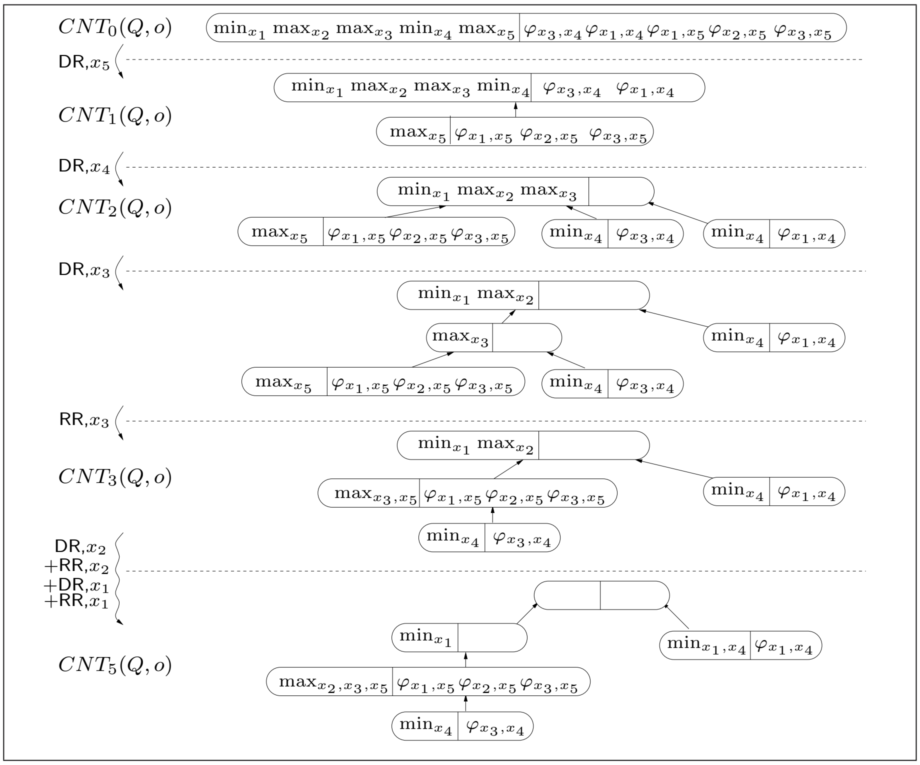

$CNT_0(Q,o)$
DR,x_5
$CNT_1(Q,o)$
DR,x_4
$CNT_2(Q,o)$
DR,x_3
RR,x_3
$CNT_3(Q,o)$
DR,x_2 +RR,x_2 +DR,x_1 +RR,x_1
$CNT_5(Q,o)$
$\min_{x_1} \max_{x_2} \max_{x_3} \min_{x_4} \max_{x_5}$ | $\varphi_{x_3,x_4} \varphi_{x_1,x_4} \varphi_{x_1,x_5} \varphi_{x_2,x_5} \varphi_{x_3,x_5}$
$\max_{x_5}$ | $\varphi_{x_1,x_5} \varphi_{x_2,x_5} \varphi_{x_3,x_5}$
$\min_{x_4}$ | φ_{x_3,x_4}
$\min_{x_4}$ | φ_{x_1,x_4}
$\max_{x_3,x_5}$
$\min_{x_1,x_4}$
$\max_{x_2,x_3,x_5}$

Fig. 2. Application of the rewriting rules on a QCSP example: $\min_{x_1} \max_{x_2,x_3} \min_{x_4} \max_{x_5}(\varphi_{x_3,x_4} \wedge \varphi_{x_1,x_4} \wedge \varphi_{x_1,x_5} \wedge \varphi_{x_2,x_5} \wedge \varphi_{x_3,x_5})$, with the elimination order $o : x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5$

Proposition 3. *Let $Q = (SOV, \mathcal{N})$ be a query and let $o \in lin(\preceq_{SOV})$. Then, $val(CNT_{k+1}(Q,o)) = val(CNT_k(Q,o))$ for all $k \in \{0, \ldots, |V|-1\}$.*

Proof. *We use four lemmas.*

Lemma 1. *Rewriting rule $DR : n_1 \rightsquigarrow n_2$ is sound, i.e. $val(n_1) = val(n_2)$ holds.*

Proof of Lemma 1. As $\otimes$ distributes over $\oplus$, $val((sov{\cdot}\oplus_x, N)) = sov \cdot \oplus_x (\otimes_{n \in N} val(n)) = sov((\otimes_{n \in N^{-x}} val(n)) \otimes \oplus_x (\otimes_{n \in N^{+x}} val(n)))$ (eq1). If $\oplus = \otimes$, Proposition 1 implies that $\oplus_x (\otimes_{n \in N^{+x}} val(n)) = \otimes_{n \in N^{+x}} (\oplus_x val(n)) = val(\{(\oplus_x, \{n\}) \mid n \in N^{+x}\})$. Therefore, using (eq1), $val((sov \cdot \oplus_x, N))$ equals $val\big((sov, N^{-x} \cup \{(\oplus_x, n) \mid n \in N^{+x}\})\big)$. Otherwise ($\oplus \neq \otimes$), one can just write $\oplus_x (\otimes_{n \in N^{+x}} val(n)) = val\big((\oplus_x, N^{+x})\big)$. This means that (eq1) can be written as $val((sov \cdot \oplus_x, N)) = val\big((sov, N^{-x} \cup \{(\oplus_x, N^{+x})\})\big)$.

Lemma 2. *Let $RR' : (\oplus_S, N_1 \cup \{(\oplus_{S'}, N_2)\}) \rightsquigarrow (\oplus_{S \cup S'}, N_1 \cup N_2)$. If $S' \cap (S \cup sc(N_1)) = \emptyset$ and $N_1 \cap N_2 = \emptyset$, then RR' is a sound rewriting rule.*

Proof of Lemma 2. Given that $\otimes$ distributes over $\oplus$ and $S' \cap sc(N_1) = \emptyset$, one can write $val((\oplus_S, N_1 \cup \{(\oplus_{S'}, N_2)\})) = \oplus_S ((\otimes_{n \in N_1} val(n)) \otimes \oplus_{S'} (\otimes_{n \in N_1} val(n))) = \oplus_S \cdot$

$\oplus_{S'}((\otimes_{n\in N_1} val(n)) \otimes (\otimes_{n\in N_1} val(n)))$. As $N_1 \cap N_2 = \emptyset$ and $S \cap S' = \emptyset$, the latter quantity also equals $\oplus_{S\cup S'}(\otimes_{n\in N_1\cup N_2} val(n))$, i.e. $val((\oplus_{S\cup S'}, N_1 \cup N_2))$.

Lemma 3. *$\forall k \in \{0,\ldots,|V|\}\forall n = (sov, N) \in CNT_k(Q,o)$, if $\oplus \neq \otimes$, then for all $n' \in N[\oplus]$, $V_e(n') \cap (V_e((N - \{n'\})[\oplus]) \cup sc(N - \{n'\})) = \emptyset$.*

Proof of Lemma 3. The property holds for $k = 0$ since $CNT_0(Q,o) = (SOV, \Phi)$ and $\Phi[\oplus] = \emptyset$. If it holds at step k, it can be shown to hold at $k+1$ (the main point being that DR splits the nodes with x in their scopes and the ones not having x in their scopes)

Lemma 4. *RR is a sound rewriting rule.*

Proof of Lemma 4. If the variable eliminated uses $\oplus \neq \otimes$ as an operator, then, thanks to Lemma 3 and the fact that all computation nodes are distinct, and since variable x considered at step k satisfies $x \notin V_e(N[\oplus])$, it is possible to recursively apply Lemma 2 to nodes in $N[\oplus]$, because the two conditions looking like $S' \cap (S \cup sc(N_1))$ and $N_1 \cap N_2$ then always hold. This shows that RR is sound when $\oplus \neq \otimes$. If $\oplus = \otimes$, then the nodes to recompose look like $(\oplus_x, \{(\oplus_S, N')\})$. As $S \cap \{x\} = \emptyset$, Lemma 3 entails that RR is sound.

As both DR and RR are sound, Proposition 3 holds. □

Proposition 4. *Let $Q=(SOV,\mathcal{N})$ be a query. Then, $val(CNT(Q,o))=Ans(Q)$ for all $o \in lin(\preceq_{SOV})$.*

Proof. Follows from Proposition 3 and from $val(CNT_0(Q,o)) = Ans(Q)$. □

Independence with Regard to the Linearization of $\preceq_{SOV}$. Proposition 5 shows that the final tree of computation nodes is independent from the arbitrary elimination order o compatible with SOV chosen at the beginning. In this sense, the structure obtained is a *unique fixed point* which can be denoted simply by $CNT(Q)$.

Proposition 5. *Let $Q = (SOV,\mathcal{N})$ be a query. Then, for all $o, o' \in lin(\preceq_{SOV})$, $CNT(Q,o) = CNT(Q,o')$*

Sketch of the proof. (a) It can be shown that for all $\oplus \in \{\oplus^i, i \in I\}$, if $CNT = (sov \cdot \oplus_x \cdot \oplus_y, N)$ and $CNT' = (sov \cdot \oplus_y \cdot \oplus_x, N)$, then $rewrite^2(CNT) = rewrite^2(CNT')$. (b) Given an elimination order $o \in lin(\preceq_{SOV})$, any elimination order $o' \in lin(\preceq_{SOV})$ can be obtained from o by successive permutations of adjacent eliminations. (a) and (b) entail that $CNT(Q,o) = CNT(Q,o')$. □

3.4 Comparison with an Unstructured Approach

Building the macrostructure of a query can induce exponential gains in theoretical complexity, as shown in Section 3.1. Stronger results can be stated, proving that the structured approach is always as least as good as existing approaches in terms of constrained induced-width.

Let us define the width w_n of a node $n = (\oplus_S, N)$ as the induced width of the hypergraph $G = (sc(N), \{sc(n'), n' \in N)$ for the elimination of the variables in S (i.e. the minimum size, among all elimination orders of S, of the largest hyperedge created when eliminating variables in S from G). The induced-width of a tree of computation nodes CNT is $w_{CNT} = \max_{n \in CNT} w_n$. One can say that $1 + w_{CNT}$ is the maximum number of variables to consider simultaneously when using an optimal elimination order in a VE algorithm. Theorem 1 shows that the macrostructuration of a query can only decrease the induced-width.

Theorem 1. *Let $Q = (SOV, (V, \Phi))$ be a query and $G = (V, \{sc(\varphi), \varphi \in \Phi\})$. Then, $w_{CNT(Q)} \leq w_G(\preceq_{SOV})$.*

Sketch of the proof. Let o^* be an elimination order s.t. $w_G(\preceq_{SOV}) = w_G(o^*)$. The idea is to apply the rewriting rules on $CNT_0(Q, o^*)$. Let H_k denote the set of hyperedges in the hypergraph G_k obtained after the k first eliminations in o^*. More precisely, $G_0 = G$ and, if $G_k = (V_k, H_k)$ and x is eliminated, then $G_{k+1} = (V_k - \{x\}, (H_k - H_k^{+x}) \cup \{h_{k+1}\})$, where $h_{k+1} = \cup_{h \in H_k^{+x}} h - \{x\}$ is the hyperedge created from step k to $k+1$. It can be proved that for all $k \in \{0, \ldots, |V| - 1\}$, if $CNT_k(Q, o^*) = (sov \cdot \oplus_x, N)$, then for all $n \in N$, there exists $h \in H_k$ s.t. $sc(n) \subset sc(h)$. This property easily holds at step 0, and if it holds at step k, then $sc((\oplus_x, N^{+x})) \subset sc(h_{k+1})$. Moreover, if duplication is used, then for all $n \in N^{+x}$, $sc((\oplus_x, \{n\})) \subset sc(h_{k+1})$. Rewriting rule RR can be shown to be always advantageous in terms of induced-width. This entails the required result. □

For the QCSP example in Figure 2, $w_{CNT(Q)} = 1$, whereas the initial constrained induced-width is $w_G(\preceq_{SOV}) = 3$ (and without duplication, $w_{CNT(Q)}$ would equal 2): the complexity decreases from $O(|\Phi| \cdot d^4)$ to $O(|\Phi| \cdot d^2)$.

More important gaps between $w_{CNT(Q)}$ and $w_G(\preceq_{SOV})$ can be observed on larger problems. More precisely, we performed experiments on instances of the QBF library (only a limited number are reported here). The results are shown in Table 2. In order to compute induced-widths and constrained induced-widths, we use usual junction tree construction techniques with the so-called *min-fill* heuristic. The results show that there can be no gain in analyzing the macrostructure of queries, as is the case for instances of the "robot" problem (which involve only 3 alternations of elimination operators), but that as soon as the number of alternation increases, revealing freedoms in the elimination order can be greatly beneficial. Note that these results provide a theoretical explanation to the experimental gains observed when using *quantifier trees* on QBF [5].

Theorem 1 shows that working directly on the structure obtained can be a good option, because it can decrease the induced-width. However, given an existing solver, an alternative approach is to see the macrostructuration of a query only as a useful preprocessing step revealing freedoms in the elimination order, thanks to Proposition 6.

Proposition 6. *Let $Q = (SOV, (V, \Phi))$ be a query. Assume that duplication is not used. $CNT(Q)$ induces a partial order $\preceq_{CNT(Q)}$ on V, defined by "if $((\oplus_{S_1}, N \cup \{(\oplus'_{S_2}, N')\}) \in CNT(Q))$, then for all $x \in S_1 \cap sc(N')$, $x \prec_{CNT(Q)} S_2$. Then, for all $o \in lin(\preceq_{CNT(Q)})$, $SOV(o)\,(\otimes_{\varphi \in \Phi}\, \varphi) = Ans(Q)$. Moreover, $\preceq_{CNT(Q)}$ is weaker than $\preceq_{SOV}$.*

Sketch of the proof. The idea is that if $o \in lin(\preceq_{CNT(Q)})$, it is possible to do the inverse operations of RR and DR, considering first smallest variables in o. These

Table 2. Comparison between $w = w_{CNT(Q)}$ and $w' = w_G(\preceq_{SOV})$ on some instances of the QBF library (*nbv*, *nbc*, *nba* denote respectively the number of variables, the number of clauses, and the number of elimination operator alternations of an instance)

Problem instance	w	w'	nbv,nbc,nba	Problem instance	w	w'	nbv,nbc,nba
adder-2-sat	12	24	$332, 113, 5$	k-branch-n-1	22	43	$133, 314, 7$
adder-4-sat	28	101	$726, 534, 5$	k-branch-n-2	39	103	$294, 793, 9$
adder-8-sat	60	411	$1970, 2300, 5$	k-branch-n-3	54	185	$515, 1506, 11$
adder-10-sat	76	644	$2820, 3645, 5$	k-branch-n-4	70	296	$803, 2565, 13$
adder-12-sat	92	929	$3822, 5298, 5$	k-branch-n-5	89	427	$1149, 3874, 15$
robots-1-5-2-1.6	2213	2213	$6916, 23176, 3$	k-branch-n-6	107	582	$1557, 5505, 17$
robots-1-5-2-1.7	1461	1461	$7904, 26810, 3$	k-branch-n-7	131	761	$2027, 7482, 19$
robots-1-5-2-1.8	3933	3933	$8892, 30444, 3$	k-branch-n-8	146	973	$2568, 10117, 21$
robots-1-5-2-1.9	1788	1788	$9880, 34078, 3$	k-branch-n-9	166	1201	$3163, 12930, 23$

inverse operations are naturally sound and lead to the structure $(SOV(o), \Phi)$, which proves that $SOV(o)\,(\otimes_{\varphi \in \Phi}\, \varphi) = Ans(Q)$.

If $o \in lin(\preceq_{SOV})$ and $x \preceq_o y$, then, for all $n = (\oplus_{S_1}, N \cup \{(\oplus'_{S_2}, N')\}) \in CNT(Q) = CNT(Q, o)$, it is impossible that $y \in S_1$ and $x \in S_2$ (because y is considered before x during the rewriting process). As this holds for all x, y such that $x \preceq_o y$, this entails that $\neg(y \preceq_{CNT(Q)} x)$. $(x \preceq_o y) \rightarrow \neg(y \preceq_{CNT(Q)} x)$ can also be written $(y \preceq_{CNT(Q)} x) \rightarrow (y \prec_o x)$, which implies that $o \in lin(CNT(Q))$. Therefore, $lin(\preceq_{SOV}) \subset lin(CNT(Q))$, i.e. $\preceq_{CNT(Q)}$ is weaker than $\preceq_{SOV}$ □

3.5 Complexity Results

The macrostructure is usable only if its computation is tractable. Based on the algorithm in Figure 3, implementing the macrostructuration of a query, Proposition 7 gives an upper bound on the complexity, showing that rewriting a query as a tree of mono-operator computation nodes is easy.

In the algorithm in Figure 3, the root node of the tree of computation nodes is rewritten. With each node $n = (op_S, N)$ are associated an operator $op(n) = op$, a set of sons $Sons(n) = N$ modeled as a list, and a set of variables eliminated $V_e(n) = S$ modeled as a list too. The scope of n is modeled using a table of $|V|$ booleans. As long as the sequence of operator-variables is not empty, the rightmost remaining elimination is considered. The pseudo-code just implements the rewrite function, which dissociates the cases $\oplus \neq \otimes$ and $\oplus = \otimes$.

Proposition 7. *The time and space complexity of the algorithm in Figure 3 are $O(|V|^2 \cdot |\Phi|)$ and $O(|V| \cdot |\Phi|)$ respectively (if $\Phi \neq \emptyset$ and $V \neq \emptyset$).*

Proof. *At each rewriting step and for each son n' of the root node, tests like "$x \in sc(n')$" and operations like "$sc(n) \leftarrow sc(n) \cup sc(n')$" or "$sc(n') \leftarrow sc(n') - \{x\}$" are $O(|V|)$, since a scope is represented as a table of size $|V|$. Operations like "$Sons(root) \leftarrow Sons(root) - \{n'\}$", "$Sons(root) \leftarrow Sons(root) \cup \{n\}$", "$V_e(n) \leftarrow V_e(n) \cup V_e(n')$" (with $V_e(n) \cap V_e(n') = \emptyset$), or "$V_e(n) \leftarrow V_e(n) \cup \{x\}$" are $O(1)$, since V_e and $Sons$ are represented as lists. Therefore, the operations performed for each rewriting step and for each son of the root are $O(|V|)$. As at each step, $|Sons(root)| \leq |\Phi|$, and as there are*

```
begin
    $root \leftarrow$ newNode$(\emptyset, \emptyset, \Phi, \emptyset)$
    while $(SOV = SOV' \cdot \oplus_x)$ do
        $SOV \leftarrow SOV'$
        if $\oplus \neq \otimes$ then
            $n \leftarrow$ newNode$(\oplus, \{x\}, \emptyset, \emptyset)$
            foreach $n' \in Sons(root)$ s.t. $x \in sc(n')$ do
                $sc(n) \leftarrow sc(n) \cup sc(n')$
                $Sons(root) \leftarrow Sons(root) - \{n'\}$
                if $op(n') = \oplus$ then
                    $V_e(n) \leftarrow V_e(n) \cup V_e(n')$
                    $Sons(n) \leftarrow Sons(n) \cup Sons(n')$
                else $Sons(n) \leftarrow Sons(n) \cup \{n'\}$
            $sc(n) \leftarrow sc(n) - \{x\}$
            $Sons(root) \leftarrow Sons(root) \cup \{n\}$
        else
            foreach $n' \in Sons(root)$ s.t. $x \in sc(n')$ do
                if $op(n') = \oplus$ then
                    $V_e(n') \leftarrow V_e(n') \cup \{x\}$
                    $sc(n') \leftarrow sc(n') - \{x\}$
                else
                    $n \leftarrow$ newNode$(\oplus, \{x\}, \{n'\}, sc(n') - \{x\})$
                    $Sons(root) \leftarrow (Sons(root) - \{n'\}) \cup \{n\}$
    return $(root)$
end
```

Fig. 3. MacroStruct$(SOV, (V, \Phi))$ (instruction newNode$(op, V_e, Sons, sc)$ creates a computation node $n = (op_{V_e}, Sons)$ and sets $sc(n)$ to sc

$|V|$ rewriting steps, the algorithm is time $O(|V|^2 \cdot |\Phi|)$. As for the space complexity, given that only the scopes of the root sons are used, we need a space $O(|V| \cdot |\Phi|)$ for the scopes. As it can be shown that the number of nodes in the tree of computation nodes is always $O(|V| + |\Phi|)$, recording $op(n)$ and $Sons(n)$ for all nodes n is $O(|V| + |\Phi|)$ too. Last, recording $V_e(n)$ for all nodes n is $O(|V| \cdot |\Phi|)$ because the sum of the number of variables eliminated in each node is lesser than $|V| \cdot |\Phi|$ (the worst case occurs when all variables are duplicated). Hence, the overall space complexity is $O(|V| \cdot |\Phi|)$. □

4 Decomposing Computation Nodes

4.1 From Computation Nodes to Multi-operator Cluster Trees

Once the macrostructure is built (in the form of a tree of mono-operator computation nodes), we use freedoms in the elimination order so as to minimize the induced-width. As $(E, \oplus, \otimes)$ is a commutative semiring for every $\oplus \in \{\oplus^i, i \in I\}$, this can be achieved by decomposing each mono-operator computation node into a cluster tree using usual cluster tree construction techniques. This cluster tree is obtained by considering for each computation node $n = (op_S, N)$ the hypergraph $G(n) = (\cup_{n \in N} sc(n), \{sc(n), n \in N\})$ associated with it.

The structure obtained then contains both a macrostructure given by the computation nodes and an internal cluster tree structure given by each of their decompositions. It is then sufficient to choose a root in the cluster tree decomposition [6] of each computation node to obtain a so-called *multi-operator cluster tree* as in Figure 4 (corresponding to an Extended-SSAT [3] problem).

Definition 8. *A Multi-operator Cluster Tree (MCTree) on a MCS* $(E, \{\oplus^i, i \in I\}, \otimes)$ *is a tree where every vertex c (called a cluster) is labeled with four elements: a set of variables* $V(c)$*, a set of scoped functions* $\Phi(c)$ *taking values in E, a set of son clusters* $Sons(c)$*, and an elimination operator* $\oplus(c) \in \{\oplus^i, i \in I\}$*. The value of a cluster c is* $val(c) = \underset{V(c)-V(pa(c))}{\oplus(c)} \left(\left(\underset{\varphi \in \Phi(c)}{\otimes} \varphi \right) \otimes \left(\underset{s \in Sons(c)}{\otimes} val(s) \right) \right)$.

It follows from the construction process that if r is the root node of the MCTree associated with a query Q, $val(r) = Ans(Q)$.

4.2 A Generic Variable Elimination Algorithm on MCTrees

To define a generic VE algorithm on a MCTree, it suffices to say that as soon as a cluster c has received $val(s)$ from all its children $s \in Sons(c)$, it computes

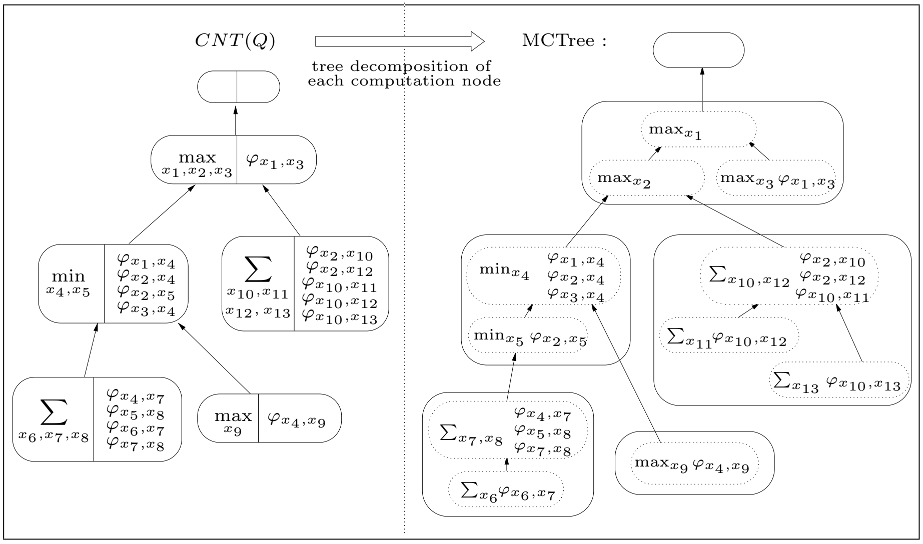

CNT(Q)
tree decomposition of each computation node
MCTree :
max x1,x2,x3 φx1,x3
min x4,x5 φx1,x4 φx2,x4 φx2,x5 φx3,x4
Σ x10,x11 x12,x13 φx2,x10 φx2,x12 φx10,x11 φx10,x12 φx10,x13
Σ x6,x7,x8 φx4,x7 φx5,x8 φx6,x7 φx7,x8
max x9 φx4,x9
maxx1
maxx2
maxx3 φx1,x3
minx4 φx1,x4 φx2,x4 φx3,x4
minx5 φx2,x5
Σx10,x12 φx2,x10 φx2,x12 φx10,x11
Σx11 φx10,x12
Σx13 φx10,x13
Σx7,x8 φx4,x7 φx5,x8 φx7,x8
Σx6 φx6,x7
maxx9 φx4,x9

Fig. 4. Example of a MCTree obtained from $CNT(Q)$. Note that a cluster c is represented by 1) the set $V(c) - V(pa(c))$ of variables it eliminates, its elimination operator $op(c)$, and the set of function $\Phi(c)$ associated with it, all these elements being put in a pointwise box; 2) the set of its sons, pointing to it in the structure.

its own value $val(c) = \oplus(c)_{V(pa(c))-V(c)} \left(\left(\otimes_{\varphi \in \Phi(c)} \varphi \right) \otimes \left(\otimes_{s \in Sons(c)} val(s) \right) \right)$ and sends it to $pa(c)$, its parent in the MCTree. The value of the root cluster then equals the answer to the query.

5 Conclusion

Solving multi-operator queries using only the sequence of elimination to define constraints on the elimination order is easy but does not take advantage of the actual structure of such queries. Performing a preprocessing finer analysis taking into account both the function scopes and operator properties can reveal extra freedoms in the elimination order as well as decompositions using more than just the distributivity of the combination operator over the elimination operators. This analysis transforms an initial unstructured multi-operator query into a tree of mono-operator computation nodes. The obtained *macrostructure* is always as least as good as the unstructured query in terms of induced-width, which can induce exponential gains in complexity. It is then possible to define a generic VE algorithm on Multi-operator Cluster Trees (MCTrees) by building a cluster-tree decomposition of each mono-operator computation node. Performing such a work using generic algebraic operators makes it applicable to various frameworks (QBF, QCSP, SCSP, SSAT, BN, MDPs).

Other algorithms than VE could be designed on MCTrees, such as a tree search enhanced by branch and bound techniques, e.g. in an AND/OR search [18] or a backtrack bounded by tree decomposition (BTD-like [19]) scheme. Ideas from the game theory field like the alpha-beta algorithm [20] can also be considered. This work was partially conducted within the EU IP COGNIRON ("The Cognitive Companion") funded by the European Commission Division FP6-IST Future and Emerging Technologies under Contract FP6-002020.

References

1. Mackworth, A.: Consistency in Networks of Relations. Artificial Intelligence **8** (1977) 99–118
2. Bordeaux, L., Monfroy, E.: Beyond NP: Arc-consistency for Quantified Constraints. In: Proc. of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02), Ithaca, New York, USA (2002)
3. Littman, M., Majercik, S., Pitassi, T.: Stochastic Boolean Satisfiability. Journal of Automated Reasoning **27** (2001) 251–296
4. Walsh, T.: Stochastic Constraint Programming. In: Proc. of the 15th European Conference on Artificial Intelligence (ECAI-02), Lyon, France (2002)
5. Benedetti, M.: Quantifier Trees for QBF. In: Proc. of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-05), St. Andrews, Scotland (2005)
6. Kjaerulff, U.: Triangulation of Graphs - Algorithms Giving Small Total State Space. Technical Report R 90-09, Aalborg University, Denmark (1990)
7. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1988)

8. Giang, P., Shenoy, P.: A Qualitative Linear Utility Theory for Spohn's Theory of Epistemic Beliefs. In: Proc. of the 16th International Conference on Uncertainty in Artificial Intelligence (UAI-00), Stanford, California, USA (2000) 220–229
9. Sabbadin, R.: A Possibilistic Model for Qualitative Sequential Decision Problems under Uncertainty in Partially Observable Environments. In: Proc. of the 15th International Conference on Uncertainty in Artificial Intelligence (UAI-99), Stockholm, Sweden (1999)
10. Puterman, M.: Markov Decision Processes, Discrete Stochastic Dynamic Programming. John Wiley & Sons (1994)
11. Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., Fargier, H.: Semiring-Based CSPs and Valued CSPs: Frameworks, Properties and Comparison. Constraints **4** (1999) 199–240
12. Howard, R., Matheson, J.: Influence Diagrams. In: Readings on the Principles and Applications of Decision Analysis. Menlo Park, CA, USA (1984) 721–762
13. Ndilikilikesha, P.: Potential Influence Diagrams. International Journal of Approximated Reasoning **10** (1994) 251–285
14. Pralet, C., Verfaillie, G., Schiex, T.: From Influence Diagrams to Multioperator Cluster DAGs. In: Proc. of the 22nd International Conference on Uncertainty in Artificial Intelligence (UAI-06), Cambridge, MA, USA (2006)
15. Dechter, R., Fattah, Y.E.: Topological Parameters for Time-Space Tradeoff. Artificial Intelligence **125** (2001) 93–118
16. Jensen, F., Jensen, F., Dittmer, S.: From Influence Diagrams to Junction Trees. In: Proc. of the 10th International Conference on Uncertainty in Artificial Intelligence (UAI-94), Seattle, WA, USA (1994) 367–373
17. Park, J., Darwiche, A.: Complexity Results and Approximation Strategies for MAP Explanations. Journal of Artificial Intelligence Research **21** (2004) 101–133
18. Marinescu, R., Dechter, R.: AND/OR Branch-and-Bound for Graphical Models. In: Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), Edinburgh, Scotland (2005)
19. Jégou, P., Terrioux, C.: Hybrid Backtracking bounded by Tree-decomposition of Constraint Networks. Artificial Intelligence **146** (2003) 43–75
20. Knuth, D., Moore, R.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence **8** (1975) 293–326

Dynamic Lex Constraints

Jean-François Puget

ILOG, 9 avenue de Verdun, 94253 Gentilly, France
puget@ilog.fr

Abstract. Symmetries are one of the difficulties constraint programming users have to deal with. One way to get rid of symmetries is to add lex constraints. However, it can adversely affect the efficiency of a tree search method if the lex constraints remove the solution that would have been found at the first place. We propose to use an alternative filtering algorithm which does not exclude the first solution. We present both a theoretical analysis and some experimental evidence that it is as efficient as lex constraints. We also show that its efficiency does not depend much on the variable ordering used in the tree search. Last, we show that it can prune more nodes than the SBDS method.

1 Introduction

Symmetries are mappings of a Constraint Satisfaction Problem (CSP) onto itself that preserve its structure as well as its solutions. If a CSP has some symmetry, then all symmetrical variants of every dead end encountered during the search may be explored before a solution can be found. Even if the problem is easy to solve, all symmetrical variants of a solution are also solutions, and listing all of them may just be impossible in practice. Breaking symmetry methods try to cure these issues.

Adding symmetry breaking constraints is one of the oldest ways of breaking variable symmetries for constraint satisfaction problems (CSPs)[12]. For instance, it is shown in [3] that all variable symmetries could be broken by adding one lexicographical ordering constraint per symmetry.

Adding lexicographic constraints can be quite efficient for breaking symmetries. It can also be quite inefficient when the symmetry breaking constraints remove the solution that would have been found first by the search procedure. There is a complex interaction between the order used in the search and the order used in stating the lex constraints [19].

Other symmetry breaking methods such as SBDD[5][4][7][14], SBDS [1][8][6], and GE-tree [17] do not interfere with the order used during search. These methods modify the search in order to avoid the exploration of nodes that are symmetrical to already explored nodes. We will denote them as SBDX.

Various experiments, e.g. [13][15][16], have shown that adding lex constraints is often much more efficient than SBDX. However, it is also known that adding constraint can be much less efficient when the interaction with the search order is not good.

We explore in this paper a middle ground between lex constraints and SBDX methods. After some definitions in Sect. 2, we revisit lex constraints and lex leader solutions in Sect. 3. Then, we introduce dynamic lex leader solutions in Sect. 4. We show that the first solution found in any search tree is a dynamic lex leader solution. Section 5 presents an effective filtering algorithm for removing all solutions that are not dynamic lex leader solutions. Section 6 provides some experimental evidence that it can be more effective than SBDS. These experiments also show it is as effective as using lex constraints when a good search order is used, and much better when the search order is modified. Section 7 summarizes our findings and discusses some future research areas.

2 Notations

We denote the set of integers ranging from 0 to $n-1$ by I^n.

A *constraint satisfaction problem* $\mathcal{P}$ (CSP) with n variables is a triple $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a finite set of variables $(v_i)_{i \in I^n}$, $\mathcal{D}$ a finite set of finite sets $dom(v_i)_{i \in I^n}$, and every constraint in $\mathcal{C}$ is a subset of the cross product $\bigotimes_{i \in I^n} dom(v_i)$. The set $dom(v_i)$ is called the *domain* of the variable v_i. Without loss of generality, we can assume that $dom(v_i) \subseteq I^k$ for some k.

The order in which variables appear in a (partial) assignment or in a solution is meaningful in the context of this paper. A *literal* is an equality $(v_j = a_j)$ where $a_j \in dom(v_j)$. An *assignment* is a sequence of literals such that the sequence of the variables in it is a permutation of the sequence of the variable v_i. A *partial assignment* is a sub sequence of an assignment.

A *solution* to $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is an assignment that is consistent with every member of $\mathcal{C}$.

The symmetries we consider are permutations, i.e. one to one mappings (bijections) from a finite set onto itself. Let S^n be the set of all permutations of the set I^n. The image of i by the permutation σ is denoted i^σ. A permutation $\sigma \in S^n$ is fully described by the vector $[0^\sigma, 1^\sigma, \ldots, (n-1)^\sigma]$. The product of two permutations σ and θ is defined by $i^{(\sigma\theta)} = (i^\sigma)^\theta$.

A *symmetry* is a bijection from literals to literals that map solutions to solutions. Our definition is similar to the semantic symmetries of [2].

Given a permutation σ of I^n, we define a variable permutation on (partial) assignments as follows :

$$((v_i = a_i)_{i \in I^n})^\sigma = (v_{i^\sigma} = a_i)_{i \in I^n}$$

Such permutation is called a *variable symmetry* if it maps solutions to solutions.

Given a permutation θ of I^k, we define a value permutation on (partial) assignments as follows :

$$((v_i = a_i)_{i \in I^n})^\theta = (v_i = (a_i)^\theta)_{i \in I^n}$$

Such permutation is called a *value symmetry* if it maps solutions to solutions.

The symmetries of a CSP form a mathematical group. The variable symmetries of a CSP form a sub group of its group of symmetries. The value symmetries of a CSP form a sub group of its group of symmetries.

3 Lex Leader Solutions

A very powerful symmetry breaking method has been proposed in [3]. The idea is to use a lexicographic order to compare solutions. Given two finite sequences $X = (x_0, x_1, \ldots, x_{n-1})$ and $Y = (y_0, y_1, \ldots, y_{n-1})$, we say that X is *lex smaller* than Y (denoted $X \preceq Y$) if, and only if :

$$\forall k \in I^n, \quad (x_0 = y_0 \wedge \ldots \wedge x_{k-1} = y_{k-1}) \Rightarrow x_k \leq y_k \tag{1}$$

Let us consider a solution $(v_i = a_i)_{i \in I^n}$ of the CSP. Let us consider the set of all solutions that are symmetric to it. These solutions are $(v_i = a_i)^{\sigma}_{i \in I^n}$ where σ ranges over the group of symmetries of the CSP. Among all these solutions, there is one that is lexicographically smaller than the others. This solution S satisfies the constraint :

$$\forall \sigma \ \in \ G, \ S \preceq S^{\sigma} \tag{2}$$

The above has been widely used for variable symmetries. We have shown in [16] that any combination of variable and value symmetries could also be broken with a combination of lex constraints and element constraints.

4 Dynamic Lex Leader Solutions

The lex constraints use a fixed variable ordering. Let us define a new lexicographic comparison that uses the order in which variables are selected in the search tree. In order to do so, let us formalize tree search.

A variable is selected at each non leaf node. Then, one branch is created for every value in the domain of this variable. We identify a node with the variable assignments that are true at the node. Variables are listed in the order in which they have been assigned during search.

Constraints can prune the tree : some nodes are inconsistent. These nodes have no children. Solutions are leaves of the search tree that are not inconsistent.

Some constraint propagation algorithm may be applied at every node. It may result in some assignment of variables. For reasons that will become clear later, we introduce a sequence of child node, one for each assignment.

Let us look at a simple CSP with four variables :

$$v_0 \in I^3, v_1 \in I^3, v_2 \in I^3, v_3 \in I^6, \text{AllDifferent}(v_1, v_2, v_3), v_3 = v_1 + v_2 \tag{3}$$

A possible search tree for this CSP is given in Fig. 1. We have used a dynamic order. For instance, the second variable to be selected is v_2 in the leftmost branch, and it is v_1 in the rightmost branch. The leftmost leaf corresponds to the solution :

$$(v_0 = 0, v_2 = 1, v_3 = 3, v_1 = 2) \tag{4}$$

The rightmost leaf corresponds to the solution :

$$(v_0 = 2, v_1 = 1, v_2 = 0, v_3 = 1) \tag{5}$$

Note that the variables appear in a different order in these two solutions.

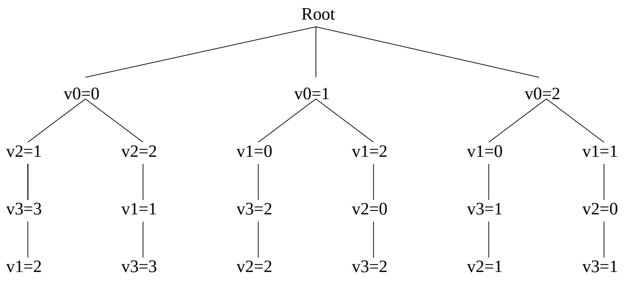


Fig. 1. A tree search

At the node $v_0 = 0, v_2 = 1$, constraint propagation assigns 2 to v_2 and 3 to v_3. This is modeled by creating a sequence of two child nodes for that node. The order in which these extra nodes are created is arbitrary.

Given $SX = (x_i = a_i)_{i \in I^n}$ and $SY = (y_i = b_i)_{i \in I^n}$, then we say that SX is *dyn smaller* than SY if and only if $(a_0, a_1, \ldots, a_{n-1}) \preceq (b_0, b_1, \ldots, b_{n-1})$. It is denoted $SX \preceq^{\mathrm{d}} SY$.

For instance, the solution (4) is dyn smaller than the solution (5) because $(0, 1, 3, 2) \preceq (2, 1, 0, 1)$

Let us state our first result :

Lemma 1. *Given two distinct solutions SX and SY, then either $SX \preceq^{\mathrm{d}} SY$ or $SY \preceq^{\mathrm{d}} SX$, but not both of them.*

Proof. We are given two distinct solutions $SX = (x_i = a_i)_{i \in I^n}$ and $SY = (y_i = b_i)_{i \in I^n}$ such that SX is reached before SY. Let $A = (z_0 = c_0, z_1 = c_1, \ldots, z_{k-1} = c_{k-1})$ be their deepest common ancestor (or maximal common partial assignment). Then, by definition, we have $x_j = z_j = y_j$ for all $j < k$, and $a_j = c_j = b_j$. Let z_k be the variable selected for branching at node A. Then $x_k = z_k = y_k$. The values assigned to z_k in SX is a_k, and the value assigned to z_k in SY is b_k. These values are different since A is the deepest common ancestor of SX and SY. Therefore, either $a \leq b$ or $b \leq a$, but not both of them. The former means $SX \preceq^{\mathrm{d}} SY$, while the latter means $SY \preceq^{\mathrm{d}} SX$. □

We can now state our first theoretical result.

Theorem 2. *The relation $\preceq^{\mathrm{d}}$ is a total order relation.*

Proof. Lemma 1 implies that $\preceq^{\mathrm{d}}$ is a total relation. We need to prove that it is an order, i.e. we need to prove that it is reflexive, antisymmetric, and transitive.

Note first that $\preceq$ is an order relation.

$\preceq^{\mathrm{d}}$ is reflexive, because $S \preceq^{\mathrm{d}} S$ is trivially true.

Let us prove $\preceq^{\mathrm{d}}$ is antisymmetric. We are given two solutions $SX = (x_i = a_i)_{i \in I^n}$ and $SY = (y_i = b_i)_{i \in I^n}$. Suppose that $SX \preceq^{\mathrm{d}} SY$ and $SY \preceq^{\mathrm{d}} SX$. By lemma 1, only one of these can be true if SX and SY are distinct. Therefore, SX and SY must be equals. □

Let us prove $\preceq^{\mathrm{d}}$ is transitive. We are given three solutions $SX = (x_i = a_i)_{i \in I^n}$, $SY = (y_i = b_i)_{i \in I^n}$ and $SZ = (z_i = c_i)_{i \in I^n}$ such that $SX \preceq^{\mathrm{d}} SY$ and $SY \preceq^{\mathrm{d}} SZ$.
Then, we have $(a_0, a_1, \ldots, a_{n-1}) \preceq (b_0, b_1, \ldots, b_{n-1})$ and $(b_0, b_1, \ldots, b_{n-1}) \preceq (c_0, c_1, \ldots, c_{n-1})$.
Since $\preceq$ is transitive, we have $(a_0, a_1, \ldots, a_{n-1}) \preceq (c_0, c_1, \ldots, c_{n-1})$, i.e. $SX \preceq^{\mathrm{d}} SZ$. □

Given a solution S, let us look at the set of all solutions that are symmetrical to it. This set is called the *orbit* of S :

$$\mathit{orbit}(S) = \{S^\sigma \mid \sigma \in G\} \text{ where } G \text{ is the group of symmetries of the CSP.}$$

Since $\preceq$ is a total order relation, there is a unique element in the orbit of S which is dyn smaller than all the others. By definition, this element is equal to S if, and only if, S satisfies the following constraint :

$$\forall \sigma \in G,\ S \preceq^{\mathrm{d}} S^\sigma \tag{6}$$

In such case, we say that S is a *dynamic lex leader* solution.

In other words, adding constraints (6) keeps only one solution in every orbit : the dynamic lex leader solution.

The constraint (6) is somewhat similar to (2). However, in (2), a static variable ordering is used in the $\preceq$ constraints. In (6), the order used is the one of the tree search, which can be dynamic.

We have made no hypothesis on how the search tree was traversed until now. Let us assume from now on that it is explored in a depth first search manner. Let us assume further that values for any given variable are explored in an increasing manner. For instance, the tree of Fig. 2. would be explored from left to right under these assumptions. Then, solutions are generated in the order defined by $\preceq^{\mathrm{d}}$:

Lemma 3. *Let us assume that the tree is explored in a depth first search manner and that values are explored in an increasing order. Then a solution SX is found before a solution SY if, and only if, $SX \preceq^{\mathrm{d}} SY$.*

Proof. We are given two distinct solutions $SX = (x_i = a_i)_{i \in I^n}$ and $SY = (y_i = b_i)_{i \in I^n}$. Let $A = (z_0 = c_0, z_1 = c_1, \ldots, z_{k-1} = c_{k-1})$ be their deepest common ancestor. Then, by definition, we have $x_j = z_j = y_j$ for all $j < k$, and $a_j = c_j = b_j$. Let z_k be the variable selected for branching at node A. Then $x_k = z_k = y_k$. The values assigned to z_k in SX is a_k, and the value assigned to z_k in SY is b_k. Then $a_k < b_k$ since the values are tried in increasing order. Therefore, $SX \preceq^{\mathrm{d}} SY$. □

It has an extremely interesting consequence : the first solution reached by the tree search is a dynamic lex leader solution :

Theorem 4. *Let us assume that the tree is explored in a depth first search manner and that values are explored in an increasing order. Then, the first solution found in each solution orbit is a dynamic lex leader solution.*

Proof. Let S be the first solution found in a given orbit. If S is not a dynamic lex leader solution, then, there exists S' in the orbit of S such that $S' \preceq^{\mathrm{d}} S$. Then, by Lemma 3, S' must be reached before S. □

Theorem 4 basically says that adding the constraint (6) will not interfere with the search. Indeed, this constraint will not remove the first solution found by the search in each orbit.

5 Filtering Dynamic Lex Constraints

In order to generate only dynamic lex leader constraints, one can check the consistency of the constraint (6) at each leaf of the search tree.

We can do better by developing a propagation algorithm for this constraint. Let us assume first that all symmetries are the composition of a variable symmetry and a value symmetry. We have shown in [16] that the effect of such symmetries could be modeled using permutations and element constraints.

5.1 Representation of Symmetry Effect

An element constraint has the following form :

$$y = A[x]$$

where $A = [a_0, a_1, \ldots, a_{k-1}]$ is an array of integers, x and y are variables. The above element constraint is equivalent to :

$$y = a_x \wedge x \in I^k$$

i.e. it says that y is the x-th element of the array A. We will only consider injective element constraints, where the values appearing in the array A are pair wise distincts. In this case, the operational semantics of the element constraint is defined by the logical equivalence :

$$\forall i \in I^k,\ (x = i) \equiv (y = a_i)$$

For the sake of clarity, we extend the element constraint to sequences of variables. If $X = (v_i)_{i \in I^n}$ is a finite sequence of variables, then we define $A[X]$ as the application of an element constraint to each element of the sequence :

$$A[X] = (A[v_i])_{i \in I^n}$$

Element constraints can be used to describe applications of finite functions. For instance,

$$y = 3^x \wedge x \in I^4$$

is equivalently expressed through the following element constraint :

$$y = A[x] \wedge A = [1, 3, 9, 27]$$

Element constraint can also be used to represent the effect of value symmetries. Indeed, let θ be a value permutation corresponding to a value symmetry. By definition, any assignment of a value a to a variable x is transformed into the assignment of a^θ to x :

$$(x = a)^\theta = (x = a^\theta)$$

Let us consider x^θ. The permutation θ is represented by the array $A_\theta = [0^\theta, 1^\theta, \ldots, (k-1)^\theta]$. It defines a finite function that maps a to a^θ. The application of this function to x can be expressed by $A_\theta[x]$. Therefore, $x^\theta = A_\theta[x]$. We have represented the effect of the value symmetry by an element constraint.

More generally, if $(a_0, a_1, \ldots, a_{n-1})$ is the sequence of values taken by the variables $\mathcal{V} = (v_0, v_1, \ldots, v_{n-1})$, then $((a_0)^\theta, (a_1)^\theta, \ldots, (a_{n-1})^\theta)$ is the sequence of values taken by the variables $A_\theta[\mathcal{V}]$. Therefore, $S^\theta = A_\theta[S]$ for all solutions S.

Let us consider now the case where any symmetry is the composition $\sigma\theta$ of a variable permutation σ and a value permutation θ. The variable permutation σ is defined by a permutation of I^n. The value permutation is defined by a permutation of I^k.

If $(a_0, a_1, \ldots, a_{n-1})$ is the sequence of values taken by the variables $\mathcal{V} = (v_0, v_1, \ldots, v_{n-1})$, then $((a_{0^\sigma})^\theta, (a_{1^\sigma})^\theta, \ldots, (a_{(n-1)^\sigma})^\theta)$ is the sequence of values taken by the variables $A_\theta[\mathcal{V}^\sigma]$. Therefore, $S^{\sigma\theta} = A_\theta[S^\sigma]$ for all solutions S.

If S is a dynamic lex leader solution, then S must satisfy $S \preceq^{\mathrm{d}} S^{\sigma\theta}$, after (6). Therefore, S must satisfy :

$$S \preceq^{\mathrm{d}} A_\theta[S^\sigma] \tag{7}$$

5.2 A Forward Checking Algorithm

We want to filter the constraint (7) for a given variable symmetry σ and a value symmetry θ.

Let us look at a given node N in the search tree. It corresponds to a partial assignment $N = (v_{i_0} = a_0, v_{i_1} = a_1, \ldots, v_{i_{k-1}} = a_{k-1})$. Let us rename v_{i_j} into x_j. Then, the partial assignment is $N = (x_0 = a_0, x_1 = a_1, \ldots, x_{k-1} = a_{k-1})$. Let us consider a dynamic lex leader solution S that extends N. It is of the form $S = (x_0 = a_0, x_1 = a_1, \ldots, x_{n-1} = a_{n-1})$. The right hand side of (7) is $A_\theta[S^\sigma] = (x_{0^\sigma} = A_\theta[a_{0^\sigma}], x_{1^\sigma} = A_\theta[a_{1^\sigma}], \ldots, x_{(n-1)^\sigma} = A_\theta[a_{(n-1)^\sigma}])$. Then constraint (7) is equivalent to :

$$(a_0, a_1, \ldots, a_{n-1}) \preceq (A_\theta[a_{0^\sigma}], A_\theta[a_{1^\sigma}], \ldots, A_\theta[a_{(n-1)^\sigma}]) \tag{8}$$

Using (1), this constraint is equivalent to the conjunction of the constraints :

$$\left.\begin{array}{r} a_0 \leq A_\theta[a_{0^\sigma}] \\ a_0 = A_\theta[a_{0^\sigma}] \Rightarrow a_1 \leq A_\theta[a_{1^\sigma}] \\ \vdots \\ (a_0 = A_\theta[a_{0^\sigma}] \wedge \ldots \wedge a_{i-1} = A_\theta[a_{(i-1)^\sigma}]) \Rightarrow a_i \leq A_\theta[a_{i^\sigma}] \\ \vdots \\ (a_0 = A_\theta[a_{0^\sigma}] \wedge \ldots \wedge a_{n-1} = A_\theta[a_{(n-1)^\sigma}]) \Rightarrow a_n \leq A_\theta[a_{n^\sigma}] \end{array}\right\} \quad (9)$$

At the node N, we only know the values $(a_0, a_1, \ldots, a_{k-1})$. Indeed, the values a_i for $i > k$ correspond to variables not fixed at that node yet. Similarly, we only know the values a_{i^σ} when $i^\sigma < k$. Let K be the first i such that $i^\sigma \geq k$ or $i \geq k$:

$$(\forall i \in I^K, i^\sigma < k) \ \wedge \ K^\sigma \geq k \quad (10)$$

Then, we can use the first K constraints in (9). We simply have to check if each of the first K implications is true or not. If not, then we prune the current node.

The above is, in fact, a forward checking algorithm for the constraint (7).

5.3 A Stronger Filtering Algorithm

We can do better that forward checking if we know which variable x_k is selected for branching at node N, when N is not a leaf of the search tree. Then, we can use one extra constraint from (9). It is the constraint :

$$(a_0 = A_\theta[a_{0^\sigma}] \wedge \ldots \wedge a_{K-1} = A_\theta[a_{(K-1)^\sigma}]) \Rightarrow a_K \leq A_\theta[a_{K^\sigma}] \quad (11)$$

However, we do not know yet the value of a_K (if $K = k$) or the value of a_{K^σ} (if $K^\sigma \geq k$). Therefore, we need to state a conditional constraint on the variables x_K and x_{K^σ}

$$(a_0 = A_\theta[a_{0^\sigma}] \wedge \ldots \wedge a_{K-1} = A_\theta[a_{(K-1)^\sigma}]) \Rightarrow x_K \leq A_\theta[x_{K^\sigma}] \quad (12)$$

Adding (12) results in more filtering than the forward checking algorithm proposed above. Note that, in fact, we enforce the following $K + 1$ conditional constraints :

$$\forall j \in I^{K+1},\ (a_0 = A_\theta[a_{0^\sigma}] \wedge \ldots \wedge a_{j-1} = A_\theta[a_{(j-1)^\sigma}]) \Rightarrow x_j \leq A_\theta[x_{j^\sigma}] \quad (13)$$

Enforcing these constraints can be done in a time linear in the number of variables. Indeed, it is sufficient to scan both sequences $(x_0, x_1, \ldots, x_k)$ and $(A_\theta[x_{0^\sigma}], A_\theta[x_{1^\sigma}], \ldots, A_\theta[x_{k^\sigma}])$ in increasing order as long as corresponding entries are fixed to equal values. It is also easy to get an incremental version of the algorithm.

Let us look at the tree search given in Fig. 2 for the CSP (3). This CSP has two variable symmetries : the identity, and the permutation σ that swaps v_1 and v_2.

Let us consider the node corresponding to $v_0 = 0$. The variable selected at this node is v_2. Then, (13) is :

$$\begin{aligned} v_0 &\leq A_{id}[v_{0^\sigma}] \\ 0 = A_{id}[0^\sigma] \Rightarrow v_2 &\leq A_{id}[v_{2^\sigma}] \end{aligned}$$

Since the value symmetry is the identity id, these constraints can be simplified into :

$$\begin{aligned} v_0 &\leq v_{0^\sigma} \\ 0 = 0^\sigma \Rightarrow v_2 &\leq v_{2^\sigma} \end{aligned}$$

Since σ is the permutation $[0, 2, 1, 3]$, it can be simplified into :

$$\begin{aligned} v_0 &\leq v_0 \\ 0 = 0 \Rightarrow v_2 &\leq v_1 \end{aligned}$$

i.e.

$$v_2 \leq v_1$$

This constraint would prune immediately the node $(v_0 = 0, v_2 = 2)$. Indeed, v_1 must be at least 2, which implies $v_1 = 2$. This is inconsistent with the AllDifferent constraint.

More generally, stating our dynamic lex leader constraint would prune the tree search into the one shown in Fig. 2. There are only 3 solutions left, the dynamic lex leader ones.

More generally, stating our dynamic lex leader constraint would prune the tree search into the one shown in Fig. 3. There are only 3 solutions left, the dynamic lex leader ones.

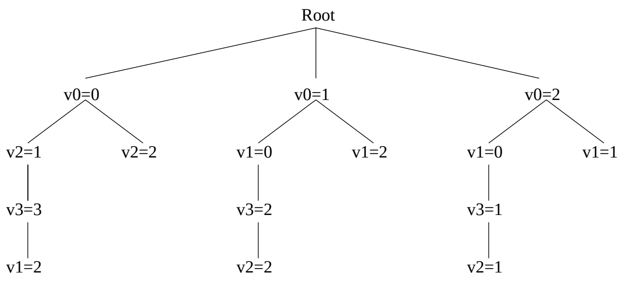


Fig. 2. A tree search with dynamic lex leader constraints

This method requires $L \times M$ constraints when there are L variables symmetries and M value symmetries. This method will not scale well with the number of value symmetries. The next section describes a way to cope with a large number of value symmetries.

5.4 A Global Constraint

We can filter all the above constraints for a given σ, regardless of the number of value symmetries θ, as follows. We reuse previous notations. Given a node $N = (x_0 = a_0, x_1 = a_1, \dots, x_{k-1} = a_{k-1})$, K such as (10), then we want to enforce the constraints (13) for a given variable symmetry σ :

$$\forall j \in I^{K+1},\ \forall\theta,\ (a_0 = A_\theta[a_{0^\sigma}] \wedge \dots \wedge a_{j-1} = A_\theta[a_{(j-1)^\sigma}]) \Rightarrow x_j \leq A_\theta[x_{j^\sigma}] \quad (14)$$

For a given j, if the left hand side is not true, then, nothing can be done. If the left hand side is true, then, θ is such that :

$$\forall i \in I^j,\ a_i = A_\theta[(a_{i^\sigma})] \quad (15)$$

Let G^σ_Σ be the set of value symmetries that satisfies (15). Then, for any of these θ, we have to enforce the right hand side :

$$\forall\theta \in G^\sigma_\Sigma, v_j \leq A_\theta[v_{j^\sigma}] \quad (16)$$

Let a_j be the minimum value in the domain of v_j. Let b be a value in the domain of v_{j^σ} in state Σ. If there exists $\theta \in G^\sigma_\Sigma$ such that $a_j > A_\theta[b]$, then b should be removed from the domain of v_{j^σ}.

Therefore, in order to enforce (16), it is necessary to remove all the values b from the domain of v_{j^σ} such that $a_j > A_\theta[b]$:

$$\forall b\ \exists\theta \in G^\sigma_\Sigma,\ a_j > A_\theta[b] \rightarrow v_{j^\sigma} \neq b \quad (17)$$

Second, let b_j be the maximum value of the expression $A_\theta[v_{j^\sigma}]$ for $\theta \in G^\sigma_\Sigma$. Then $v_j \leq b_j$, i.e :

$$v_j \leq max_{(\theta\ \in\ G^\sigma_\Sigma)}(A_\theta[v_{j^\sigma}]) \quad (18)$$

In order to implement our method, one needs to compute the value symmetries that satisfy (15) efficiently. It can be done in polynomial using computational group theory algorithms (see [18] for instance).

A simple implementation is possible when any value permutation is a value symmetry. In this case, it is easy to compute G^σ_Σ from (15). Indeed, (15) is of the form :

$$A_\theta[b_0] = c_0, \dots, A_\theta[b_{j-1}] = c_{j-1} \quad (19)$$

Let $\mathcal{C}$ be the set of the c_i that appear in (19), and let $\mathcal{B}$ be the set of the b_i that appears in (19). Then, the set of value symmetries θ that are consistent with (19) are :

$$\begin{array}{c} \forall i \in I^j, A_\theta[b_i] = c_i \\ \forall b \in I^n - \mathcal{B}, A_\theta[b] \in I^n - \mathcal{C} \end{array} \quad (20)$$

Then, (17) becomes :

$$\begin{array}{rr} \forall i \in I^j, & a_j > c_i \rightarrow v_{j^\sigma} \neq b_i \\ \forall b \in I^n - \mathcal{B}, & a_j > min(I^n - \mathcal{C}) \rightarrow v_{j^\sigma} \neq b \end{array} \quad (21)$$

Similarly, the computation of right hand side of (18) becomes straightforward.

It is worth looking at the case where σ is the identity. In this case, the above reasoning can be simplified. First of all, G^{id}_{Σ} is now the set of value symmetries θ such that :

$$\forall i \in I^j,\ a_i = A_\theta[a_i]$$

It is called the point wise stabilizer of $(a_0, a_1, \ldots, a_{k-1})$. This set is denoted $G_{(a_0,a_1,\ldots,a_{k-1})}$. Then, condition (17) becomes simpler. We only have to remove from the domain of v_j all the values b such that there exists θ in $G_{(a_0,a_1,\ldots,a_{j-1})}$ such that $b > b^\theta$. It is exactly the definition of the GE-tree method of [17].

The above algorithm is similar to the global constraint described in [16]. the main difference is that we use a dynamic variable ordering here, whereas [16] uses a static ordering.

6 Experimental Results

We have implemented the filtering algorithm described in section 5.3. We have compared it to both lex constraints and SBDS. All experiments were run with ILOG Solver 6.3 [9] on a 1.4 GHz Dell D800 laptop running Windows XP.

The first experiment uses graceful graph coloring. A graph with m edges is *graceful* if there exists a labeling f of its vertices such that :

- $0 \leq f(i) \leq m$ for each vertex i,
- the set of values $f(i)$ are all different,
- the set of values $abs(f(i) - f(j))$ for every edge (i, j) are all different. They are a permutation of $(1, 2, \ldots, m)$.

A straightforward translation into a CSP exists where there is a variable x_i for each vertex i. These are hard CSPs introduced in [10]. They have been used as test bed for symmetry breaking methods, see for instance [11][19][16]. A more efficient CSP model for graceful graphs has recently been introduced in [20]. In this model, any symmetry of the graph induces a value symmetry. Together with this model, a clever search strategy is proposed. It is shown in [20] that this search strategy clashes with lex constraints when they are used for breaking symmetries. For this reason, symmetries are broken using SBDS in [20].

We present in Table 1. and Table 2. results for various graceful graph problems. We give the number of symmetries for each graph. We compare 4 methods : no symmetry breaking, static lex constraints, SBDS, and dynamic lex constraints. We use the implementation of [8] for SBDS. Since all experiments use the same version of ILOG Solver and since they are run on the same computer we believe the comparison is fair. For each method we report the number of solutions found, the number of backtracks, and the time needed to solve the problem. Table 1. present results for finding all solutions (or prove there are none when the problem is unsatisfiable). Table 2. presents results for finding one solution when the problem is satifiable.

Table 1. Results for computing all solutions for graceful graphs

Graph	SYM	No sym break			static lex			SBDS			dynamic lex		
		SOL	BT	sec.	SOL	BT	sec.	SOL	BT	sec.	SOL	BT	sec.
K3×P2	24	96	336	0.30	4	20	0.02	4	16	0.02	4	14	0.01
K4×P2	96	1440	14640	18.7	15	260	0.43	15	166	0.3	15	151	0.27
K5×P2	480	480	337360	2267	1	1789	10.3	1	828	5.51	1	725	5.18
K6×P2	2880				0	6751	121.6	0	1839	43.2	0	1559	40.5
K7×P2	20160				0	18950	786	0	2437	149.4	0	1986	139.4

Table 2. Results for computing one solution for graceful graphs

Graph	SYM	No sym break		static lex		SBDS		dynamic lex	
		BT	sec.	BT	sec.	BT	sec.	BT	sec.
K3×P2	24	0	0	6	0.01	5	0.01	5	0.01
K4×P2	96	16	0.08	14	0.06	12	0.05	12	0.05
K5×P2	480	2941	19.1	557	3.19	428	2.79	392	2.77

These results show a significant increase of efficiency from no symmetry breaking to static lex constraints, SBDS, and dynamic lex constraints. They also show that all three symmetry breaking methods improve the search for one solution.

We have performed a second set of experiments to test our claim, which is that dynamic lex constraints are less sensitive to the order used in search than static lex constraints. In order to test this, we compared the performance of static lex constraints and dynamic lex constraints with various variable order for search :

- SAME : A static order corresponding to the order used in the static lex constraints.
- FF : A fail first principle (selecting the variable with the smallest domain) where ties are broken with the order used in the static lex constraints.
- INV : A static order corresponding to the inverse of the order used in the static lex constraints.
- INVFF : A fail first principle where ties are broken with the inverse of the order used in the static lex constraints.

We expect a good performance of static lex constraint for SAME and FF orders, and we expect a bad performance for INV and INVFF. We also expect dynamic lex constraints to be much better for INV and INVFF.

We use a simple CSP taken from [7] :

$$A^3 + B^3 + C^3 + D^3 = E^3 + G^3 + H^3$$

There are 144 symmetries corresponding to the 24 permutations of A, B, C, D times the 6 permutations of E, F, G.

Results of the experiments are given in Table 3. and Table 4. The last two columns give the ratio of static lex constraints over synmaic lex constraints for

Table 3. Results for finding all solutions of $A^3 + B^3 + C^3 + D^3 = E^3 + G^3 + H^3$

Order	SOL	static lex		dynamic lex		static/dynamic	
		BT	time	BT	time	BT	time
SAME	265	208,264	4.90	208,264	6.32	1	0.8
FF	265	208,264	5.02	208,264	6.39	1	0.8
INV	265	1,240,819	63.0	212,930	7.14	6.3	8.8
INVFF	265	1,031,711	58.5	212,930	7.25	4.8	8.1

Table 4. Results for finding one solution of $A^3 + B^3 + C^3 + D^3 = E^3 + G^3 + H^3$

Order	SOL	static lex		dynamic lex		static/dynamic	
		BT	time	BT	time	BT	time
SAME	1	989	0.08	989	0.08	1	1
FF	1	989	0.08	989	0.08	1	1
INV	1	414,886	19.6	1,318	0.11	315	178
INVFF	1	301,865	17.5	1,318	0.12	229	146

the number of backtracks a and the number of fails . A value above 1 means that the performance is better for dynamic lex constraints.

These results show that :

– The performance of dynamic lex constraints does not depend much on the search order, whereas they vary enormously for static lex constraints;
– the performance of dynamic lex constraint is in the same ballpark than the performance of static lex constraints when the search order is good for static lex constraints (SAME and FF);
– the performance of dynamic lex constraints is much better when the order is not good for static lex constraint (INV and INVFF);
– the difference between static and dynamic lex constraints is much more important when we search for one solution than when we search for all solutions.

7 Conclusions and Future Research

We have provided a new definition for symmetry breaking in term of dynamic lex leader solutions. These solutions are similar to the lex leader solutions of [3], except that the ordering used is the variable ordering used during search. We have shown that the first solution in any tree search is a dynamic lex leader solution. It means that filtering out solutions which are not dynamic lex leader solutions will not slow down search. We have described simple, yet efficient, algorithms for this filtering. These algorithms are similar to the ones presented in [16], except that we use a dynamic ordering for variables here. Preliminary experimental results show that dynamic lex constraints filtering can be stronger and more efficient than SBDS. They also show that dynamic lex constraints can be as efficient as static lex constraints when the variable order used in the search is the same as the one used in lex constraints. Last, they show that dynamic lex constraints can be much more efficient than static lex constraints when variable orders clash.

Even if these result are encouraging, there is much room for improvement. Indeed, the algorithm described in section 5.3 requires to state one constraint per symmetry of the problem. It is not tractable for problems where the number of symmetries is very large. One possible way to solve this issue is to implement the global constraint described in section 5.4. We are confident it can be done, since we have developed a similar global constraints when the order of variables is fixed [16]. In such case, the number of symmetry breaking constraint no longer depends on the number of value symmetries. It remains to be seen if we can state less constraints than the number of variable symmetries. It would be interesting to see if we can combine our work for instance with the one of [13].

Acknowledgements

We thank Barbara Smith for providing both the SBDS implementation of [8] and her code for the graceful graph problem. We also thank Marie Puget for her support and her proof reading.

References

1. Backofen, R., Will, S.: “Excluding Symmetries in Constraint Based Search” Proceedings of CP’99 (1999).
2. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K., Smith, B.: “Symmetry Definitions for Constraint Satisfaction Problems” In *proceedings of CP 2005*, ed. Peter van Beek, pp. 17-31, Springer, LNCS 3709, 2005.
3. Crawford, J., Ginsberg, M., Luks E.M., Roy, A.: “Symmetry Breaking Predicates for Search Problems”. In *proceedings of KR’96*, pp. 148-159.
4. Fahle, T., Shamberger, S., Sellmann, M.: “Symmetry Breaking” In *proceedings of CP01* (2001) pp. 93-107.
5. Focacci, F., and Milano, M.: “Global Cut Framework for Removing Symmetries” In *proceedings of CP’01* (2001) PP. 75-92.
6. Gent, I.P., and Harvey, W., and Kelsey, T.: “Groups and Constraints: Symmetry Breaking During Search” *In proceedings of CP 2002*, pp. 415-430.
7. Gent, I.P., Harvey, W., Kelsey, T., Linton, S.: “Generic SBDD Using Computational Group Theory” *In proceedings of CP 2003*, pp. 333-437.
8. Gent, I.P., and Smith, B.M.: “Symmetry Breaking During Search in Constraint Programming” In *proceedings of ECAI’2000*, pp. 599-603.
9. ILOG: *ILOG Solver 6.3. User Manual* ILOG, S.A., Gentilly, France, July 2006.
10. I. J. Lustig and J.-F. Puget. “Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming”. INTERFACES, 31(6):29–53, 2001.
11. Petrie, K., Smith, B.M. 2003. “Symmetry breaking in graceful graphs.” In *proceedings of CP’03*, LNCS 2833, pp. 930-934, Springer Verlag, 2003.
12. Puget, J.-F.: “On the Satisfiability of Symmetrical Constraint Satisfaction Problems.” In *proceedings of ISMIS’93* (1993), pp.350–361.

13. Puget, J.-F.: "Breaking symmetries in all different problems". In *proceedings of IJCAI 05*, pp. 272-277, 2005.
14. Puget, J.-F.: "Symmetry Breaking Revisited". Constraints 10(1): 23-46 (2005)
15. Puget J.-F.: "Breaking All Value Symmetries in Surjection Problems" In *proceedings of CP 05*, pp. 490-504, 2005.
16. Puget J.-F.: "An Efficient Way of Breaking Value Symmetries" To appear in *proceedings of AAAI 06*.
17. Roney-Dougal C.M., Gent, I.P., Kelsey T., Linton S.: "Tractable symmetry breaking using restricted search trees" In *proceedings of ECAI'04*.
18. Seress, A. 2003. *Permutation Group Algorithms* Cambrige University Press, 2003.
19. Smith, B.: "Sets of Symmetry Breaking Constraints" In *proceedings of SymCon05*, the 5th International Workshop on Symmetry in Constraints, 2005.
20. Smith, B.: "Constraint Programming Models for Graceful Graphs", To appear in *Proceedings of CP 06*

Generalizing AllDifferent: The SomeDifferent Constraint

Yossi Richter, Ari Freund, and Yehuda Naveh

IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
{richter, arief, naveh}@il.ibm.com

Abstract. We introduce the SomeDifferent constraint as a generalization of AllDifferent. SomeDifferent requires that values assigned to *some* pairs of variables will be different. It has many practical applications. For example, in workforce management, it may enforce the requirement that the same worker is not assigned to two jobs which are overlapping in time. Propagation of the constraint for hyper-arc consistency is NP hard. We present a propagation algorithm with worst case time complexity $O(n^3\beta^n)$ where n is the number of variables and $\beta \approx 3.5$ (ignoring a trivial dependence on the representation of the domains). We also elaborate on several heuristics which greatly reduce the algorithm's running time in practice. We provide experimental results, obtained on a real-world workforce management problem and on synthetic data, which demonstrate the feasibility of our approach.

1 Introduction

In this paper we consider a generalization of the well known AllDifferent constraint. The AllDifferent constraint requires that the variables in its scope be assigned different values. It is a fundamental primitive in constraint programming (CP), naturally modeling many classical constraint satisfaction problems (CSP), such as the n-queen problem, air traffic management [2,9], rostering problems [18], and many more. Although a single AllDifferent constraint on n variables is semantically equivalent to $n(n-1)/2$ binary NotEqual constraints, in the context of maintain-arc-consistency algorithms it is vastly more powerful in pruning the search space and reducing the number of backtracks. It also simplifies and compacts the modeling of complex problems. It has thus attracted a great deal of attention in the CP literature (see [19] for a survey).

Despite its usefulness, the AllDifferent constraint is too restrictive in many applications. Often we only desire that certain pairs of variables assume different values, and do not care about other pairs. A simple example of this is the following workforce management problem [21]. We are given a set of jobs, a set of workers, and a list specifying which jobs can be done by which workers. In addition, the jobs may be specified to start and end at different times, or require only partial availability of a worker. Consequently, some pairs of jobs may be assigned to the same worker while others may not. A simple way to model the problem is to let each job correspond to a variable, and let the domain of each variable be the set of workers qualified to do the job. Then one can add a binary NotEqual constraint for every two jobs which overlap in their time of execution and require together more than 100% availability. Additional constraints can then be added according to the detailed specification of the problem. While semantically correct, this model suffers from the same disadvantages of modeling an AllDifferent problem by multiple NotEqual constraints. On the other hand, the AllDifferent constraint is inappropriate here because in general, the same worker can be assigned to two jobs.

While workforce management is a CP application of prime, and growing, importance (especially in light of the current trends towards globalization and strategic outsourcing), it is not the only problem in which the above situation arises. Some further examples include: circuit design, in which any two macros on a chip may or may not overlap, depending on their internal structure; university exam scheduling, in which the time-slot of any two exams may or may not be the same, depending on the number of students taking both exams; and computer-farm job scheduling, in which, depending on the types of any two jobs, the same machine may or may not process them simultaneously.

All of this calls for a generalization of AllDifferent in which some, but not all, pairs of variables require distinct values. We call this generalization SomeDifferent. Formally, the SomeDifferent constraint is defined over a set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D = \{D_1, \ldots, D_n\}$, and an underlying graph $G = (X, E)$. The tuples allowed by the constraint are: $\texttt{SomeDifferent}(X, D, G) = \{(a_1, \ldots, a_n) : a_i \in D_i \ \wedge \ a_i \neq a_j \text{ for all } (i, j) \in E(G)\}$. The special case in which G is a clique is the familiar AllDifferent constraint. Another special case that has received some attention is the case of two AllDifferent constraints sharing some of their variables [1].

Our focus in this paper is on hyper-arc consistency propagation for the SomeDifferent constraint. Since most CSP algorithms use arc-consistency propagation as a subroutine (see, e.g., [7]), it is important to develop specialized propagation algorithms for specific constraint types—algorithms that are able to exploit the concrete structure of these constraints. For example, the AllDifferent constraint admits a polynomial-time hyper-arc consistency propagation algorithm based on its bipartite graph structure [14]. In contrast, and despite the similarity between SomeDifferent and AllDifferent, there is little hope for such an algorithm for the SomeDifferent constraint, as the hyper-arc consistency propagation problem for SomeDifferent contains the NP hard problem of graph 3-colorability as a special case.

The NP hardness of the propagation problem accommodates two approaches. One is to aim for relaxed or approximated propagation. This approach has been taken previously in the context of other NP Hard propagation problems [15,16,17]. The second approach is to tackle the problem heuristically. In this paper we combine a theoretically grounded exact algorithm (with exponential worst case running time) with several heuristics that greatly speed it up in practice.

Our results. We introduce an exact propagation algorithm for hyper-arc consistency of the `SomeDifferent` constraint. The algorithm has time complexity of $O(n^3\beta^n)$, with $\beta \approx 3.5$, and depends on the domain sizes only for the unavoidable deletion operations. We have implemented the algorithm (with multiple additional heuristics) and tested it on two kinds of data:

– IBM's workforce management instances.
– Synthetic data generated through a random graph model.

In both cases the implementation performed well, much better than expected from the theoretical bounds. It also compared favorably (though not in all cases) with the approach of modeling the problem by `NotEqual` constraints.

Organization of this paper. The remainder of the paper is organized as follows. In Section 2 we describe a graph theoretical approach to the problem of propagating the `SomeDifferent` constraint for hyper-arc consistency. In Section 3 we present our algorithm and its worst-case analysis. In Section 4 we discuss a few practical heuristic improvements. In Section 5 we report experimental results. In Section 6 we conclude and discuss future work.

2 A Graph-Theoretical Approach

The problem of propagation of `SomeDifferent` for hyper-arc consistency can be formulated as the following graph coloring problem. The input is a graph $G = (V, E)$, where each vertex u is endowed with a finite set of *colors* D_u. The vertices correspond to variables; the sets of colors correspond to the domains of the respective variables; the graph edges correspond to pairs of variables that may not be assigned the same value, as mandated by the constraint. A valid *coloring* of the graph is an assignment $c : V \rightarrow \bigcup_{u\in V} D_u$ of colors to vertices such that: (1) each vertex u is assigned a color $c(u) \in D_u$; and, (2) no two adjacent vertices are assigned the same color. If a valid coloring exists, we say the graph is *colorable.*[1] Valid colorings correspond to assignments respecting the `SomeDifferent` constraint. We view a coloring of the graph as a collection of individual *point colorings*, where by *point coloring* we mean a coloring of a single vertex by one of the colors. We denote the point coloring of vertex u by color c by the ordered pair (u, c). We say that (u, c) is *extensible* if it can be *extended* into a valid coloring of the entire graph (i.e., if there exists a valid coloring of the entire graph in which u is colored by c). The problem corresponding to propagation of `SomeDifferent` for hyper-arc consistency is: given the graph and color sets, prune the color sets such that: (1) every extensible point coloring in the original graph remains in the pruned graph; and (2) every point coloring in the pruned graph is extensible. (We view the color sets as part of the graph. Thus we speak of the "pruned graph" despite the fact that it is actually the color sets that are pruned, not the graph itself.) This problem is NP hard, as even the problem

[1] We emphasize that we do not refer here to the usual terminology of graph coloring, but rather to coloring from domains.

of determining whether the graph is colorable contains the NP hard problem of 3-colorability as the special case in which all color sets are identical and contain three colors.

The solution to the above pruning problem is of course unique. It consists of pruning all point colorings that are non-extensible with respect to the original graph. Thus the problem reduces to identifying the non-extensible point colorings and eliminating them. The direct approach to this is to test each point coloring (u, c) for extensibility by applying it (i.e., conceptually coloring u by c, deleting c from the color sets of the neighbors of u, and deleting u from the graph) and testing whether the remaining graph is colorable. Thus the number of colorability testings is $\sum_u |D_u|$, and the time complexity of each is generally exponential in the total number of colors available. This can be prohibitively costly when the color sets are large, even if the number of vertices is small. Fortunately, it is possible to do better in such cases. Let us denote $D(U) = \bigcup_{u \in U} D_u$ for any $U \subseteq V$. Following the terminology of [13], we make the following definition.

Definition 1. *We say that a set of nodes U is a* failure set *if $|D(U)| < |U|$, in which case we also say that the subgraph induced by U is a* failure subgraph*. Note that the empty set is not a failure set.*

The key observation on which our algorithm is based is contained in Lemma 2 below, whose proof requires the next lemma.

Lemma 1. *If the graph contains no failure sets, then it is colorable.*

Proof. This is a straightforward application of Hall's Theorem (see, e.g., Reference [20], Chapter 3.1):

> Let $H = (L, R, F)$ be a bipartite graph, and for every subset $U \subseteq L$, let $N(U) = \{v \in R \,|\, \exists u \in U, uv \in F\}$. If $|N(U)| \geq |U|$ for all $U \subseteq L$, then F admits a matching that *saturates* L, i.e., a matching in which every vertex in L is matched.

We apply Hall's theorem by defining the following bipartite graph $H = (L, R, F)$: $L = V$; $R = D(V)$; and $F = \{uc \,|\, u \in L \wedge c \in D_u\}$. The condition that no subset of V is a failure set translates into the condition of Hall's Theorem for H, and therefore a matching saturating L exists. This matching defines a coloring of G in which each vertex is assigned a color from its corresponding color set, and no color is shared by two vertices. Such a coloring is necessarily valid. □

Lemma 2. *The graph is colorable if and only if each of its failure subgraphs is colorable.*

Proof. Clearly, if the graph is colorable, then so is every failure subgraph. Conversely, if the graph is not colorable, then by Lemma 1 it contains a failure subset. Let $U \subseteq V$ be a maximal failure set, i.e., U is a failure set and is not a proper subset of any other failure set. If $U = V$ we are done. Otherwise, let $D_1 = D(U)$ and $D_2 = D(V) \setminus D_1 = D(V \setminus U) \setminus D_1$. Consider the subgraph induced by $V \setminus U$ with its color sets restricted to the colors in D_2 (i.e., with each color set D_v ($v \in V \setminus U$) replaced by $D_v \setminus D_1$). Neither this graph, nor any of its induced subgraphs may be failure subgraphs, for had any subset of $V \setminus U$ been a failure set with respect to D_2, then so would its union with U be (with respect to $D_1 \cup D_2$, i.e., when the original color sets for the vertices in $V \setminus U$ are reinstated), contradicting the maximality of U. Thus, by Lemma 1, the subgraph induced by $V \setminus U$ is colorable using only colors from D_2. It follows that the subgraph induced by U is not colorable. (Otherwise the entire graph would be colorable, since the coloring of U would only use colors from D_1 and so could coexist with the coloring of $V \setminus U$ that uses only colors from D_2). Thus, the non-colorability of the graph implies the existence of a failure subgraph that is non-colorable, which completes the proof. □

3 The Algorithm

Testing whether a point coloring (u, c) is extensible can be carried out by removing c from the color sets of u's neighbors and deleting u (thus effectively coloring u by c), and checking whether the resulting graph is colorable. Lemma 2 implies that it is sufficient to check whether some induced subgraph of the resulting graph is colorable. This seems to buy us very little, if anything, since we now replace the checking of a single graph with the checking of many (albeit smaller) subgraphs. (And, of course, we must still do this for each of the $\sum_u |D_u|$ point colorings.) However, we can realize significant savings by turning the tables and enumerating subsets of vertices rather than point assignments. More specifically, we enumerate the non-empty proper subsets of V, and test point colorings only for those that are failure sets. As we shall see shortly, this reduces the number of colorability checkings to at most $n^2 2^n$, where n is the number of vertices.

Postponing to later the discussion of how to check whether a subgraph is colorable, we now present the pruning algorithm.

```
1. For each $\emptyset \subsetneq U \subsetneq V$:
2.     If $|D(U)| \leq |U|$:
3.         For each $c \in D(U)$ and $v \in V \setminus U$ such that $c \in D_v$:
4.             Check whether the subgraph induced by $U$ is
               colorable with $c$ removed from the color sets of the
               neighbors of $v$. If not, report $(v, c)$ as non-extensible.
5. Prune the point colorings reported as non-extensible.
6. Check whether the pruned graph contains a vertex with an empty
   color set. If so, announce that the graph is not colorable.
```

Note that as the algorithm stands, the same point coloring may be reported multiple times. We address this issue in Section 4.

3.1 Correctness of the Algorithm

We now argue that the algorithm correctly identifies the non-extensible point coloring when the graph is colorable, and announces that the graph is not colorable when it is not.

Proposition 1. *If the graph is colorable, the algorithm reports all non-extensible point colorings, and only them.*

Proof. A point coloring is reported by the algorithm as non-extensible only if applying it renders some subgraph (and hence the entire graph) non-colorable. Thus all point colorings reported by the algorithm are indeed non-extensible. Conversely, to see that all non-extensible point colorings are reported, consider any such point coloring (v, c). Coloring v by c effectively removes c from the color sets of v's neighbors, and the resulting graph (in which v is also removed) is non-colorable (since (v, c) is non-extensible). By Lemma 2, this graph contains a subgraph induced by some failure set U that is not colorable. But originally this subgraph was colorable, since the original graph is colorable. This is only possible if $c \in D(U)$. Additionally, because U is a failure set with c removed, it must be the case that $|D(U)| \leq |U|$ before c is removed. Thus, when the algorithm considers U it will report (v, c) as non-extensible. □

Proposition 2. *If the graph is non-colorable, the algorithm detects this.*

Proof. If the graph is non-colorable, then by Lemma 2 it contains a non-colorable subgraph induced by some failure set U. If U is a singleton $U = \{v\}$, then $D_v = \emptyset$, and we are done. Otherwise, U must contain at least one vertex v such that $D_v \subseteq D(U \setminus \{v\})$, for otherwise U would not be a failure set (and furthermore, the subgraph induced by it would be colorable). Thus, $|D(U \setminus \{v\}\,| = |D(U)| \leq |U \setminus \{v\}\,|$, so when the algorithm considers $U \setminus \{v\}$ it will enter the inner loop and report all the point colorings involving v as non-extensible. □

3.2 Checking Whether a Subgraph is Colorable

We are left with the problem of testing whether a given subgraph is colorable. We do this by reducing the colorability problem to a *chromatic number* computation. The *chromatic number* of a given graph G, denoted $\chi(G)$, is the minimum number of colors required for coloring G's vertices such that no two neighbors receive the same color. In the terms of our coloring framework, we are considering here the special case in which all color sets are identical. The chromatic number of the graph is the minimum size of the color set for which a valid coloring still exists. Computing the chromatic number of a graph is known to be NP hard.

We use the following construction. To test the colorability of the subgraph induced by U, extend this subgraph as follows. Let $r = |D(U)|$. Add r new vertices, each corresponding to one of the colors in $D(U)$, and connect every pair of these by an edge (i.e., form a clique). Then add an edge between each vertex $v \in U$ and each new vertex that corresponds to a color that is *not* in D_v. Let G' be the resulting graph.

Claim. The subgraph induced by U is colorable if and only if $\chi(G') = r$.

Proof. If the subgraph induced by U is colorable, then we already have a coloring of the vertices in U by at most r colors. We can extend this coloring to

G' by coloring each of the new vertices by the color (from $D(U)$) to which it corresponds. It is easy to see that in so doing, no two neighbors in G' receive the same color. Thus $\chi(G') \leq r$ and as G' contains a clique of size r, $\chi(G') = r$ Conversely, if $\chi(G') = r$, then there exists a coloring of the vertices in G', using r colors, such that no two neighbors receive the same color. Note that the new vertices all have different colors since they are all adjacent to each other. Without loss of generality, assume that the colors are $D(U)$ and that each of the new vertices is given the color it corresponds to. (Otherwise, simply rename the colors.) Now consider a vertex $v \in U$. For each of the colors $c \in D(U) \setminus D_v$ the new vertex corresponding to c is adjacent to v and is colored by c. Thus v cannot be colored by any of these colors, and is therefore colored by a color in its color set D_v. Thus the restriction of the coloring to the subgraph induced by U is valid. □

Therefore, to test whether the subgraph induced by U is colorable we construct the extended graph G' and compute its chromatic number. Computing the chromatic number of a graph is a well researched problem, and a host of algorithms were developed for it and related problems (e.g., [3,4,5,6,8,10,11]). Some of these algorithms (e.g., DSATUR [5]) are heuristic—searching for a good, though not necessarily optimal, coloring of the vertices. By appropriately modifying them and properly setting their parameters (e.g., backtracking versions of DSATUR) they can be forced to find an optimal coloring, and hence also the chromatic number of the graph. In contrast, there are other algorithms, so called *exact* algorithms, that are designed specifically to find an optimal coloring (and the chromatic number) while attempting to minimize the worst case running time (e.g., [4,6,8,11]). These algorithms are based on enumerating (maximal) independent sets of vertices. Their time complexity is $O(a^k)$, where a is some constant (usually between 2 and 3) and k is the number of vertices in the graph. The algorithms differ from one another by the various divide-and-conquer techniques they employ to reduce the running time or the memory consumption. The worst case running times of the heuristic algorithms are of course exponential too, but with higher exponent bases.

An important observation regarding the complexity of computing the chromatic number of G' is that the number of vertices it contains is $|U| + |D(U)| \leq 2|U|$, since we only perform the computation for sets U such that $|D(U)| \leq |U|$. Thus for an $O(\alpha^k)$ algorithm, the actual time complexity bound is $O\big((\alpha^2)^{|U|}\big)$.

While little can be said analytically with respect to the heuristic algorithms, the exact algorithms are open to improvement due to their reliance on independent sets and the fact that the new vertices in G' form a clique. This can be demonstrated, e.g., on Lawler's algorithm [11], whose running time (in conjunction with Eppstein's maximal independent set enumeration algorithm [8]) is $O(2.443^k)$. The proof of the following proposition is omitted due to lack of space.

Proposition 3. *A suitably modified version of Lawler's algorithm runs on G' in $O(|U| \cdot 2.443^{|U|})$, rather than $O\big((2.443^2)^{|U|}\big)$.*

3.3 Complexity Analysis

The time complexity of the algorithm is determined by the loops. The outer loop iterates $2^n - 2$ times (n is the number of vertices). For each subset U with $|D(U)| \leq |U|$, the inner loop is iterated at most $|D(U)| \cdot |V \setminus U| \leq |U| \cdot |V \setminus U| \leq n^2/4$ times. Thus the total number of chromatic-number computations is at most $\frac{1}{4}n^2 2^n$. Letting $\mathrm{O}(n\alpha^n)$ denote the time complexity of the chromatic-number algorithm used, we get a time bound of $\mathrm{O}(n^3 2^n \alpha^n)$ on the work involved in chromatic-number computations. This bound can be tightened to $\mathrm{O}(n^3(\alpha+1)^n)$ by observing that the chromatic-number algorithm is called at most $n^2\binom{n}{k}$ times with a subgraph of size k, and so the total time is bounded by $\mathrm{O}\big(n^2 \sum_{k=1}^{n} \binom{n}{k} k\alpha^k\big) = \mathrm{O}(n^3(\alpha+1)^n)$.

In addition to the chromatic number computations, there is some work involved in manipulating color sets—specifically, evaluating $|D(U)| \leq |U|$ for each subset U, determining which vertices have colors from $D(U)$ in their color sets, and deleting colors from color sets. The time complexity of these operations will depend on the way the color sets are represented (e.g., by explicit lists, number ranges, bit masks, etc.), but, under reasonable assumptions, will not exceed $\mathrm{O}(n^3)$ time in the worst case (and in practice will be much better) for each iteration of the outer loop. There is an additional, unavoidable, dependence on the domain sizes of vertices not in U, which is incurred when the algorithm considers and deletes point colorings. This dependence is logarithmic at worst, using a straightforward implementation. Such unavoidable dependences notwithstanding, the worst case total running time of the algorithm is $\mathrm{O}(n^3 2^n + n^3(\alpha+1)^n) = \mathrm{O}(n^3(\alpha+1)^n)$.

4 Practical Improvements

There are a number of heuristics which when added to the algorithm greatly enhance its practicality. We list some of them next.

Connected components and superfluous edges. The two sources of exponentiality are the enumeration of subsets and the chromatic number computations (which are exponential in the number of vertices). It is fairly obvious that if the graph is not connected, the algorithm can be run on each connected component separately without affecting its correctness. Also, if an edge has two endpoints with disjoint color sets, then it can be dropped without affecting the semantics of the graph. We can therefore delete any such edges, decompose the resultant graph into connected components and run the algorithm on each separately. Doing so yields a speedup that is (potentially) exponential in the difference between the graph size and the connected component size (e.g., if the graph contains 200 nodes and consists of twenty components of size 10 each, then the potential speedup is $\beta^{200}/(20 \cdot \beta^{10}) = \frac{1}{20} \cdot \beta^{190}$).

A refinement of this heuristic is based on the observation that a failure subgraph is non-colorable if and only if it contains a non-colorable connected failure subgraph of itself. Thus it is sufficient to enumerate only subsets that induce connected subgraphs (although we do not know how to do so efficiently).

Enumeration order. A further reduction in running time comes from the simple observation that if a subset U satisfies $D(U) \geq n$, then so do all of U's supersets, and so they need not be considered at all. A simple way to exploit this is to enumerate the subsets by constructing them incrementally, using the following recursive procedure. Fix an order on the vertices $v_1, \ldots, v_n$, and call the procedure below with $i = 0$, $U = \emptyset$.

```
Procedure **Enumerate**($i, U$)
1. If $i = n$, process the subgraph induced by $U$.
2. Else:
3.     Enumerate($i + 1, U$).
4.     If $|D(U \cup \{v_{i+1}\})| \leq n$, Enumerate($i + 1, U \cup \{v_{i+1}\}$).
```

This procedure is sensitive to the order of vertices; it makes sense to sort them by decreasing size of color set. A further improvement is to perform a preprocessing step in which all vertices u such that $|D_u| \geq n$ are removed from the list.

Early pruning of point colorings. It is not difficult to verify that the correctness of the algorithm is preserved if instead of just reporting each non-extensible point coloring it finds, it also immediately deletes it (so that the subsequent computation applies to the partially pruned graph). The advantage here is that we avoid having to consider over and over again point assignments that have already been reported as non-extensible. This eliminates the problem of the same point assignment being reported multiple times, and, more importantly, it also eliminates the superfluous chromatic-number computation performed each time the point coloring is unnecessarily considered. In addition, the algorithm can immediately halt (and report that the original graph is not colorable) if the color set of some vertex becomes empty. The downside of this heuristic is that the sets $D(U)$ grow smaller as the algorithm progresses, thus causing more subsets U to pass the $|D(U)| \leq |U|$ test and trigger unnecessary chromatic number computations. Of course, in this case it is an easy matter to eat the cake and have it too. We simply compute $|D(U)|$ with respect to the original graph, and do everything else on the (partially) pruned graph.

We remark that this last heuristic is especially powerful in conjunction with the **Enumerate** procedure above, because that procedure considers (to some extent) subsets before it considers their supersets. This allows the algorithm to detect, and prune, point assignments using small subsets, and so avoid chromatic-number computations on larger subgraphs. This phenomenon could be enhanced by enumerating the subsets in order of increasing size (i.e., first all singletons, then all pairs, then all triplets, etc.) but then it would not be easy to avoid enumerating subsets with $|D(U)| > n$. A hybrid approach might work best.

Speeding the chromatic number computation. In our algorithm we are really only interested in knowing whether the chromatic number of the extended graph G' is r, and not in finding its actual value. Thus we can preprocess G' as follows: as long as G' contains a vertex whose degree is less than r, delete this vertex. (The process is iterative: deleting a vertex decreases the degree of its neighbors, possibly causing them to be deleted too, etc.) The rationale is that deleting a vertex cannot increase the chromatic number of the graph, and, on the other hand, if after deleting the vertex the graph can be colored with r colors, then this coloring can be extended to the vertex by giving it a color different from the (at most $r-1$) colors given to its neighbors.

Redundant chromatic number computations. It is only necessary to perform a chromatic-number computation on failure subgraphs. Although a (non-failure) set U such that $|D(U)| = |U|$ must pass into the inner loop, it is quite possible that for a given point assignment (v, c), $v \in V \setminus U$, $c \in D(U)$, the color c appears in the color set of some vertex in U that is not adjacent to v. In such a case, applying the point coloring does not decrease $|D(U)|$ and does not turn U into a failure set, thus obviating the chromatic-number computation.

5 Experimental Results

We created a prototype implementation of our algorithm, incorporating most of the improvements mentioned in the previous section. The code was created by adapting and extending pre-existing generic CP code implementing the MAC-3 algorithms, and was not optimized for the needs of our algorithm. For chromatic-number computations, we used Michael Trick's implementation of an exact version of the DSATUR algorithm (available, as of July 5, 2006, at !http:// www . cs . sunysb . edu / algorith / implement / trick / distrib / trick.c!). We evaluated the code's performance on a Linux machine powered by a 3.6 GHz Intel Pentium 4 processor.

We performed two sets of performance testings: one using real workforce management data, and the other using synthetically generated data. Next we describe the data on which we ran the algorithm, and following that, the results we obtained.

Workforce management data. The real data we used originated in an instance of a workforce management (WM) problem in a certain department in IBM, the likes of which are routinely handled by that department. We were given a file containing 377 job descriptions. Each job description consisted of the dates during which the job was to be performed and the list of people qualified to perform it. The total number of people was 1111. In our model of the problem, jobs correspond to vertices, people correspond to colors, and pairs of jobs overlapping in time correspond to graph edges.

By running our algorithm on randomly selected subsets of jobs we discovered that subsets of more than 50 jobs were almost always unsatisfiable because there were quite a few instances of pairs of jobs overlapping in time that could only be performed by a single person—the same person. The algorithm would detect this almost instantly. In order to stress the algorithm and test its limits in a more meaningful manner, we artificially deleted the offending jobs (leaving one of each pair). We ended up with 312 jobs. We tested the algorithm on random subsets of these jobs, in sizes ranging from 20 to 300 jobs, at increments of ten. For each subset size $n = 20, 30, \ldots, 300$, we selected (uniformly and independently) ten subsets of n jobs and ran the algorithm on each subset. In total, we ran the algorithm on 290 instances.

Synthetic data. In addition to the WM data on which we evaluated our algorithm, we also tested it on randomly generated graphs. The random process by which the graphs were generated was controlled by four parameters: n, the number of vertices; p, the edge creation probability; m, the total number of colors; and k, the maximum color-set size. Each random graph was generated as follows (all random choices were made independently and uniformly). A set of n vertices was created. For each (unordered) pair of vertices, an edge was added between them with probability p. Then, for every vertex v a random number k_v was chosen in the range $1, 2, \ldots, k$, and then k_v colors were randomly chosen from the common pool of m colors. The values we used for the parameters were: n ranged from 20 to 100 at increments of 10; p was either 0.1, 0.3, or 0.6; $m = 300$ throughout; and k was either 10 or 20. For each set of parameters we generated ten graphs and ran the algorithm on them. In total, we ran the algorithm on 540 instances.

Results. We carried out two types of performance measurement, both on the same data sets. The first type is the absolute performance of a single call to the `SomeDifferent` propagator. The second type of measurement was comparative: we compared the running time of a CSP solver on two models, one consisting of a single `SomeDifferent` constraint, and the second consisting of the equivalent set of `NotEqual` constraints. In addition to the running time measurements we also collected various statistics concerning the input instances and the algorithm's run-time behavior. This extra data helped us understand the variation in the algorithm's performance. We used a CSP solver implementing MAC-3 and breaking arc-consistency by instantiating a random variable with a random value [12].

Figure 1 shows the running times of the propagator on the WM data instances. We see that the running time follows a slowly increasing curve from about 4msec (for 20 vertices) to about 100msec (for 300 vertices), but in the range 140–240 vertices, roughly half of the instances have increased running times, up to a factor of about 3 relative to the curve. Examination of these instances revealed that they entailed significantly more chromatic number computations than the other instances, which explains the increased running times.

We remark that up to 110 vertices nearly all instances were satisfiable, starting at 120 the percentage of unsatisfiable instances increased rapidly, and from 150 onward nearly all instances were unsatisfiable. It is interesting to note that the running times do not display a markedly different behavior in these different ranges.

In summary, despite the algorithm's worst case exponential complexity, it seems to perform well on the real-life instances we tested. The main contributing factor to this is the fact that the graphs decomposed into relatively small

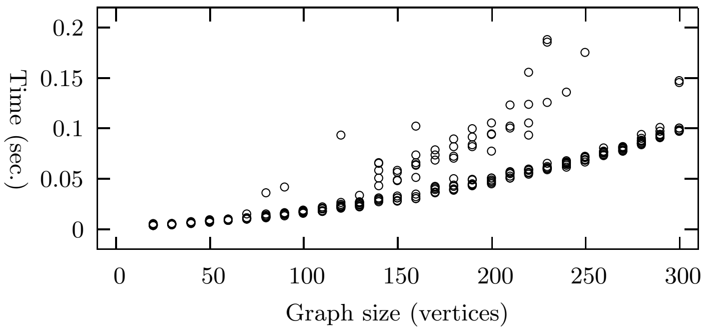


Fig. 1. Running time of the `SomeDifferent` propagator on WM data instances. Some of the instances shown were satisfiable, while others were unsatisfiable.

connected components. An earlier version that did not implement the *connected components* heuristic performed less well by several orders of magnitude.

Table 1. Running time (msec.) of the propagator on randomly generated graphs

	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$	$n = 70$	$n = 80$	$n = 90$	$n = 100$
$k = 10$									
$p = 0.1$	3.26	3.59	4.00	4.08	6.84	4.40	7.63	12.4	18.3
$p = 0.3$	3.27	3.64	4.93	9.39	345	13037	170416	471631	606915
$p = 0.6$	3.26	5.40	42.8	625	7065	121147	574879	607262	608733
$k = 20$									
$p = 0.1$	3.30	4.23	4.14	8.06	28.5	90.7	591	3954	13939
$p = 0.3$	3.31	5.27	12.5	42.1	167	1442	5453	29478	186453
$p = 0.6$	3.92	6.20	14.2	57.8	309	1398	6549	38196	270353

Table 1 shows the running times, in milliseconds, on the randomly generated graphs, all of which were satisfiable. The time shown in each table entry is the average over the ten instances. We see that for small graphs the algorithm's performance is similar to its performance on the WM data instances. However, as the graphs become larger, the running time increases steeply. The reason for this is the well known phenomenon whereby graphs generated by the random process we have described contain with high probability a "giant" connected component (i.e., one that comprises nearly all vertices), rendering the *connected components* heuristic powerless. All the same, the algorithm was able to handle graphs containing 100 vertices in approximately 10 minutes—well below what its worst case complexity would suggest.

We also see a significant difference between the cases $p = 0.1$ and $p \in \{0.3, 0.6\}$. At $p = 0.1$, the *giant component* effect had not set in in full force and the connected components were fairly small, whereas at $p \in \{0.3, 0.6\}$ the phenomenon was manifest and the giant component accounted for more than 90% of the graph.

There is also a noticeable difference between the cases $k = 10$ and $k = 20$. There are two contradictory effects at play here. A small value of k increases the likelihood of edges becoming superfluous due to their endpoints having disjoint color sets, thus mitigating the *giant component* effect. But at the same time, it reduces the color set sizes, thus diminishing the effectiveness of the *enumeration order* heuristic and forcing the algorithm to spend more time enumerating sets. Indeed, we have observed that in large connected components the algorithm spends most of its time enumerating sets that do not pass the $|D(U)| \leq |U|$ test; the chromatic number computations tend to be few and involve small subgraphs, and therefore do not contribute much to the running time. For $p = 0.1$ the graph is sparse enough so that the *giant component* effect is indeed canceled out by the emergence of superfluous edges, whereas for $p \in \{0.3, 0.6\}$, the small decrease in component size due to superfluous edges is outweighed by the effect of the smaller color sets.

We now turn to a comparison of the `SomeDifferent` and `NotEqual` models. We measured the time it took the solver to solve the CSP (or prove it unsatisfiable). When using `NotEqual` constraints, the backtrack limit was set to 10000. (The `SomeDifferent` model is backtrack free.)

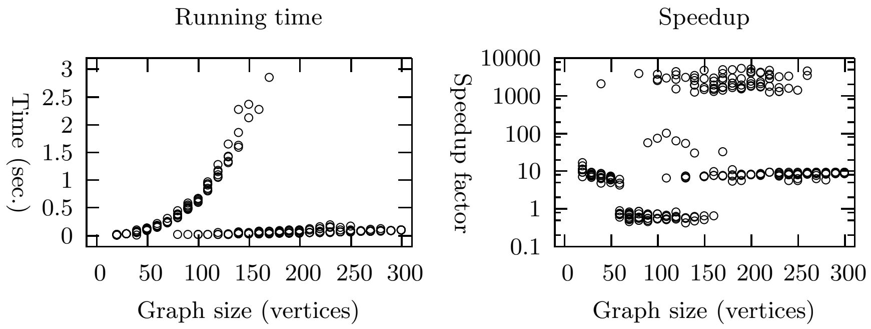


Fig. 2. Running time of the CSP solver on the `SomeDifferent` model, and speedup factor relative to `NotEqual`, on WM data instances

The graph on the left of Figure 2 shows the running times on the WM data instances using the `SomeDifferent` model. The graph on the right shows (in log-scale) the corresponding speedup factors relative to the `NotEqual` model. We see that the running times follow two curves. The rising curve represents the satisfiable instances, where the `SomeDifferent` propagator is called multiple times. The flat curve represents the unsatisfiable instances, where the unsatisfiability is established in a single `SomeDifferent` propagation. The speedup factors are grouped (on the logarithmic scale) roughly into three bands. The top band (consisting of 75 instances with speedup factors around 5000) represents instances that caused the solver to hit the backtrack limit. The middle band represents a mixture of 104 large instances identified as unsatisfiable and 42 small instances which the solver was able to satisfy, as well as 6 satisfiable instances on which the solver hit the backtrack limit. The lower band represents the remaining 63 instances, which are of intermediate size (60–160 vertices), and which were satisfied by the solver with a very small number of backtracks (nearly always 0). For these instances the NotEqual model was superior to the SomeDifferent model; the slowdown factors were around 2 (the maximum slowdown factor was 2.35).

Upon a closer examination of the data we observed that in nearly all cases (in all three bands) the NotEqual model either hit the backtrack limit or solved the CSP with no backtracks at all. For most cases in which the solver proved unsatisfiability, this was a result of the instances containing several vertices with singleton color sets which immediately forced inconsistency. The satisfiable cases had singleton color sets too, which definitely helped the solver. In contrast, our SomeDifferent propagator does not benefit from singleton color sets in any obvious manner. Adding a preprocessing stage (essentially, a NotEqual propagation stage) to handle them can obviously improve its performance.

In summary, our WM data instances seem biased in favor of the NotEqual model, due to the presence of singleton color sets. Nonetheless, the SomeDifferent approach still outperformed the NotEqual approach in the cases requiring backtracking. In virtually all of these cases, the NotEqual model hit the limit, whereas the SomeDifferent model performed quite well. Adding a preprocessing stage to our algorithm would make it as good as NotEqual propagation in the easy cases too.

On the random graph instances the variations in the speedup factors did not display easily understandable patterns. Generally speaking, the speedup factor started in the range 40–100 for 20 vertices, and declined roughly linearly to 1 at 40 or 50 vertices (depending on the k and p) parameters. For larger instances the speedup became slowdown, which grew to a maximum in the range 4000–6000 for the largest instances. Here too, all instances were satisfied in the NotEqual model with no backtracks. This demonstrates again that the main source of hardness for our algorithm, primarily the graph's connectivity, has little effect on the NotEqual approach.

6 Conclusion and Future Work

We have introduced the SomeDifferent constraint, which generalizes the familiar AllDifferent constraint in that it requires that only *some* pairs of variables take on different values. We have developed a hyper-arc consistency propagation algorithm for the SomeDifferent constraint. The problem is NP hard, and our algorithm's worst-case time complexity is exponential in the number of variables, but is largely independent of the domain sizes. We have also suggested and implemented some heuristics to improve the algorithm's running time in practice. We have offered empirical evidence suggesting that our algorithm is indeed practical.

Future work. The results we have obtained are encouraging, but clearly more experimentation with real data, preferably from a variety of application domains, is required in order to establish the usefulness of our approach. Also, we have already identified some areas for improvement, namely, adding a preprocessing stage to handle singleton color sets, and improving the set enumeration algorithm. A different, and important, research direction is to devise approximation techniques for unsatisfiable instances. A natural problem with practical importance is to remove as small (or cheap) a subset of vertices as possible in order to make the instance satisfiable.

Acknowledgments. We are grateful to Dan Connors and Donna Gresh for analyzing the raw workforce management data, extracting the parts pertinent to our work, and bringing them into a form we could use in our experiments.

References

1. G. Appa, D. Magos, and I. Mourtos. On the system of two all_different predicates. *Info. Process. Letters*, 94(3):99–105, 2005.
2. N. Barnier and P. Brisset. Graph coloring for air traffic flow management. In *Proc. of 4th CP-AI-OR workshop*, pages 133–147, 2002.
3. R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *J. of Algorithms*, 54(2):168–204, 2005.
4. H. L. Bodlaender and D. Kratsch. An exact algorithm for graph coloring with polynomial memory. Technical Report UU-CS-2006-015. Department of Information and Computing Sciences. Utrecht University, 2006.
5. D. Brelaz. New methods to color the vertices of a graph. *Communication of ACM*, 22(4):251–256, 1979.
6. J. M. Byskov. Exact algorithms for graph colouring and exact satisfiability. PhD thesis, University of Aarhus, Aarhus, Denmark. 2005.
7. R. Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, San Francisco, 2003.
8. D. Eppstein. Small maximal independent sets and faster excat graph coloring. *J. Graph Algorithms and Applications*, 7(2):131–140, 2003.
9. M. Grönkvist. A constraint programming model for tail assignment. In *Proc. of 1st CP-AI-OR Conf.*, pages 142–156, 2004.
10. W. Klotz. Graph coloring algorithms. Technical Report Mathematik-Bericht 2002/5 TU Clausthal, 2002.
11. E. L. Lawler. A note on the complexity of the chromatic number problem. *Info. Process. Letters*, 5(3):66–67, 1976.
12. Y. Naveh and R. Emek. Random stimuli generation for functional hardware verification as a cp application. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 882–882. Springer, 2005.
13. C.-G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for global cardinality constraint. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 600–614, 2003.
14. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. Technical Report R.R.LIRMM 93-068, 1993. Conf. version at AAAI-94.
15. M. Sellmann. Approximated consistency for knapsack constraints. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 679–693, 2003.
16. M. Sellmann. Cost-based filtering for shorter path constraints. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 694–708, 2003.

17. M. Sellmann. Approximated consistency for the automatic recording problem. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 822–826, 2005.
18. E. Tsang, J. Ford, P. Mills, R. Bradwell, R. Williams, and P. Scott. ZDC-rostering: A personnel scheduling system based on constraint programming. Technical Report 406, University of Essex, Colchester, UK, 2004.
19. W.-J. van Hoeve. The Alldifferent constraint: a systematic overview. 2006. Under construction. *http://www.cs.cornell.edu/∼vanhoeve.*
20. D. B. West. *Introduction to graph theory.* Prentice Hall, 2000.
21. R. Yang. Solving a workforce management problem with constraint programming. In *The 2nd International Conference on the Practical Application of Constraint Technology*, pages 373–387. The Practical Application Company Ltd, 1996.

Mini-bucket Elimination with Bucket Propagation

Emma Rollon and Javier Larrosa

Universitat Politecnica de Catalunya,
Jordi Girona 1-3, 08034 Barcelona, Spain
erollon@lsi.upc.edu, larrosa@lsi.upc.edu

Abstract. Many important combinatorial optimization problems can be expressed as *constraint satisfaction problems* with *soft constraints.* When problems are too difficult to be solved exactly, approximation methods become the best option. *Mini-bucket Elimination* (MBE) is a well known approximation method for combinatorial optimization problems. It has a control parameter z that allow us to trade time and space for accuracy. In practice, it is the space and not the time that limits the execution with high values of z. In this paper we introduce a new propagation phase that MBE should execute at each bucket. The purpose of this propagation is to jointly process as much information as possible. As a consequence, the undesirable lose of accuracy caused by MBE when splitting functions into different mini-buckets is minimized. We demonstrate our approach in *scheduling*, *combinatorial auction* and *max-clique* problems, where the resulting algorithm MBE^p gives important percentage increments of the lower bound (typically 50% and up to 1566%) with only doubling the cpu time.

1 Introduction

It is well recognized that many important problems belong to the class of combinatorial optimization problems. In general, combinatorial optimization problems are NP-hard. Therefore, they cannot be solved efficiently with current technologies. Then, the only thing that we can possibly do is to find near-optimal solutions. In that context, it is also desirable to have a quality measure of the solution. One way to achieve this goal is to provide a lower and an upper bound of the optimum. The smaller the gap, the closer we are to the true optimum.

Typically, best results are obtained developing *ad-hoc* techniques for the instances of interest. However, this requires a lot of work, including the time to learn specific domain peculiarities. An alternative, is to used generic techniques. Although they may not give so accurate results, it may be enough in some applications. Besides, they may provide the starting reference point to evaluate new *ad-hoc* techniques.

Mini-bucket Elimination (MBE) [1] is one of the most popular bounding techniques. Assuming minimization problems, MBE provides a lower bound of the optimum and can be combined with local search which provide upper bounds [1]. MBE is very general, since it can be applied to any problem that falls into the category of *graphical models*. Graphical models include very important optimization frameworks such as soft constraint satisfaction problems [2], Max-SAT, bayesian networks [3], etc. These frameworks have important applications in fields such as *routing* [4], *bioinformatics* [5], *scheduling* [6] or *probabilistic reasoning* [7]. The good performance of MBE in different contexts has been widely proved [1,8,7].

Interestingly, MBE has a parameter z which allow us to trade time and space for accuracy. With current computers, it is the space and not the time what bounds the maximum value of z that can be used in practice. In our previous work [9], we introduced a set of improvements on the way MBE handles memory. As a result, MBE became orders of magnitude more efficient. Thus, higher values of z can be used which, in turn, yields significantly better bounds. In this paper we continue improving the practical applicability of MBE. In particular, we introduce a new propagation phase that MBE must execute at each bucket. Mini-buckets are structured into a tree and costs are moved along branches from the leaves to the root. As a result, the root mini-bucket accumulates costs that will be processed together, while classical MBE would have processed them independently. Note that the new propagation phase does not increase the complexity with respect classical MBE.

Our experiments on *scheduling*, *combinatorial auctions* and *maxclique* show that the addition of this propagation phase increases the quality of the lower bound provided by MBE quite significatively. Although the increase depends on the benchmark, the typical percentage is 50%. However, for some instances, the propagation phase gives a dramatic percentage increment up to 1566%.

2 Preliminaries

2.1 Soft CSP

Let $\mathcal{X} = (x_1, \dots, x_n)$ be an ordered set of variables and $\mathcal{D} = (D_1, \dots, D_n)$ an ordered set of domains, where D_i is the finite set of potential values for x_i. The assignment (i.e, instantiation) of variable x_i with $a \in D_i$ is noted $(x_i \leftarrow a)$. A *tuple* t is an ordered set of assignments to different variables $(x_{i_1} \leftarrow a_{i_1}, \dots, x_{i_k} \leftarrow a_{i_k})$. The *scope* of t, noted $var(t)$, is the set of variables that it assigns. The *arity* of t is $|var(t)|$. The *projection* of t over $Y \subseteq var(t)$, noted $t[Y]$, is a sub-tuple of t containing only the instantiation of variables in Y. Let t and s be two tuples having the same instantiations to the common variables. Their *join*, noted $t \cdot s$, is a new tuple which contains the assignments of both t and s. Projecting a tuple t over the empty set $t[\emptyset]$ produces the empty tuple λ. We say that a tuple t is a *complete instantiation* when $var(t) = \mathcal{X}$. In the following, abusing notation, when we write $\forall_{t \in Y}$ we will mean $\forall_{t \text{ s.t. } var(t)=Y}$.

Let A be an ordered set of values, called valuations, and $+$ a conmutative and associative binary operation $+ : A \times A \rightarrow A$ such that exists an identity element

[1] In the original description MBE also provides an upper bound, but in this paper we will disregard this feature.

0 (namely, $\forall a \in A,\ a + 0 = a$), and satisfies monotonicity (namely, $\forall a, b, b \in A$, if $a \geq b$ then $(a + b \geq b + c)$).

$\mathcal{F} = \{f_1, \ldots, f_r\}$ is a set of functions. Each function f_j is defined over a subset of variables $var(f_j) \subseteq \mathcal{X}$ and returns values of A (namely, if $var(t) = var(f_j)$ then $f_j(t) \in A$). For convenience, we allow to evaluate $f_j(t)$ when $var(t) \supset var(f_j)$, being equivalent to $f_j(t[var(f_j)])$. In this paper we assume functions explicitly stored as tables.

A *soft CSP* is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{F})$ where each function $f \in \mathcal{F}$ specifies how good is each different partial assignment of $var(f)$. The sum $+$ is used to *aggregate* values from different functions. The global quality of an assignment is the sum of values given by all the functions. The usual task of interest is to find the best complete assignment $\mathcal{X}$ in terms of A. Different soft CSP frameworks differ in the semantics of A. Well-known frameworks include *probabilistic* CSPs, *weighted* CSPs, *fuzzy* CSPs, etc [2].

A soft CSP framework is *fair* [10] if for any pair of valuations $\alpha, \beta \in A$, with $\alpha \leq \beta$, there exists a maximum difference of β and α. This unique maximum difference of β and α is denoted by $\beta - \alpha$. This property ensures the equivalence of the problem when the two operations $+$ and $-$ are applied. In [10] it is shown that the most important soft constraint frameworks are fair. Although our approach can be used in any fair soft constraint framework, for the sake of simplicity, we will focus on weighted CSPs. In weighted CSPs (WCSPs) A is the set of natural numbers, $+$ and $-$ are the usual sum and subtraction. Thus, the set of soft constraints define the following objective function to be minimized,

$$F(X) = \sum_{i=1}^{r} f_i(X)$$

2.2 Operations over Functions

– The *sum* of two functions f and g denoted $(f + g)$ is a new function with scope $var(f) \cup var(g)$ which returns for each tuple $t \in var(f) \cup var(g)$ the sum of costs of f and g,

$$(f + g)(t) = f(t) + g(t)$$

– Let f and g be two functions such that $var(g) \subseteq var(f)$ and $\forall t \in var(f)$, $f(t) \geq g(t)$. Their *subtraction*, noted $f - g$ is a new function with scope $var(f)$ defined as,

$$(f - g)(t) = f(t) - g(t)$$

for all tuple $t \in var(f)$.

– The *elimination* of variable x_i from f, denoted $f \downarrow x_i$, is a new function with scope $var(f) - \{x_i\}$ which returns for each tuple t the minimum cost extension of t to x_i,

$$(f \downarrow x_i)(t) = \min_{a \in D_i} \{f(t \cdot (x_i \leftarrow a))\}$$

```
function BE($\mathcal{X}, \mathcal{D}, \mathcal{F}$)
1. for each $i = n..1$ do
2.     $\mathcal{B} := \{f \in F |\ x_i \in var(f)\}$
3.     $g := (\sum_{f \in \mathcal{B}} f) \downarrow x_i$;
4.     $F := (F \cup \{g\}) - \mathcal{B}$;
5. endfor
6. return($\mathcal{F}$);
endfunction
```

Fig. 1. Bucket Elimination. Given a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{F})$, the algorithm returns $\mathcal{F}$ containing a constant function with the optimal cost.

where $t \cdot (x_i \leftarrow a)$ means the extension of t so as to include the assignment of a to x_i. Observe that when f is a unary function (*i.e.*, arity one), eliminating the only variable in its scope produces a constant.

– The *projection* of function f over $Y \subset var(f)$, denoted $f[Y]$, is a new function with scope Y which returns for each tuple t the minimum cost extension of t to $var(f)$,

$$(f[Y])(t) = \min_{t' \in var(f) \text{ s.t. } t' = t \cdot t''} f(t')$$

Observe that variable elimination and projection are related with the following property,

$$(f \downarrow x_i) = f[var(f) - \{x_i\}]$$

2.3 Bucket and Mini-bucket Elimination

Bucket elimination (BE, Figure 1)[11,12] is a well-known algorithm for weighted CSPs. It uses an arbitrary variable ordering o that we assume, without loss of generality, lexicographical (i.e, $o = (x_1, x_2, \ldots, x_n)$). The algorithm eliminates variables one by one, from last to first, according to o. The elimination of variable x_i is done as follows: $\mathcal{F}$ is the set of current functions. The algorithm computes the so called *bucket* of x_i, noted $\mathcal{B}$, which contains all cost functions in $\mathcal{F}$ having x_i in their scope (line 2). Next, BE computes a new function g by summing all functions in $\mathcal{B}$ and subsequently eliminating x_i (line 3). Then, $\mathcal{F}$ is updated by removing the functions in $\mathcal{B}$ and adding g (line 4). The new $\mathcal{F}$ does not contain x_i (all functions mentioning x_i were removed) but preserves the value of the optimal cost. The elimination of the last variable produces an empty-scope function (*i.e.*, a constant) which is the optimal cost of the problem. The time and space complexity of *BE* is exponential in a structural parameter called *induced width*. In practice, it is the space and not the time what makes the algorithm unfeasible in many instances.

Mini-bucket elimination (MBE) [1] is an approximation of BE that can be used to bound the optimum when the problem is too difficult to be solved exactly. Given a control parameter z, MBE partitions buckets into smaller subsets called mini-buckets such that their join arity is bounded by $z + 1$. Each mini-bucket is processed independently. Consequently, the output of MBE is a lower bound of the true optimum. The pseudo-code of MBE is the result of replacing lines 3 and 4 in the algorithm of Figure 1 by,

```
3.      $\{\mathcal{P}_1, \ldots, \mathcal{P}_k\} :=$ Partition($\mathcal{B}$);
3b.     for each $j = 1..k$ do $g_j := (\sum_{f \in \mathcal{P}_j} f) \downarrow x_i$;
4.      $F := (F \cup \{g_1, \ldots, g_k\}) - \mathcal{B}$;
```

The time and space complexity of MBE is $O(d^{z+1})$ and $O(d^z)$, respectively. Parameter z allow us to trade time and space for accuracy, because greater values of z increment the number of functions that can be included in each mini-bucket. Therefore, the bounds will be presumably tighter. MBE constitutes a powerful yet extremely general mechanism for lower bound computation.

3 Equivalence-Preserving Transformations in Fair Frameworks

We say that two WCSPs are equivalent if they have the same optimum. There are several transformations that preserve the equivalence. For instance, if we take any pair of cost functions $f, g \in \mathcal{F}$ from a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{F})$ and replace them by their sum $f + g$, the result is an equivalent problem. The replacement of $\mathcal{B}$ by g performed by BE (Figure 1) is another example of equivalence-preserving transformation. Very recently, a new kind of WCSP transformation has been used in the context of soft local consistency [13,14]. The general idea is to *move* costs from one cost function to another. More precisely, costs are subtracted from one cost function and added to another. Formally, let f and h be two arbitrary functions. The *movement of costs* from f to g is done sequentially in three steps:

$h := f[var(f) \cap var(g)]$
$f := f - h$
$g := g + h$

In words, function h contains costs in f that can be captured in terms of the common variables with g. Hence, they can be kept either in h or in f. Then, this costs are moved from f to g. The time complexity of this operation is $O(d^{\max\{|var(f)|,|var(g)|\}})$. The space complexity is the size of h stored as a table, $O(d^{|var(h)|\}})$, which is negligible in comparison with the larger function f.

Example 1. Consider the functions on Figure 2 (a). They are defined over boolean domains and given as a table of costs. Let function h represents the costs that can be moved from function f to function g. Observe that, as f and g only share variable x_i, then $h = f[x_i]$, where $h(false) = 2$ and $h(true) = 4$. Figure 2 (b), shows the result of moving the costs from f to g. Observe that costs of tuples t such that $var(t) = \{x_i, x_j, x_k\}$ are preserved.

g: x_i	x_j	
f	f	5
f	t	4
t	f	1
t	t	6

f: x_i	x_k	
f	f	2
f	t	3
t	f	4
t	t	5

(a)

g: x_i	x_j	
f	f	7
f	t	6
t	f	5
t	t	10

f: x_i	x_k	
f	f	0
f	t	1
t	f	0
t	t	1

(b)

$g \downarrow x_i$ x_j	
f	1
t	4

$f \downarrow x_i$ x_k	
f	2
t	3

(c)

$g \downarrow x_i$ x_j	
f	5
t	6

$f \downarrow x_i$ x_k	
f	0
t	1

(d)

Fig. 2. Example of functions

4 Mini Buckets with Propagation

In this Section we introduce a refinement of MBE. It consists on performing a movement of costs in each bucket before processing it. We incorporate the concept of equivalence-preserving transformation into MBE, but only at the bucket level. The idea is to *move* costs between minibuckets aiming at a propagation effect. We pursue the accumulation of as much information as possible in one of the mini-buckets.

The following example illustrates and motivates the idea. Suppose that MBE is processing a bucket containing two functions f and g, each one forming a mini-bucket. Variable x_i is the one to be eliminated. Standard MBE would process independently each minibucket, eliminating variable x_i in each function. It is precisely this independent elimination of x_i from each mini-bucket where the lower bound of MBE may lose accuracy. Ideally (i.e, in BE), f and g should be added and their information should *travel* together along the different buckets. However, in MBE their information is split into two pieces for complexity reasons. What we propose is to transfer costs from f to g (or conversely) before processing the mini-buckets. The purpose is to put as much information as possible in the same mini-bucket, so that all this information is jointly processed as BE would do. Consequently, the pernicious effect of splitting the bucket into mini-buckets will presumably be minimized. Figure 2 depicts a numerical illustration. Consider functions f and g from Figure 2 (a). If variable x_i is eliminated independently, we obtain the functions in Figure 2 (c). If the problem contains no more functions, the final lower bound will be 3. Consider now the functions in Figure 2 (b) where costs have been moved from f to g. If variable x_i is eliminated independently, we obtain the functions in Figure 2 (d), with which the lower bound is 5.

The previous example was limited to two mini-buckets containing one function each. Nevertheless, the idea can be easily generalized to arbitrary mini-bucket arrangements. At each bucket $\mathcal{B}$, we construct a *propagation tree* $T = (V, E)$ where nodes are associated with mini-buckets and edges represent movement of

```
function $MBE^p(z)$
1. for each $i = n..1$ do
2.     $\mathcal{B} := \{f \in F | \; x_i \in var(f)\}$;
3.     $\{\mathcal{P}_1, \ldots, \mathcal{P}_k\} :=$ Partition$(\mathcal{B}, z)$;
4.     for each $j = 1..k$ do $g_j := \sum_{f \in \mathcal{P}_j} f$;
5.     $(V, E) :=$ PropTree$(\{g_1, \ldots, g_k\})$;
6.     Propagation$((V, E))$;
7.     for each $j = 1..k$ do $g_j := g_j \downarrow x_i$;
8.     $F := (F \cup \{g_1, \ldots, g_k\}) - \mathcal{B}$;
9. endfor
10. return$(g_1)$;
endfunction
procedure Propagation$((V, E))$
11. repeat
12.     select a node $j$ s.t it has received the messages from all its children;
13.     $h_j := g_j[var(g_j) \cap var(g_{parent(j)})]$;
14.     $g_j := g_j - h_j$;
15.     $g_{parent(j)} := g_{parent(j)} + h_j$;
16. until root has received all messages from its children;
endprocedure
```

Fig. 3. Mini-Bucket Elimination with Propagation (preliminary version). Given a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{F})$, the algorithm returns a zero-arity function g_1 with a lower bound of the optimum cost.

costs along branches from the leaves to the root. Each node waits until receiving costs from all its children. Then, it sends costs to its parent. This flow of costs accumulates and propagates costs towards the root.

The refinement of MBE that incorporates this idea is called MBE^p. In Figure 3 we describe a preliminary version. A more efficient version regarding space will be discussed in the next subsection. MBE^p and MBE are very similar and, in the following, we discuss the main differences. After partitioning the bucket into mini-buckets (line 3), MBE^p computes the sum of all the functions in each mini-bucket (line 4). Next, it constructs a propagation tree $T = (V, E)$ with one node j associated to each function g_j. Then, costs are propagated (lines 6, 11-16). Finally, variable x_i is eliminated from each mini-bucket (line 7) and resulting functions are added to the problem in replacement of the bucket (line 8).

Procedure `Propagation` is also depicted in Figure 3. Let j be an arbitrary node of the propagation tree such that has received costs from all its children. It must send costs to its parent $parent(j)$. First, it computes in function h_j the costs that can be sent from j to its parent (line 13). Then, function h_j is subtracted from g_j and summed to $g_{parent(j)}$ (lines 14 and 15). The propagation phase terminates when the root receives costs from all its children.

4.1 Improving the Space Complexity

Observe that the previous implementation of MBE^p (Figure 3) computes in two steps (lines 4 and 7), what plain MBE computes in one step. Consequently,

MBE^p stores functions with arity up to $z+1$ while MBE only stores functions with arity up to z. Therefore, the previous description of MBE^p has a space complexity slightly higher than MBE, given the same value of z. In the following, we show how the complexity of MBE^p can be made similar to the complexity of MBE. First, we extend the concept of movement of costs to deal with sets of functions. Let F and G be two sets of costs functions. Let $var(F) = \cup_{f \in F} var(f)$, $var(G) = \cup_{g \in G} var(g)$ and $Y = var(F) \cap var(G)$. The *movement of costs* from F to G is done sequentially in three steps:

$h := (\sum_{f \in F} f)[Y]$
$F := F \cup \{-h\}$
$G := G \cup \{h\}$

where $-h$ means that costs contained in h are to be subtracted instead of summed, when evaluating costs of tuples on F. Observe that the first step can be efficiently implemented as,

$$\forall_{t \in Y}, h(t) := \min_{(t' \in var(F) \text{ s.t. } t'=t \cdot t'')}\{\textstyle\sum_{f \in F} f(t')\}$$

This implementation avoids computing the sum of all the functions in F. The time complexity of the operation is $O(d^{|var(F)|})$. The space complexity is $O(d^{|Y|})$.

Figure 4 depicts the new version of MBE^p. The difference with the previous version is that functions in mini-buckets do not need to be summed before the propagation phase (line 4 is omitted). Procedure `Propagation` moves costs between mini-buckets preserving the set of original functions. Line 7, sums the functions in the mini-buckets and eliminates variable x_i in one step, as plain MBE would do.

Observe that the time complexity of line 13 is $O(d^{z+1})$, because $|var(\mathcal{P}_j)| \leq z+1$ (by definition of mini-bucket). The space complexity is $O(d^z)$ because $|var(h)| \leq z$ (note that $var(\mathcal{P}_j) \neq var(\mathcal{P}_{parent(j)})$ because otherwise they would have been merged into one mini-bucket). The previous observation leads to the following result.

Theorem 1. *The time and space complexity of MBE^p is $O(d^{z+1})$ and $O(d^z)$, respectively, where d is the largest domain size and z is the value of the control parameter.*

4.2 Computation of the Propagation Tree

In our preliminary experiments we observed that the success of the propagation phase of MBE^p greatly depends on the flow of information, which is captured in the propagation tree. In the following we discuss two ideas that heuristically lead to good propagation trees. Then, we will propose a simple method to construct good propagation trees.

For the first observation, consider MBE with $z=1$ in a problem with four binary functions $f_1(x_1,x_2), f_2(x_2,x_3), f_3(x_2,x_4), f_4(x_3,x_4)$. Variable x_4 is the first to be eliminated. Its bucket contains f_3 and f_4. Each function forms a mini-bucket. MBE^p must decide whether to move costs from f_3 to f_4 or conversely.

```
function $MBE^p(z)$
1.  for each $i = n..1$ do
2.      $\mathcal{B} := \{f \in F |\ x_i \in var(f)\}$;
3.      $\{\mathcal{P}_1, \ldots, \mathcal{P}_k\} :=$ Partition($\mathcal{B}, z$);
5.      $(V, E) :=$ PropTree($\{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$);
6.      Propagation($(V, E)$);
7.      for each $j = 1..k$ do $g_j := ((\sum_{f \in \mathcal{P}_j} f) - h_j) \downarrow x_i$;
8.      $F := (F \cup \{g_1, \ldots, g_k\}) - \mathcal{B}$;
9.  endfor
10. return($g_1$);
endfunction
procedure Propagation($(V, E)$)
11. repeat
12.     select a node $j$ s.t it has received the messages from all its children;
13.     $h_j := (\sum_{f \in \mathcal{P}_j} f)[var(\mathcal{P}_j) \cap var(\mathcal{P}_{parent(j)})]$;
14.     $\mathcal{P}_j := \mathcal{P}_j \cup \{-h_j\}$;
15.     $\mathcal{P}_{parent(j)} := \mathcal{P}_{parent(j)} \cup \{h_j\}$;
16. until root has received all messages from its children;
endprocedure
```

Fig. 4. Mini-Bucket Elimination with Propagation. Given a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{F})$, the algorithm returns a zero-arity function g_1 with a lower bound of the optimum cost.

Observe that after the elimination of x_4, f_4 will go to the bucket of x_3 where it will be summed with f_2. Then, they will go to the bucket of x_2. However, f_3 will *jump* directly to the bucket of x_2. For this reason, it seems more appropriate to move costs from f_3 to f_4. In f_4 the costs go to a higher mini-bucket, so they have more chances to propagate useful information. One way to formalize this observation is the following: We associate to each mini-bucket $\mathcal{P}_j$ a binary number $N_j = b_n b_{n-1} \ldots b_1$ where $b_i = 1$ iff $x_i \in \mathcal{P}_j$. We say that mini-bucket $\mathcal{P}_j$ is smaller than $\mathcal{P}_k$ (noted $\mathcal{P}_j < \mathcal{P}_k$) if $N_j < N_k$. In our propagation trees parents will always be larger than their children.

For the second observation, consider three functions $f(x_7, x_6, x_5, x_4)$, $g(x_7, x_3, x_2, x_1)$, $h(x_7, x_6, x_5, x_1)$. Observe that f shares 1 variable with g and 3 with h. The number of common variables determines the arity of the function that is used as a *bridge* in the cost transfer. The narrower the bridge, the less information that can be captured. Therefore, it seems better to move costs between f and h than between f and g.

In accordance with the two previous observations, we construct the propagation tree as follows: the parent of mini-bucket $\mathcal{P}_u$ will be mini-bucket $\mathcal{P}_w$ such that $\mathcal{P}_u < \mathcal{P}_w$ and they share a maximum number of variables. This strategy combines the two criteria discussed above.

5 Experimental Results

We have tested our approach in three different domains. The purpose of the experiments is to evaluate the effectiveness of the propagation phase and the impact of the propagation tree on that propagation. To that end, we compare the lower bound obtained with three algorithms: standard MBE, MBE with bucket propagation using as a propagation tree a chain of mini-buckets randomly ordered (i.e., MBE^p_r), and MBE with bucket propagation using a propagation tree heuristically built as explained in Section 4.2 (i.e., MBE^p_h). For each domain, we execute those three algorithms with different values of the control parameter z in order to analyze its effect (the highest value of z reported is the highest feasible value given the available memory). In all our experiments, the order of variable elimination is established with the *min-fill* heuristic. All the experiments are executed in a Pentium IV running Linux with 2Gb of memory and 3 GHz.

5.1 Scheduling

For our first experiment, we consider the scheduling of an earth observation satellite. We experiment with instances from Spot5 satellite [15]. These instances have unary, binary and ternary cost functions, and domains of size 2 and 4. Some instances include in their original formulation an additional capacity constraint that we discard on this benchmark.

Figure 5 shows the results. The first column identifies the instance. The second column indicates the value of the control parameter z with which the algorithms are executed. Columns third and fourth report the lower bound obtained and the execution time for standard MBE, respectively. Columns fifth and sixth indicates for MBE^p_r the percentage increment of the lower bound measured as $((Lb_{MBE^p_r} - Lb_{MBE})/Lb_{MBE}) * 100$ and the execution time. Columns seventh and eighth reports the same information for MBE^p_h.

The first thing to be observed is that the results obtained with MBE^p_r does not follow a clear tendency. MBE^p_r increases and decreases the lower bound obtained with standard MBE almost the same times. However, MBE^p_h increases the lower bound obtained with MBE for all the instances. Moreover, when both MBE^p_r and MBE^p_h increase the lower bound, MBE^p_h is always clearly superior. Therefore, it is clear that an adequate propagation tree impacts on the bounds obtained.

Regarding MBE^p_h, it increases up to 139% the lower bound with respect MBE (e.g. instance 408). The mean increment is 54%, 38%, and 28% when the value of the control parameter z is 10, 15, and 20, respectively. Note that the effect of the propagation is higher for lower values of z because, as we increase the value of z, the number of functions in each mini-bucket increases and the number of mini-buckets decreases. Therefore, the propagated information also decreases and the effect of the propagation is diminished. Moreover, the lower bounds obtained with MBE^p_h and z set to 10 outperforms the ones obtained with MBE and z set to 20 in almost all the instances, which means that the time and space required for obtaining a bound of a given quality is decreased.

Regarding cpu time, MBE^p_h is from 2 to 3 times slower than MBE. The reason is that cost functions are evaluated twice: the first one during the propagation phase for establishing the costs to be moved, and the second one during the regular process of variable elimination. However, it is important to note that it

Instance	z	MBE(z)		MBE_r^p(z)		MBE_h^p(z)	
		Lb.	Time(sec.)	%	Time(sec.)	%	Time(sec.)
1506	20	184247	827.63	1.6	1628.93	29.8	1706.6
	15	163301	25.43	-5.5	51.48	30.6	51.39
	10	153274	1.33	-13.7	2.65	21.5	2.64
1401	20	184084	691.08	16.8	1469.36	58.6	1574.26
	15	170082	20.82	4.7	47.35	45.8	46.92
	10	155075	1.02	-10.3	2.13	53.5	2.17
1403	20	181184	814.55	7.1	1702.82	59.6	1919.48
	15	162170	27.82	7.3	55.94	57.3	56.9
	10	146155	1.3	10.9	2.58	60.2	2.6
1405	20	191258	1197.06	0.5	2537.64	42.3	2622.88
	15	169233	33.88	-2.3	93.88	54.9	81.17
	10	142206	1.7	-25.3	3.51	64.7	3.5
1407	20	191342	1415.91	-4.0	2935.78	53.8	3008.78
	15	166298	47.44	3.5	94.17	60.1	102.78
	10	144264	2.03	13.8	4.19	68.6	4.23
28	20	134105	252.14	2.2	500.97	38.0	510.72
	15	121105	7.77	-1.6	15	52.8	16.16
	10	103105	0.36	16.4	0.71	49.4	0.71
29	20	8058	4.92	-0.01	5.3	0.01	5.32
	15	8055	0.28	-0.1	0.34	0.02	0.34
	10	8050	0.01	-0.01	0.02	0.07	0.02
408	20	5212	51.19	19.1	75.39	19.3	72.5
	15	5200	2.11	18.7	3.29	19.3	3.41
	10	2166	0.11	38.1	0.2	139.0	0.2
412	20	17314	167.91	5.4	278.29	40.5	278.7
	15	15270	6.49	6.2	10.98	72.1	11.1
	10	10233	0.27	87.8	0.5	88.4	0.78
414	20	23292	629.36	-12.9	1278.39	17.4	1306.98
	15	18268	20.14	-16.3	42.87	49.4	42.99
	10	16213	1.05	-31.0	2.35	49.8	2.09
42	20	127050	38.9	-4.7	71.47	7.8	68.35
	15	111050	1.43	-1.8	2.52	14.4	2.55
	10	93050	0.06	2.1	0.12	19.3	0.12
505	20	19240	51.36	-36.3	66.9	5.2	63.16
	15	16208	2.2	-18.5	3.35	0.1	3.23
	10	13194	0.15	-15.2	0.21	15.1	0.21
507	20	16292	276.74	-6.1	510.66	0.2	520.3
	15	14270	9.84	6.7	19.01	42.2	18.88
	10	11226	0.47	8.6	0.92	53.7	0.92
509	20	22281	507.64	4.6	1026.43	22.5	1046.89
	15	20267	16.2	-24.6	34.68	34.7	34.72
	10	14219	0.83	14.0	1.64	77.7	1.62

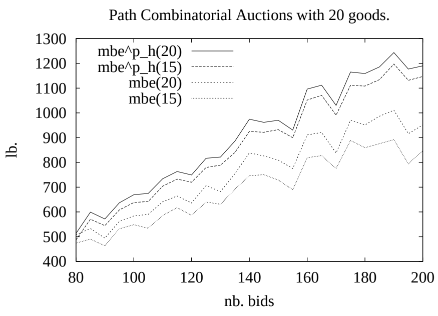
Fig. 5. Experimental results on Spot5 instances

is the space and not the time what bounds the maximum value of z that can be used in practice. As a consequence, that constant increase in time is not that significant as the space complexity remains the same.

5.2 Combinatorial Auctions

Combinatorial auctions (CA) allow bidders to bid for indivisible subsets of goods. Consider a set of goods $\{1, 2, \ldots, n\}$ that go on auction. There are m bids. Bid j is defined by the subset of requested goods $X_j \subseteq \{1, 2, \ldots, n\}$ and the money offer b_j. The bid-taker must decide which bids are to be accepted maximizing the benefits.

We have generated CA using the *path* and *regions* model of the CATS generator [16]. We experiment on instances with 20 and 50 goods, varying the number



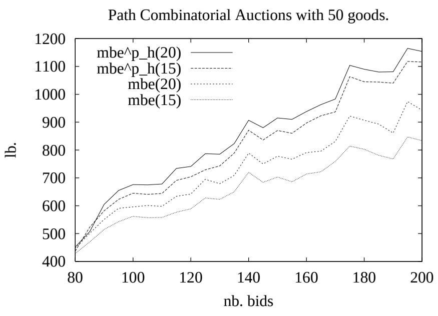


Fig. 6. Combinatorial Auctions. Path distribution.

of bids from 80 to 200. For each parameter configuration, we generate samples of size 10. We execute algorithms MBE, MBE_r^p, and MBE_h^p with z equal to 15 and 20. We do not report results with MBE_r^p because it was always very inferior than MBE_h^p. For space reasons, we only report results on the *path* model. The results for the *regions* model follows the same pattern.

Figure 6 reports the results for *path* instances with 20 and 50 goods, respectively. As can be observed, the behaviour for both configurations is almost the same. Regarding the algorithms, it is clear that MBE_h^p always outperformes MBE. Note that the lower bound obtained with $MBE_h^p(z = 15)$ is clearly superior than that obtained with $MBE(z = 20)$. Moreover, as pointed out in the previous domain, the effect of the propagation in each sample point is higher for $z = 15$ than for $z = 20$. That is, the percentage of increment in the lower bound obtained with $MBE_h^p(z = 15)$ is higher than that of $MBE_h^p(z = 20)$. Finally, it is important to note that the impact of the propagation is higher when the problems become harder (i.e., as the number of bids increase).

5.3 Maxclique

A *clique* of a graph $G = (V, E)$ is a set $S \subseteq V$, such that every two nodes in S are joined by an edge of E. The *maximum clique problem* consists on finding the largest cardinality of a clique. The maximum clique problem can be easily encoded as a minimization problem (i.e., minimize the number of nodes in $V - S$).

We test our approach on the dimacs benchmark [17]. Figure 7 reports the results. The first column identifies the instance. The second column indicates the value of the control parameter z with which the algorithms are executed. The third column report the lower bound obtained with standard MBE. Columns fourth and fifth indicates, for MBE_r^p and MBE_l^p, the percentage of increment in the lower bound with respect MBE, respectively. As the behaviour of the cpu time is the same as for the previous benchmark, we do not report this information.

MBE_r^p increases the lower bound obtained with standard MBE for all the instances except for those of *hamming* and *johnson*. The percentage of increment is up to 1226% when the value of the control parameter z is 10, and up to 812% when z is the highest value. The best results are obtained with MBE_h^p which

Instance	z	MBE	MBE_r^p	MBE_h^p
		Lb.	%	%
brock200-1	18	66	30.3	48.4
	10	51	52.9	78.4
brock200-2	18	55	67.2	103.6
	10	29	200	268.9
brock200-3	18	64	48.4	68.7
	10	38	139.4	173.6
brock200-4	18	63	36.5	65.0
	10	41	121.9	131.7
brock400-1	18	79	100	141.7
	10	46	256.5	273.9
brock400-2	18	75	114.6	157.3
	10	44	261.3	277.2
brock400-3	18	87	88.5	114.9
	10	44	250	286.3
brock400-4	18	76	106.5	160.5
	10	47	248.9	289.3
brock800-1	18	71	336.6	454.9
	10	41	675.6	773.1
brock800-2	18	63	395.2	520.6
	10	37	748.6	875.6
brock800-3	18	68	352.9	483.8
	10	44	604.5	706.8
brock800-4	18	71	343.6	460.5
	10	36	758.3	902.7
c-fat200-1	18	71	32.3	78.8
	10	62	27.4	112.9
c-fat200-2	18	63	38.0	82.5
	10	48	77.0	156.2
c-fat200-5	18	55	23.6	12.7
	10	37	32.4	70.2
c-fat500-10	18	77	115.5	123.3
	10	52	173.0	253.8
c-fat500-1	18	132	84.0	137.1
	10	107	126.1	196.2
c-fat500-2	18	108	108.3	164.8
	10	85	160	254.1
c-fat500-5	18	83	145.7	202.4
	10	74	163.5	264.8
hamming10-2	18	412	-66.9	-72.0
	10	419	-72.0	-73.7
hamming10-4	18	119	264.7	413.4
	10	77	451.9	720.7
hamming6-2	18	32	-28.1	-31.2
	10	32	-50	-59.3
hamming6-4	18	45	-4.4	2.2
	10	33	9.0	33.3
hamming8-2	18	114	-59.6	-64.9
	10	113	-74.3	-78.7
hamming8-4	18	82	46.3	89.0
	10	51	113.7	215.6
johnson16-2-4	18	72	-4.1	11.1
	10	56	10.7	48.2
johnson32-2-4	18	195	27.6	71.2
	10	134	75.3	150
johnson8-2-4	18	23	-4.3	0
	10	20	-20	-5
johnson8-4-4	18	45	-22.2	-11.1
	10	40	-15	-10
keller4	18	70	27.1	54.2
	10	41	97.5	168.2
keller5	18	90	246.6	394.4
	10	61	414.7	634.4
MANN-a27	15	247	0.4	0.4
	10	244	-1.2	0.8

Instance	z	MBE	MBE_r^p	MBE_h^p
		Lb.	%	%
MANN-a45	15	677	-0.7	0.4
	10	671	-0.1	0.1
MANN-a81	15	2177	0.0	0.3
	10	2171	-0.1	0.5
p-hat1000-1	15	85	380	654.1
	10	63	577.7	873.0
p-hat1000-2	15	57	589.4	821.0
	10	36	1013.8	1325
p-hat1000-3	15	82	364.6	415.8
	10	50	668	764
p-hat1500-1	15	69	802.8	1292.7
	10	82	686.5	1021.9
p-hat1500-2	15	64	812.5	1112.5
	10	45	1226.6	1566.6
p-hat1500-3	15	79	624.0	706.3
	10	54	924.0	1111.1
p-hat300-1	18	62	112.9	195.1
	10	48	187.5	306.2
p-hat300-2	18	61	121.3	168.8
	10	38	247.3	328.9
p-hat300-3	18	76	71.0	100
	10	51	145.0	172.5
p-hat500-1	18	74	170.2	301.3
	10	50	330	524
p-hat500-2	18	75	178.6	248
	10	39	407.6	556.4
p-hat500-3	18	93	125.8	169.8
	10	50	300	338
p-hat700-1	15	66	340.9	581.8
	10	52	482.6	711.5
p-hat700-2	18	63	357.1	492.0
	10	36	672.2	919.4
p-hat700-3	18	78	260.2	330.7
	10	44	543.1	588.6
san1000	15	89	319.1	493.2
	10	100	260	438
san200-0.7-1	18	69	26.0	53.6
	10	50	82	86
san200-0.7-2	18	84	40.4	51.1
	10	53	75.4	115.0
san200-0.9-1	18	108	-1.8	0
	10	82	18.2	14.6
san200-0.9-2	18	85	20	17.6
	10	68	25	27.9
san200-0.9-3	18	83	21.6	18.0
	10	67	34.3	26.8
san400-0.5-1	18	79	115.1	194.9
	10	58	189.6	289.6
san400-0.7-1	18	84	95.2	144.0
	10	55	138.1	209.0
san400-0.7-2	18	78	105.1	158.9
	10	42	247.6	309.5
san400-0.7-3	18	73	138.3	180.8
	10	47	225.5	287.2
san400-0.9-1	18	97	63.9	75.2
	10	75	93.3	98.6
sanr200-0.7	18	61	42.6	63.9
	10	45	80	104.4
sanr200-0.9	18	77	12.9	23.3
	10	61	31.1	37.7
sanr400-0.5	18	67	152.2	223.8
	10	32	406.2	543.7
sanr400-0.7	18	76	103.9	152.6
	10	47	231.9	270.2

Fig. 7. Experimental results on maxclique instances

obtains a percentage increment of 1566% (see instance *p-hat1500-2*). In this case, the increase ranges from 14.6% to 1566% when z is set to 10, and from 17.6% to 1292% for the highest value of z.

It is important to note that the bound obtained with MBE_h^p is always higher than that of MBE_r^p. For some instances, the percentage of increment of MBE_h^p is more than 4 times higher the one obtained with MBE_r^p (e.g. instance *c-fat200-1*). Therefore, it is clear that an adequate propagation tree impacts on the propagation phase and, as a consequence, on the bounds obtained.

6 Conclusions and Future Work

Mini-bucket elimination (MBE) is a well-known approximation algorithm for combinatorial optimization problems. It has a control parameter z which allow us to trace time and space for approximation accuracy. In practice, it is usually the space rather than the cpu time which limits the control parameter.

In this paper we introduce a new propagation phase that MBE should execute at each bucket. In the new algorithm, that we call MBE^p, the idea is to *move* costs along mini-buckets in order to accumulate as much information as possible in one of them. The propagation phase is based on a *propagation tree* where each node is a mini-bucket and edges represent movements of costs along branches from the leaves to the root. Finally, it is important to note that the propagation phase does not increase the asymptotical time and space complexity of the original MBE algorithm.

We demonstrate the effectiveness of our algorithm in *scheduling*, *combinatorial auction* and *maxclique* problems. The typical percentage of increment in the lower bound obtained is 50%. However, for almost all maxclique instances the percentage of increment ranges from 250% to a maximum of 1566%. Therefore, MBE^p is able to obtain much more accurate lower bounds than standard MBE using the same amount of resources.

In our future work we want to integrate the propagation phase into the depth-first mini-bucket elimination algorithm [9]. The two main issues are how the computation tree rearrangements affect the bucket propagation and how to efficiently deal with the functions maintaining the transferred costs.

Acknowledgement

This research has been funded with project TIN2005-09312-C03-02.

References

1. Dechter, R., Rish, I.: Mini-buckets: A general scheme for bounded inference. Journal of the ACM **50** (2003) 107–153
2. Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G.: Semiring-based CSPs and valued CSPs: Frameworks, properties and comparison. Constraints **4** (1999) 199–240

3. Pearl, J.: Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference. Morgan Kaufmann, San Mateo, CA (1988)
4. H. Xu, R.R., Sakallah, K.: sub-sat: A formulation for relaxed boolean satisfiability with applications in rounting. In: Proc. Int. Symp. on Physical Design, CA (2002)
5. D.M. Strickland, E.B., Sokol, J.: Optimal protein structure alignment using maximum cliques. Operations Research **53** (2005) 389–402
6. Vasquez, M., Hao, J.: A logic-constrained knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. Journal of Computational Optimization and Applications **20(2)** (2001)
7. Park, J.D.: Using weighted max-sat engines to solve mpe. In: Proc. of the 18^{th} AAAI, Edmonton, Alberta, Canada (2002) 682–687
8. Kask, K., Dechter, R.: A general scheme for automatic generation of search heuristics from specification dependencies. Artificial Intelligence **129** (2001) 91–131
9. Rollon, E., Larrosa, J.: Depth-first mini-bucket elimination. In: Proc. of the 11^{th} CP, Sitges (Spain), LNCS 3709. Springer-Verlag. (2005) 563–577
10. Cooper, M., Schiex, T.: Arc consistency for soft constraints. Artificial Intelligence **154** (2004) 199–227
11. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113** (1999) 41–85
12. Bertele, U., Brioschi, F.: Nonserial Dynamic Programming. Academic Press (1972)
13. Larrosa, J., Schiex, T.: Solving weighted csp by maintaining arc-consistency. Artificial Intelligence **159** (2004) 1–26
14. Cooper, M.: High-order consistency in valued constraint satisfaction. Constraints **10** (2005) 283–305
15. Bensana, E., Lemaitre, M., Verfaillie, G.: Earth observation satellite management. Constraints **4(3)** (1999) 293–299
16. K.Leuton-Brown, M., Y.Shoham: Towards a universal test suite for combinatorial auction algorithms. ACM E-Commerce (2000) 66–76
17. Johnson, D.S., Trick, M.: Second dimacs implementation challenge: cliques, coloring and satisfiability. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS **26** (1996)

Constraint Satisfaction with Bounded Treewidth Revisited

Marko Samer[1,⋆] and Stefan Szeider[2,⋆⋆]

[1] Institute of Information Systems (DBAI)
Vienna University of Technology, Austria
samer@dbai.tuwien.ac.at

[2] Department of Computer Science
University of Durham, UK
stefan.szeider@durham.ac.uk

Abstract. The constraint satisfaction problem can be solved in polynomial time for instances where certain parameters (e.g., the treewidth of primal graphs) are bounded. However, there is a trade-off between generality and performance: larger bounds on the parameters yield worse time complexities. It is desirable to pay for more generality only by a constant factor in the running time, not by a larger degree of the polynomial. Algorithms with such a uniform polynomial time complexity are known as fixed-parameter algorithms.

In this paper we determine whether or not fixed-parameter algorithms for constraint satisfaction exist, considering all possible combinations of the following parameters: the treewidth of primal graphs, the treewidth of dual graphs, the treewidth of incidence graphs, the domain size, the maximum arity of constraints, and the maximum size of overlaps of constraint scopes. The negative cases are subject to the complexity theoretic assumption FPT $\neq$ W[1] which is the parameterized analog to P $\neq$ NP. For the positive cases we provide an effective fixed-parameter algorithm which is based on dynamic programming on "nice" tree decompositions.

1 Introduction

An instance of the constraint satisfaction problem (CSP) consists of a set of variables that range over a domain of values together with a set of constraints that allow certain combinations of values for certain sets of variables. The question is whether one can instantiate the variables in such a way that all constraints are simultaneously satisfied; in that case the instance is called consistent or satisfiable. Constraint satisfaction provides a general framework which allows direct structure preserving encodings of numerous problems that arise in practice.

Although constraint satisfaction is NP-complete in general, many efforts have been made to identify restricted problems that can be solved in polynomial time. Such restrictions can either limit the constraints used in the instance [5] or limit

⋆ Research supported by the Austrian Science Fund (FWF), P17222-N04.
⋆⋆ Research supported in part by the Nuffield Foundation, NAL/01012/G.

the overall structure of the instance, i.e., how variables and constraints interact in the instance [7]. In this paper we focus on the latter form of restrictions which are also referred to as "structural restrictions." Structural restrictions are usually formulated in terms of certain graphs and hypergraphs that are associated with a constraint satisfaction instance:

The *primal graph* has the variables as its vertices; two variables are joined by an edge if they occur together in the scope of a constraint. The *dual graph* has the constraints as its vertices; two constraints are joined by an edge if their scopes have variables in common. The *incidence graph* is a bipartite graph and has both the variables and the constraints as its vertices; a variable and a constraint are joined by an edge if the variable occurs in the scope of the constraint. Finally, the *constraint hypergraph* is a hypergraph whose vertices are the variables and the scopes of constraints form the hyperedges.

Fundamental classes of tractable instances are obtained if the associated (hyper)graphs are *acyclic* with respect to certain notions of acyclicity. Acyclicity can be generalized by means of (hyper)graph decomposition techniques which give rise to "width" parameters that measure how far an instance deviates from being acyclic. Already in the late 1980s Freuder [11] and Dechter and Pearl [8] observed that constraint satisfaction is polynomial-time solvable if

– the *treewidth of primal graphs*, **tw**,

is bounded by a constant. The graph parameter treewidth, introduced by Robertson and Seymour in their Graph Minors Project, has become a very popular object of study as many NP-hard graph problems are polynomial-time solvable for graphs of bounded treewidth; we define treewidth in Section 2.2. In subsequent years several further structural parameters have been considered, such as

– the *treewidth of dual graphs*, $\mathbf{tw}^d$,
– the *treewidth of incidence graphs*, $\mathbf{tw}^*$,

and various width parameters on constraint hypergraphs, including

– the *(generalized) hypertree width*, (**g**)**hw**, (Gottlob, Leone, and Scarcello[14]),
– the *spread-cut width*, **scw**, (Cohen, Jeavons, and Gyssens [6]), and
– the *fractional hypertree width*, **fhw**, (Grohe and Marx [17]).

Considering constraint satisfaction instances where the width parameter under consideration is bounded by some fixed integer k gives rise to a class W_k of tractable instances. The larger k gets, the larger is the resulting tractable class W_k. However, for getting larger and larger tractable classes one has to pay by longer running times. A fundamental question is the *trade-off between generality and performance*. A typical time complexity of algorithms known from the literature are of the form

$$O(\|I\|^{O(f(k))}) \tag{1}$$

for instances I belonging to the class W_k; here $\|I\|$ denotes the input size of I and $f(k)$ denotes a slowly growing function. Such a running time is polynomial when k is considered as a constant. However, since k appears in the exponent, such algorithms become impractical—even if k is small—when large instances are considered. It is significantly better if instances I of the class W_k can be solved in time

$$O(f(k)\|I\|^{O(1)}) \tag{2}$$

where f is an arbitrary (possibly exponential) computable function. In that case the order of the polynomial does not depend on k, and so considering larger and larger classes does not increase the order of the polynomial. Thus, it is a significant objective to classify the trade-off between generality and performance of a width parameter under consideration: whether the parameter allows algorithms of type (1) or of type (2).

1.1 Parameterized Complexity

The framework of *parameterized complexity* provides the adequate concepts and tools for studying the above question. Parameterized complexity was initiated by Downey and Fellows in the late 1980s and has become an important branch of algorithm design and analysis; hundreds of research papers have been published in that area [9,19,10]. It turned out that the distinction between tractability of type (2) and tractability of type (1) is a robust indication of problem hardness.

A *fixed parameter algorithm* is an algorithm that achieves a running time of type (2) for instances I and parameter k. A parameterized problem is *fixed-parameter tractable* if it can be solved by a fixed-parameter algorithm. FPT denotes the class of all fixed-parameter tractable decision problems.

Parameterized complexity offers a *completeness theory*, similar to the theory of NP-completeness, that allows the accumulation of strong theoretical evidence that a parameterized problem is *not* fixed-parameter tractable. This completeness theory is based on the *weft hierarchy* of complexity classes $\mathrm{W}[1], \mathrm{W}[2], \ldots, \mathrm{W}[\mathrm{P}]$. Each class is the equivalence class of certain parameterized satisfiability problems under *fpt-reductions* (for instance, the canonical W[1]-complete problem asks whether a given 3SAT instance can be satisfied by setting at most k variables to *true*). Let Π and Π' be two parameterized problems. An *fpt-reduction R from Π to Π'* is a many-to-one transformation from Π to Π', such that (i) $(I, k) \in \Pi$ if and only if $(I', k') \in \Pi'$ with $k' \leq g(k)$ for a fixed computable function g and (ii) R is of complexity $O(f(k)\|I\|^{O(1)})$ for a computable function f. The class XP consists of parameterized problems which can be solved in polynomial time if the parameter is considered as a constant. The above classes form the chain

$$\mathrm{FPT} \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots \subseteq \mathrm{W}[\mathrm{P}] \subseteq \mathrm{XP}$$

where all inclusions are assumed to be proper. A parameterized analog of Cook's Theorem gives strong evidence to assume that $\mathrm{FPT} \neq \mathrm{W}[1]$ [9]. It is known that $\mathrm{FPT} \neq \mathrm{XP}$ [9]; hence the term "parameterized tractable" (which is sometimes used to indicate membership in XP [6]) must be carefully distinguished from "fixed-parameter tractable." Although XP contains problems which are very unlikely to be fixed-parameter tractable, it is often a significant improvement to show that a problem belongs to this class, in contrast to, e.g., k-SAT which is NP-complete for every constant $k \geq 3$.

The following parameterized clique-problem is W[1]-complete [9]; this problem is the basis for the hardness results considered in the sequel.

CLIQUE
Instance: A graph G and a non-negative integer k.
Parameter: k.
Question: Does G contain a clique on k vertices?

1.2 Parameterized Constraint Satisfaction

We consider any computable function p that assigns to a constraint satisfaction instance I a non-negative integer $p(I)$ as a *constraint satisfaction parameter.* For a finite set $\{p_1, \ldots, p_r\}$ of constraint satisfaction parameters we consider the following generic parameterized problem:

$\mathbf{CSP}(\{p_1, \ldots, p_r\})$
Instance: A constraint satisfaction instance I and non-negative integers $k_1, \ldots, k_r$ with $p_1(I) \leq k_1, \ldots, p_r(I) \leq k_r$.
Parameters: $k_1, \ldots, k_r$.
Question: Is I consistent?

Slightly abusing notation, we will also write $\mathbf{CSP}(p_1, \ldots, p_r)$ instead of $\mathbf{CSP}(\{p_1, \ldots, p_r\})$. We write $\mathbf{CSP}_{\mathbf{boole}}(S)$ to denote $\mathbf{CSP}(S)$ with the *Boolean domain* $\{0, 1\}$, and $\mathbf{CSP}_{\mathbf{bin}}(S)$ to denote $\mathbf{CSP}(S)$ where all constraints have arity at most 2.

Note that we formulate this problem as a "promise problem" in the sense that for solving the problem we do not need to verify the assumption $p_1(I) \leq k_1, \ldots, p_r(I) \leq k_r$. However, unless otherwise stated, for all cases considered in the sequel where $\mathbf{CSP}(p_1, \ldots, p_r)$ is fixed-parameter tractable, also the verification of the assumption $p_1(I) \leq k_1, \ldots, p_r(I) \leq k_r$ is fixed-parameter tractable. For a constraint satisfaction instance I we have the basic parameters

- the number of variables, **vars**,
- the size of the domain, **dom**,
- the largest size of a constraint scope, **arity**, and
- the largest number of variables that occur in the *overlap* of the scopes of two distinct constraints, **ovl**.

If we parameterize by the domain size, then we have obviously an NP-complete problem, since, e.g., 3-colorability can be expressed as a constraint satisfaction problem with constant domain. Thus $\mathbf{CSP}(\mathbf{dom})$ is not fixed-parameter tractable unless P = NP. On the other hand, if we parameterize by the number of variables *and* the domain size, i.e., $\mathbf{CSP}(\mathbf{vars}, \mathbf{dom})$, then we have a trivially fixed-parameter tractable problem: we can decide the consistency of an instance I by checking all $\mathbf{dom}(I)^{\mathbf{vars}(I)}$ possible assignments. However, without the parameter **dom** we get $\mathbf{CSP}(\mathbf{vars})$, a W[1]-complete problem [20].

Gottlob, Scarcello, and Sideri [16] have determined the parameterized complexity of constraint satisfaction with bounded treewidth of primal graphs: $\mathbf{CSP}(\mathbf{tw}, \mathbf{dom})$ is fixed-parameter tractable, and $\mathbf{CSP}(\mathbf{tw})$ is W[1]-hard. The parameterized complexity of constraint satisfaction with respect to other structural parameters like treewidth of dual graphs, treewidth of incidence graphs, and the more general width parameters defined in terms of constraint hypergraphs remained open. In this paper we determine exactly those combinations of parameters from $\mathbf{tw}$, $\mathbf{tw}^d$, $\mathbf{tw}^*$, $\mathbf{dom}$, $\mathbf{arity}$, and $\mathbf{ovl}$ that render constraint satisfaction fixed-parameter tractable.

To this end we introduce the notion of *domination*. Let S and S' be two finite sets of constraint satisfaction parameters. S *dominates* S' if for every $p \in S$ there exists a monotonously growing computable function f such that for every constraint satisfaction instance I we have $p(I) \leq f(\max_{p' \in S'}(p'(I)))$. See Lemma 2 for examples that illustrate this notion (if S or S' is a singleton, we omit the braces to improve readability). It is easy to see that whenever S dominates S', then fixed-parameter tractability of $\mathbf{CSP}(S)$ implies fixed-parameter tractability of $\mathbf{CSP}(S')$, and W[1]-hardness of $\mathbf{CSP}(S')$ implies W[1]-hardness of $\mathbf{CSP}(S)$ (see Lemma 1).

1.3 Results

We obtain the following classification result (see also the diagram in Figure 1 and the discussion in Section 3).

Theorem 1 (Classification Theorem). *Let* $S \subseteq \{\mathbf{tw}, \mathbf{tw}^d, \mathbf{tw}^*, \mathbf{dom}, \mathbf{arity}, \mathbf{ovl}\}$.

1. *If* $\{\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl}\}$ *dominates* S*, then* $\mathbf{CSP}(S)$ *is fixed-parameter tractable.*
2. *If* $\{\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl}\}$ *does not dominate* S*, then* $\mathbf{CSP}(S)$ *is not fixed-parameter tractable unless* FPT = W[1].

The fixed-parameter tractability results are established by a dynamic programming algorithm. The established upper bounds to its worst-case running time show that the algorithm is feasible in practice. Let us remark that many constraint satisfaction instances appearing in industry have bounded overlap. For example, the `Adder`, `Bridge`, and `NewSystem` instances from *DaimlerChrysler* have by construction an overlap bounded by 2 [12].

We extend the fixed-parameter tractability result of the Classification Theorem to the additional parameters **diff** and **equiv**; definitions are given in Section 6. We show that he problems $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{diff})$ and $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{equiv})$ are fixed-parameter tractable.

The notion of domination allows us to extend the W[1]-hardness results of the Classification Theorem to all parameters that are more general than the treewidth of incidence graphs. In particular, we obtain the following corollary.

Corollary 1. *The problems* $\mathbf{CSP}(p, \mathbf{dom})$ *and* $\mathbf{CSP}_{\mathbf{boole}}(p)$ *are* W[1]*-hard if* p *is any of the parameters treewidth of incidence graphs, hypertree width, generalized hypertree width, spread-cut width, and fractional hypertree width.*

Recently, Gottlob *et al.* [13] have shown that the problem of deciding whether a given hypergraph has (generalized) hypertree width at most k, is W[2]-hard with respect to the parameter k. We note that this result does not imply W[1]-hardness of $\mathbf{CSP}(\mathbf{hw}, \mathbf{dom})$ (respectively $\mathbf{CSP}(\mathbf{ghw}, \mathbf{dom})$), since it is possible to design algorithms for constraint satisfaction instances of bounded (generalized) hypertree width that avoid the decomposition step. Chen and Dalmau [4] have recently proposed such an algorithm, which, however, is not a fixed-parameter algorithm.

Our results are in contrast to the situation for the propositional satisfiability problem (SAT). A SAT instance is a set of clauses, representing a propositional formula in conjunctive normal form. The question is whether the instance is satisfiable. Primal, dual, and incidence graphs and the corresponding treewidth parameters $\mathbf{tw}$, $\mathbf{tw}^d$, and $\mathbf{tw}^*$ can be defined for SAT similarly as for constraint satisfaction [22], as well as the parameterized decision problem $\mathbf{SAT}(p)$ for a parameter p. In contrast to the $W[1]$-hardness of $\mathbf{CSP_{boole}}(\mathbf{tw}^*)$, as established in Corollary 1, the problem $\mathbf{SAT}(\mathbf{tw}^*)$ is fixed-parameter tractable. This holds also true for $\mathbf{SAT}(\mathbf{tw}^d)$ and $\mathbf{SAT}(\mathbf{tw})$ since $\mathbf{tw}^*$ dominates $\mathbf{tw}^d$ and $\mathbf{tw}$. The fixed-parameter tractability of $\mathbf{SAT}(\mathbf{tw}^*)$ was shown by Szeider [22] using a general result on model checking for Monadic Second Order (MSO) logic on graphs.

2 Preliminaries

2.1 Constraint Satisfaction

Formally, a constraint satisfaction instance I is a triple (V, D, F), where V is a finite set of *variables*, D is a finite set of *domain values*, and F is a finite set of *constraints*. Each constraint in F is a pair (S, R), where S, the *constraint scope*, is a sequence of distinct variables of V, and R, the *constraint relation*, is a relation over D whose arity matches the length of S. We write $\text{var}(C)$ for the set of variables that occur in the scope of a constraint C and $\text{rel}(C)$ for the relation of C. An *assignment* is a mapping $\tau : X \to D$ defined on some set X of variables. Let $C = ((x_1, \ldots, x_n), R)$ be a constraint and $\tau : X \to D$ an assignment. We define

$$C[\tau] = \{\, (d_1, \ldots, d_n) \in R : x_i \notin X \text{ or } \tau(x_i) = d_i,\ 1 \le i \le n \,\}.$$

Similarly, for a set $\mathcal{T}$ of assignments and a constraint C we define

$$C[\mathcal{T}] = \bigcup_{\tau \in \mathcal{T}} C[\tau].$$

An assignment $\tau : X \to D$ is *consistent* with a constraint C if $C[\tau] \neq \emptyset$. An assignment $\tau : X \to D$ *satisfies* a constraint C if $\text{var}(C) \subseteq X$ and τ is consistent with C. An assignment satisfies a constraint satisfaction instance I if it satisfies all constraints of I. The instance I is *consistent* (or *satisfiable*) if it is satisfied by some assignment. The *constraint satisfaction problem* **CSP** is the problem of deciding whether a given constraint satisfaction instance is satisfiable.

2.2 Tree Decompositions

Let G be a graph, let T be a tree, and let χ be a labelling of the vertices of T by sets of vertices of G. We refer to the vertices of T as "nodes" to avoid confusion with the vertices of G, and we call the sets $\chi(t)$ "bags." The pair (T, χ) is a *tree decomposition* of G if the following three conditions hold:

1. For every vertex v of G there exists a node t of T such that $v \in \chi(t)$.
2. For every edge vw of G there exists a node t of T such that $v, w \in \chi(t)$.
3. For any three nodes t_1, t_2, t_3 of T, if t_2 lies on the unique path from t_1 to t_3, then $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$ ("Connectedness Condition").

The *width* of a tree decomposition (T, χ) is defined as the maximum $|\chi(t)| - 1$ over all nodes t of T. The *treewidth* $\mathrm{tw}(G)$ of a graph G is the minimum width over all its tree decompositions.

As shown by Bodlaender [1], there exists for every k a linear time algorithm that checks whether a given graph has treewidth at most k and, if so, outputs a tree decomposition of minimum width. Bodlaender's algorithm does not seem feasible to implement [3]. However, there are several other known fixed-parameter algorithms that are feasible. For example, Reed's algorithm [21] runs in time $O(|V| \log |V|)$ and decides either that the treewidth of a given graph $G = (V, E)$ exceeds k, or outputs a tree decomposition of width at most $4k$, for any fixed k. The algorithm produces tree decompositions with $O(|V|)$ many nodes.

Let (T, χ) be a tree decomposition of a graph G and let r be a node of T. The triple (T, χ, r) is a *nice tree decomposition* of G if the following three conditions hold; here we consider $T = (V(T), E(T))$ as a tree rooted at r.

1. Every node of T has at most two children.
2. If a node t of T has two children t_1 and t_2, then $\chi(t) = \chi(t_1) = \chi(t_2)$; in that case we call t a *join node.*
3. If a node t of T has exactly one child t', then exactly one of the following prevails:
   (a) $|\chi(t)| = |\chi(t')| + 1$ and $\chi(t') \subset \chi(t)$; in that case we call t an *introduce node.*
   (b) $|\chi(t)| = |\chi(t')| - 1$ and $\chi(t) \subset \chi(t')$; in that case we call t a *forget node.*

Let (T, χ, r) be a nice tree decomposition of a graph G. For each node t of T let T_t denote the subtree of T rooted at t. Furthermore, let V_t denote the set of vertices of G that occur in the bags of nodes of T_t; i.e., $V_t = \bigcup_{t' \in V(T_t)} \chi(t')$.

It is well known (and easy to see) that for any constant k, given a tree decomposition of a graph $G = (V, E)$ of width k and with $O(|V|)$ nodes, there exists a linear-time algorithm that constructs a nice tree decomposition of G with $O(|V|)$ nodes and width at most k [3].

3 The Domination Lattice

The next lemma follows directly from the definitions of fpt-reductions and domination.

Lemma 1. *Let S and S' be two sets of constraint satisfaction parameters such that S dominates S'. Then there is an fpt-reduction from* $\mathbf{CSP}(S')$ *to* $\mathbf{CSP}(S)$.

In particular, fixed-parameter tractability of $\mathbf{CSP}(S)$ *implies fixed-parameter tractability of* $\mathbf{CSP}(S')$*, and* W[1]*-hardness of* $\mathbf{CSP}(S')$ *implies* W[1]*-hardness of* $\mathbf{CSP}(S)$.

Lemma 2

1. *If $S \subseteq S'$, then S dominates S'.*
2. **ovl** *dominates* **arity**.
3. **arity** *dominates* **tw**.
4. $\mathbf{tw}^*$ *dominates* **tw**.
5. $\mathbf{tw}^*$ *dominates* $\mathbf{tw}^d$.
6. **tw** *dominates* $\{\mathbf{tw}^*, \mathbf{arity}\}$.

Proof. Parts 1 and 2 are obvious. Part 3 is also easy to see since a constraint of arity r yields a clique on r vertices in the primal graph; it is well known that if a graph G contains a clique with r vertices, then $\mathrm{tw}(G) \geq r - 1$ [2]. Part 4 follows from the inequality $\mathbf{tw}^*(I) \leq \mathbf{tw}(I) + 1$ shown by Kolaitis and Vardi [18]. A symmetric argument gives $\mathbf{tw}^*(I) \leq \mathbf{tw}^d(I) + 1$, hence Part 5 holds as well. Part 6 follows by the inequality $\mathbf{tw}(I) \leq \mathbf{tw}^*(I)(\mathbf{arity}(I) - 1)$ which is also due to Kolaitis and Vardi [18]. □

We note that parts 2–5 of the above lemma are *strict* in the sense that p dominates q but q does not dominated p.

Let $\mathcal{S} = \{\mathbf{tw}, \mathbf{tw}^d, \mathbf{tw}^*, \mathbf{dom}, \mathbf{arity}, \mathbf{ovl}\}$. The following arguments will make it easier to classify $\mathbf{CSP}(S)$ for all subsets S of $\mathcal{S}$.

First we note that whenever $S \cap \{\mathbf{tw}, \mathbf{tw}^d, \mathbf{tw}^*\} = \emptyset$, then S dominates $\{\mathbf{dom}, \mathbf{arity}, \mathbf{ovl}\}$. However, $\mathbf{CSP}(\mathbf{dom}, \mathbf{arity}, \mathbf{ovl})$ is not fixed-parameter tractable unless P = NP, since graph 3-colorability can be expressed as a constraint satisfaction problem with constant **dom**, **arity**, and **ovl**.

Second, if a set S dominates a proper subset S' of S, then $\mathbf{CSP}(S)$ and $\mathbf{CSP}(S')$ are of the same parameterized complexity; this follows by Lemmas 1 and 2(1); in this case we can disregard S. For example, we can disregard the set $\{\mathbf{tw}, \mathbf{arity}\}$ since it dominates $\{\mathbf{tw}\}$ by Lemma 2(3). Similarly we can disregard the set $\{\mathbf{tw}, \mathbf{ovl}\}$ since it dominates $\{\mathbf{tw}, \mathbf{arity}\}$ by Lemma 2(2) and so it dominates $\{\mathbf{tw}\}$ as well.

Third, by Lemma 2(4 and 6), every set $S \cup \{\mathbf{tw}^*, \mathbf{arity}\}$ has the same parameterized complexity as the set $S \cup \{\mathbf{tw}\}$.

Hence it suffices to classify the parameterized complexity of $\mathbf{CSP}(S)$ for the following twelve sets $S \subseteq \mathcal{S}$.

$$\begin{array}{llll}
\{\mathbf{tw}\}, & \{\mathbf{tw}, \mathbf{dom}\}, & \{\mathbf{tw}^d\}, & \{\mathbf{tw}^d, \mathbf{dom}\}, \\
\{\mathbf{tw}^*\}, & \{\mathbf{tw}^*, \mathbf{dom}\}, & \{\mathbf{tw}^d, \mathbf{ovl}\}, & \{\mathbf{tw}^d, \mathbf{dom}, \mathbf{ovl}\}, \\
\{\mathbf{tw}^*, \mathbf{ovl}\}, & \{\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl}\}, & \{\mathbf{tw}^d, \mathbf{arity}\}, & \{\mathbf{tw}^d, \mathbf{dom}, \mathbf{arity}\}.
\end{array}$$

Figure 1 shows the relationships among all twelve sets as implied by Lemma 2: a set S dominates a set S' if and only if S is above S' and there is a path from

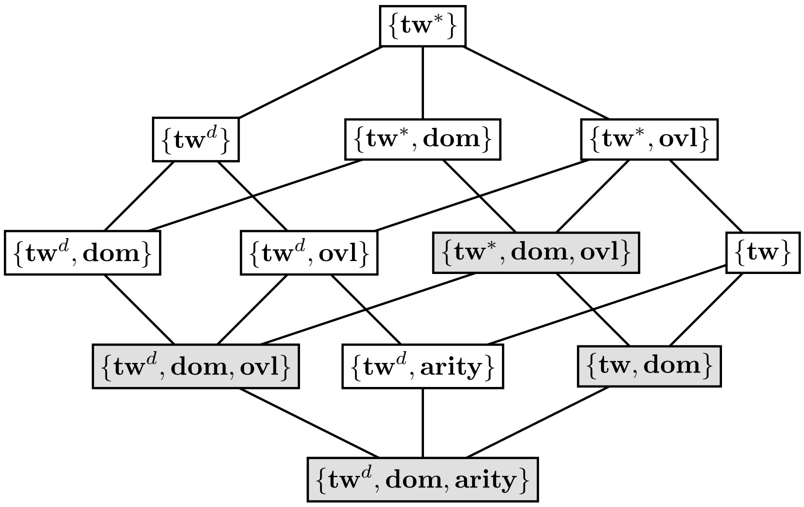


Fig. 1. Domination lattice

S to S'. The sets S for which $\mathbf{CSP}(S)$ is fixed-parameter tractable according to the Classification Theorem are indicated in the diagram by shaded boxes. In view of the relationships between the sets, the Classification Theorem is established if we show (i) W[1]-hardness of $\mathbf{CSP}(\mathbf{tw}^d, \mathbf{dom})$, (ii) W[1]-hardness of $\mathbf{CSP}(\mathbf{tw}^d, \mathbf{arity})$, and (iii) fixed-parameter tractability of $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl})$.

4 Proof of the W[1]-Hardness Results

We give an fpt-reduction from **CLIQUE** to $\mathbf{CSP_{boole}}(\mathbf{tw}^d)$. To this aim, consider an instance of **CLIQUE**, i.e., a graph $G = (\{v_1, \dots, v_n\}, E)$ and an integer k. We construct a Boolean constraint satisfaction instance $I = (\{\, x_{i,j} : 1 \leq i \leq k, 1 \leq j \leq n\,\}, \{0,1\}, F)$ such that I is consistent if and only if there exists a clique of size k in G.

We construct the relation $R \subseteq \{0,1\}^{2n}$ that encodes the edges of G using Boolean values 0 and 1 as follows: For each edge $v_p v_q$ of G, $1 \leq p < q \leq n$, we add to R the $2n$-tuple

$$(t_{p,1}, \dots, t_{p,n}, t_{q,1}, \dots, t_{q,n})$$

where $t_{p,i} = 1$ if and only if $p = i$, and $t_{q,i} = 1$ if and only if $q = i$, $1 \leq i \leq n$. We let F be the set of constraints

$$C_{i,j} = ((x_{i,1}, \dots, x_{i,n}, x_{j,1}, \dots, x_{j,n}), R)$$

for $1 \leq i < j \leq k$. It is easy to verify that G contains a clique on k vertices if and only if I is consistent. The reduction can be carried out in polynomial time.

Next we construct a trivial tree decomposition of the dual graph of I by creating a single tree node and putting all constraints in its bag. Since

$|F| = \binom{k}{2}$, the width of our decomposition is $\binom{k}{2} - 1$. Hence, $\mathbf{tw}^d(I) \leq \binom{k}{2} - 1$. Thus we have indeed an fpt-reduction from **CLIQUE** to $\mathbf{CSP_{boole}}(\mathbf{tw}^d)$ and $\mathbf{CSP}(\mathbf{tw}^d, \mathbf{dom})$. Consequently, the latter problems are W[1]-hard. Since $\mathbf{tw}^*$ dominates $\mathbf{tw}^d$ (Lemma 2(5)), also $\mathbf{CSP_{boole}}(\mathbf{tw}^*)$ and $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom})$ are W[1] hard (Lemma 1).

In turn, it is well known that the treewidth of incidence graphs is dominated by each of the parameters (generalized) hypertree width, fractional hypertree width, and spread-cut width of constraint hypergraphs [6,15,17,18]. Hence, Corollary 1 follows by Lemma 1.

The W[1]-hardness of $\mathbf{CSP}(\mathbf{tw}^d, \mathbf{arity})$ can be shown by an fpt-reduction from **CLIQUE** as well. This reduction, which is easier than the reduction above, was used by Papadimitriou and Yannakakis for showing W[1]-hardness of $\mathbf{CSP}(\mathbf{vars})$: Given a graph $G = (V, E)$ and an integer k, we construct a CSP instance $I = (\{x_1, \dots, x_k\}, V, F)$ where F contains constraints $((x_i, x_j), E)$ for all $1 \leq i < j \leq k$. Evidently, G contains a clique of size k if and only if I is consistent. Since there are $\binom{k}{2}$ constraints, the dual graph of I has a trivial tree decomposition of width $\binom{k}{2} - 1$. Thus $\mathbf{tw}^d(I) \leq \binom{k}{2} - 1$ and $\mathbf{arity}(I) = 2$. Whence $\mathbf{CSP_{bin}}(\mathbf{tw}^d, \mathbf{arity})$ and $\mathbf{CSP}(\mathbf{tw}^d, \mathbf{arity})$ are W[1]-hard.

For establishing the Classification Theorem it remains to show that $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl})$ is fixed-parameter tractable.

5 Fixed-Parameter Algorithm for $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl})$

For this section, let (T, χ, r) be a nice tree decomposition of width k of the incidence graph of a constraint satisfaction instance $I = (V, D, F)$.

For each node t of T, let F_t denote the set of all the constraints in V_t, and let X_t denote the set of all variables in V_t; that is, $F_t = V_t \cap F$ and $X_t = V_t \cap V$. We also use the shorthands $\chi_c(t) = \chi(t) \cap F$ and $\chi_v(t) = \chi(t) \cap V$ for the set of variables and the set of constraints in $\chi(t)$, respectively. Moreover, $\chi_v^*(t)$ denotes the set of all variables of X_t that are in $\chi_v(t)$ or in $\operatorname{var}(C_1) \cap \operatorname{var}(C_2)$ for two distinct constraints $C_1, C_2 \in \chi_c(t)$. That is, $\chi_v^*(t)$ is the set of variables in $\chi_v(t)$ together with all "forgotten" variables in X_t that occur in at least two constraints in $\chi_c(t)$.

Let t be a node of T and let $\alpha : \chi_v^*(t) \to D$ be an assignment. We define $N(t, \alpha)$ as the set of assignments $\tau : X_t \to D$ such that $\tau|_{\chi_v^*(t)} = \alpha$ and τ is consistent with all constraints in F_t. Consequently, I is consistent if and only if $N(r, \alpha) \neq \emptyset$ for some $\alpha : \chi_v^*(r) \to D$. The following lemmas show that we can decide consistency of I by dynamic programming along a bottom-up traversal of T.

Lemma 3. *Let t be a* join node *of T with children t_1, t_2. Let $\alpha : \chi_v^*(t) \to D$ be an assignment and $\alpha_i = \alpha|_{\chi_v^*(t_i)}$, $i = 1, 2$. Then the following holds:*

1. *$N(t, \alpha) \neq \emptyset$ if and only if $N(t_1, \alpha_1) \neq \emptyset$, $N(t_2, \alpha_2) \neq \emptyset$, and $C[N(t_1, \alpha_1)] \cap C[N(t_2, \alpha_2)] \neq \emptyset$ for all $C \in \chi_c(t)$.*
2. *If $N(t, \alpha) \neq \emptyset$, then $C[N(t, \alpha)] = C[N(t_1, \alpha_1)] \cap C[N(t_2, \alpha_2)]$ for all $C \in \chi_c(t)$.*

Proof. (1) For the *only if* direction, let $\tau \in N(t, \alpha)$, and $\tau_1 = \tau|_{X_{t_1}}$ and $\tau_2 = \tau|_{X_{t_2}}$. It is then easy to verify that $\tau_1 \in N(t_1, \alpha_1)$ and $\tau_2 \in N(t_2, \alpha_2)$. Moreover, it holds that $C[\tau] \neq \emptyset$ for all $C \in \chi_c(t)$, which implies $C[\tau_1] \cap C[\tau_2] \neq \emptyset$ for all $C \in \chi_c(t)$. Hence, we have $C[N(t_1, \alpha_1)] \cap C[N(t_2, \alpha_2)] \neq \emptyset$ for all $C \in \chi_c(t)$. For the *if* direction, let $\tau_1 \in N(t_1, \alpha_1)$ and $\tau_2 \in N(t_2, \alpha_2)$ such that $C[\tau_1] \cap C[\tau_2] \neq \emptyset$ for all $C \in \chi_c(t)$. Now, let us define the assignment $\tau : X_t \to D$ by $\tau|_{X_{t_1}} = \tau_1$ and $\tau|_{X_{t_2}} = \tau_2$. To verify that $\tau \in N(t, \alpha)$, note that, by the Connectedness Condition, we have $V_{t_1} \cap V_{t_2} = \chi(t)$, that is, $X_{t_1} \cap X_{t_2} = \chi_v(t) \subseteq \chi_v^*(t)$ and $F_{t_1} \cap F_{t_2} = \chi_c(t)$. Moreover, it holds that $\chi_v^*(t_1) \cup \chi_v^*(t_2) = (\chi_v^*(t) \cap X_{t_1}) \cup (\chi_v^*(t) \cap X_{t_2}) = \chi_v^*(t) \cap (X_{t_1} \cup X_{t_2}) = \chi_v^*(t) \cap X_t = \chi_v^*(t)$. (2) follows immediately from the above constructions. □

Lemma 4. *Let t be an* introduce node *with child t' where $\chi(t) = \chi(t') \cup \{x\}$ for a variable x. Let $\alpha : \chi_v^*(t') \to D$ be an assignment and $\beta = \alpha \cup \{(x, d)\}$ for some domain element $d \in D$. Then the following holds:*

1. *$N(t, \beta) \neq \emptyset$ if and only if $N(t', \alpha) \neq \emptyset$ and $C[N(t', \alpha)] \cap C[\{(x, d)\}] \neq \emptyset$ for all $C \in \chi_c(t)$.*
2. *If $N(t, \beta) \neq \emptyset$, then $C[N(t, \beta)] = C[N(t', \alpha)] \cap C[\{(x, d)\}]$ for all $C \in \chi_c(t)$.*

Proof. (1) For the *only if* direction, let $\tau \in N(t, \beta)$ and $\tau' = \tau|_{X_{t'}}$. Thus, it follows that $\tau' \in N(t', \alpha)$. Moreover, it holds that $C[\tau] \neq \emptyset$ for all $C \in \chi_c(t)$, which implies $C[\tau'] \cap C[\{(x, d)\}] \neq \emptyset$ for all $C \in \chi_c(t)$. Hence, we have $C[N(t', \alpha)] \cap C[\{(x, d)\}] \neq \emptyset$ for all $C \in \chi_c(t)$. For the *if* direction, let $\tau' \in N(t', \alpha)$ such that $C[\tau'] \cap C[\{(x, d)\}] \neq \emptyset$ for all $C \in \chi_c(t)$. Now, let us define the assignment $\tau : X_t \to D$ by $\tau|_{X_{t'}} = \tau'$ and $\tau(x) = d$ for the single variable $x \in X_t \setminus X_{t'}$. It is then easy to show that $\tau \in N(t, \beta)$. (2) follows immediately from the above constructions. □

Lemma 5. *Let t be an* introduce node *with child t' where $\chi(t) = \chi(t') \cup \{B\}$ for a constraint B. Let $\alpha : \chi_v^*(t) \to D$ be an assignment. Then the following holds:*

1. *$N(t, \alpha) \neq \emptyset$ if and only if $N(t', \alpha) \neq \emptyset$ and $B[\alpha] \neq \emptyset$.*
2. *If $N(t, \alpha) \neq \emptyset$, then $C[N(t, \alpha)] = C[N(t', \alpha)]$ for all $C \in \chi_c(t')$.*

Proof. (1) For the *only if* direction, let $\tau \in N(t, \alpha)$ and $\tau' = \tau|_{X_{t'}}$. So we have $\tau' \in N(t', \alpha)$. Moreover, since $C[\tau] \neq \emptyset$ for all $C \in \chi_c(t)$, we know that $B[\tau] \neq \emptyset$. Thus, since $\tau|_{\chi_v^*(t)} = \alpha$, we obtain $B[\alpha] \neq \emptyset$. For the *if* direction, let $\tau' \in N(t', \alpha)$ and $B[\alpha] \neq \emptyset$. By the construction of a tree decomposition of an incidence graph, we know that $\mathrm{var}(B) \cap X_t \subseteq \chi_v(t) \subseteq \chi_v^*(t)$. Thus, since $\tau'|_{\chi_v^*(t)} = \alpha$, we have $B[\tau'] \neq \emptyset$. So it can be easily verified that $\tau' \in N(t, \alpha)$. (2) follows immediately from the above constructions. □

Lemma 6. *Let t be a* forget node *with child t' where $\chi(t) = \chi(t') \setminus \{x\}$ for a variable x. Let $\alpha : \chi_v^*(t) \to D$ be an assignment. If $x \in \chi_v^*(t)$, then the following holds:*

1. *$N(t, \alpha) \neq \emptyset$ if and only if $N(t', \alpha) \neq \emptyset$.*
2. *If $N(t, \alpha) \neq \emptyset$, then $C[N(t, \alpha)] = C[N(t', \alpha)]$ for all $C \in \chi_c(t)$.*

Otherwise, if $x \notin \chi_v^*(t)$, then the following holds:

1. *$N(t, \alpha) \neq \emptyset$ if and only if $N(t', \alpha \cup \{(x, d)\}) \neq \emptyset$ for some $d \in D$.*
2. *If $N(t, \alpha) \neq \emptyset$, then $C[N(t, \alpha)] = \bigcup_{d \in D} C[N(t', \alpha \cup \{(x, d)\})]$ for all $C \in \chi_c(t)$.*

Proof. The case of $x \in \chi_v^*(t)$ is trivial, since $\chi_v^*(t) = \chi_v^*(t')$ and $\chi_c(t) = \chi_c(t')$. Let us therefore consider the more interesting case of $x \notin \chi_v^*(t)$. (1) For the *only if* direction, let $\tau \in N(t, \alpha)$. Thus, since $X_t = X_{t'}$, we know that $\tau(x) = d$ for the single variable $x \in \chi_v^*(t') \setminus \chi_v^*(t)$ and some domain element $d \in D$. Consequently, $\tau \in N(t', \alpha \cup \{(x, d)\})$ for some $d \in D$. For the *if* direction, let $\tau' \in N(t', \alpha \cup \{(x, d)\})$ for some $d \in D$. Then we trivially have $\tau' \in N(t, \alpha)$. (2) follows immediately from the above constructions. □

Lemma 7. *Let t be a* forget node *with child t' where $\chi(t) = \chi(t') \setminus \{B\}$ for a constraint B. Let $\alpha : \chi_v^*(t) \to D$ be an assignment. Then the following holds:*

1. *$N(t, \alpha) \neq \emptyset$ if and only if $N(t', \alpha') \neq \emptyset$ for some $\alpha' : \chi_v^*(t') \to D$ s.t. $\alpha = \alpha'|_{\chi_v^*(t)}$.*
2. *If $N(t, \alpha) \neq \emptyset$, then $C[N(t, \alpha)] = \bigcup_{\alpha = \alpha'|_{\chi_v^*(t)}} C[N(t', \alpha')]$ for all $C \in \chi_c(t)$.*

Proof. (1) For the *only if* direction, let $\tau \in N(t, \alpha)$. Thus, since $X_t = X_{t'}$, we know that for all variables $x \in \chi_v^*(t') \setminus \chi_v^*(t)$ there exists some domain element $d \in D$ such that $\tau(x) = d$. Consequently, $\tau \in N(t', \alpha')$ for some $\alpha' : \chi_v^*(t') \to D$ such that $\alpha = \alpha'|_{\chi_v^*(t)}$. For the *if* direction, let $\tau' \in N(t', \alpha')$ for some $\alpha' : \chi_v^*(t') \to D$ such that $\alpha = \alpha'|_{\chi_v^*(t)}$. Then we trivially have $\tau' \in N(t, \alpha)$. (2) follows immediately from the above constructions. □

Lemma 8. *Let t be a* leaf node *and $\alpha : \chi_v^*(t) \to D$ be an assignment. Then the following holds:*

1. *$N(t, \alpha) \neq \emptyset$ if and only if $C[\alpha] \neq \emptyset$ for all $C \in \chi_c(t)$.*
2. *If $N(t, \alpha) \neq \emptyset$, then $C[N(t, \alpha)] = C[\alpha]$ for all $C \in \chi_c(t)$.*

Proof. Since $X_t = \chi_v(t) = \chi_v^*(t)$ and $F_t = \chi_c(t)$ for every leaf node t, we immediately obtain the above properties. □

In the following, we represent the sets $C[N(t, \alpha)]$ for each $\alpha : \chi_v^*(t) \to D$ and $C \in \chi_c(t)$ by a table M_t with at most $|\chi_v^*(t)| + |\chi_c(t)|m$ columns and $|D|^{|\chi_v^*(t)|}$ rows, where $m = \max_{C \in F} |\mathrm{rel}(C)|$. The first $|\chi_v^*(t)|$ columns of M_t contain values from D encoding $\alpha(x)$ for variables $x \in \chi_v^*(t)$. The further columns of M_t represent the tuples in $\mathrm{rel}(C)$ for $C \in \chi_c(t)$ and contain Boolean values. The proof of the next lemma is straightforward and omitted due to the space restrictions.

Lemma 9. *Let t be a node of T. Given the tables of the children of t, we can compute the table M_t in time $O(d^p pks)$, where $p = |\chi_v^*(t)|$, $d = |D|$, and s is the size of a largest constraint relation of constraints of I.*

Theorem 2. *Given a constraint satisfaction instance I together with a nice tree decomposition (T, χ) of the incidence graph of I. Let d be the size of the domain of I and let s be the size of a largest constraint relation of constraints of I. Furthermore, let k be the width and n the number of nodes of (T, χ), and let p denote the maximum $|\chi_v^*(t)|$ over all nodes t of T. Then we can decide in time $O(d^p pksn)$ whether I is consistent.*

Proof. We compute the tables M_t for all nodes t of T in a bottom up ordering, starting from the leaf nodes. By Lemma 9, each table can be computed in time $O(d^p pks)$. Since I is consistent if and only if M_r is nonempty, the theorem follows. □

Corollary 2. $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl})$ *is fixed-parameter tractable.*

Proof. Let I be a constraint satisfaction instance whose incidence graph has treewidth at most k and $c = \mathbf{ovl}(I)$. Recall from Section 2.2 that we can find in linear time a nice tree decomposition of the incidence graph of G of width at most k. Now the corollary follows immediately from Theorem 2 and the fact that $|\chi_v^*(t)| \le k + ck^2$ holds for all nodes t of T. □

Corollary 2 provides the last step for the proof of the Classification Theorem.

6 Fixed-Parameter Tractability for Further Parameters

In this section, we describe two further constraint satisfaction parameters for which Theorem 2 applies.

Let $I = (V, D, F)$ be a constraint satisfaction instance. For a subset F' of F we define $\delta_I(F')$ as the set of variables that occur in the scopes of all constraints of F' but in no scope of constraints of $F \setminus F'$; i.e., $\delta_I(F') = (\bigcap_{C \in F'} \text{var}(C)) \setminus (\bigcup_{C \in F \setminus F'} \text{var}(C))$. We define $\mathbf{equiv}(I)$ as the maximum size of $\delta_I(F')$ over all subsets $F' \subseteq F$ that contain at least two constraints. Furthermore, we define $\mathbf{diff}(I)$ as the maximum size of $\text{var}(C_1) \setminus \text{var}(C_2)$ over all pairs of constraints $C_1, C_2 \in F$.

Corollary 3. $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{equiv})$ *is fixed-parameter tractable.*

Proof. Let I be a constraint satisfaction instance whose incidence graph has treewidth at most k. We compute in linear time a nice tree decomposition (T, χ, r) of the incidence graph of I of width at most k. Let $q = \mathbf{equiv}(I)$. Evidently, we have $|\chi_v^*(t)| \le k + q2^k$ for all nodes t of T. Hence the corollary follows immediately from Theorem 2. □

Note that the running time of our fixed-parameter algorithm for $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{ovl})$ is significantly smaller than the running time for $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{equiv})$. However, there exist instances with bounded **equiv** and arbitrarily large **ovl** (i.e., **equiv** dominates **ovl**, but not *vice versa*). For example, let us construct an instance in the following way: We start with any constraint C_0 and add in each step a new constraint C_n and a new variable x_n such that $\bigcap_{0\leq i\leq n}\mathrm{var}(C_i) = \{x_n\}$. By construction $\mathbf{equiv}(I) = 1$, but $\mathbf{ovl}(I) \geq n$ since $|C_0 \cap C_1| = n$.

Corollary 4. $\mathbf{CSP}(\mathbf{tw}^*, \mathbf{dom}, \mathbf{diff})$ *is fixed-parameter tractable.*

Proof. Again, let $I = (V, D, F)$ be a constraint satisfaction instance whose incidence graph has treewidth at most k. Let $q' = \mathbf{diff}(I)$ and let $d = |D|$. We compute in linear time a nice tree decomposition (T, χ, r) of the incidence graph of I of width at most k and n nodes. Next we obtain from I a solution-equivalent constraint satisfaction instance I' by computing the join of all relations of constraints in $\chi_c(t)$ for each node t of T. Note that the result of a join operation of two relations over D having a size at most s can be of size at most $sd^{q'}$ under our restriction. Thus, the join of at most k relations can be computed in time $O(s^2 d^{q'k})$ and the size of the largest relation of I' is bounded by $sd^{q'k}$. Moreover, note that the tree decomposition of the incidence graph of I gives rise to a tree decomposition of the incidence graph of I'; for the latter we have $\chi_v^*(t) = \chi_v(t)$; that is, $|\chi_v^*(t)| \leq k$, for all nodes t of T. Hence, in view of Theorem 2, we obtain a running time of $O(nd^{(q'+1)k}k^2s^2)$ for the dynamic programming algorithm. □

Since $\mathbf{tw}^*$ dominates $\mathbf{tw}^d$, we obtain from these corollaries that the problems $\mathbf{CSP}(\mathbf{tw}^d, \mathbf{dom}, \mathbf{equiv})$ and $\mathbf{CSP}(\mathbf{tw}^d, \mathbf{dom}, \mathbf{diff})$ are fixed-parameter tractable.

7 Conclusion

We have presented a general framework for studying the trade-off between generality and performance for parameterized constraint satisfaction problems. Within our framework we have classified the parameterized complexity of combinations of natural parameters including the treewidth of primal, dual, and incidence graphs, the domain size, and the size of overlaps of constraint scopes. The parameterized complexity of further parameters and their combinations remain open for future research. Furthermore, it would be interesting to extend the hardness results of this paper to completeness results for classes of the weft hierarchy.

Acknowledgement. We thank Moshe Vardi for suggesting us to include the treewidth of dual graphs into our considerations.

References

1. H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.
2. H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoret. Comput. Sci.*, 209(1-2):1–45, 1998.
3. H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*.

4. H. Chen and V. Dalmau. Beyond hypertree width: Decomposition methods without decompositions. In *Proc. CP'05*, LNCS, vol. 3709, 167–181. Springer, 2005.
5. D. Cohen and P. Jeavons. The complexity of constraint languages. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, part I, chapter 8. Elsevier, 2006 (forthcoming).
6. D. Cohen, P. Jeavons, and M. Gyssens. A unified theory of structural tractability for constraint satisfaction and spread cut decomposition. In *Proc. IJCAI'05*, 72–77, 2005.
7. R. Dechter. Tractable structures for constraint satisfaction problems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, part I, chapter 7. Elsevier, 2006 (forthcoming).
8. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.
9. R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer, 1999.
10. J. Flum and M. Grohe. *Parameterized Complexity Theory.* Springer, 2006.
11. E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.
12. T. Ganzow, G. Gottlob, N. Musliu, and M. Samer. A CSP hypergraph library. Technical Report DBAI-TR-2005-50, Database and Artificial Intelligence Group, Vienna University of Technology, 2005.
13. G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In D. Kratsch, editor, *Proc. WG'05*, LNCS, vol. 3787, 1–15. Springer, 2005.
14. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions: a survey. In *Proc. MFCS'01*, LNCS, vol. 2136, 37–57. Springer, 2001.
15. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. of Computer and System Sciences*, 64(3):579–627, 2002.
16. G. Gottlob, F. Scarcello, and M. Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1-2):55–86, 2002.
17. M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *Proc. SODA'06*, 289–298, ACM, 2006.
18. P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. of Computer and System Sciences*, 61(2):302–332, 2000.
19. R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.
20. C. H. Papadimitriou and M. Yannakakis. On the complexity of database queries. *J. of Computer and System Sciences*, 58(3):407–427, 1999.
21. B. Reed. Finding approximate separators and computing tree width quickly. In *Proc. STOC'92*, 221–228. ACM, 1992.
22. S. Szeider. On fixed-parameter tractable parameterizations of SAT. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability, 6th International Conference, SAT 2003, Selected and Revised Papers*, LNCS, vol. 2919, 188–202. Springer, 2004.

Preprocessing QBF

Horst Samulowitz, Jessica Davies, and Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada
{horst, jdavies, fbacchus}@cs.toronto.edu

Abstract. In this paper we investigate the use of preprocessing when solving Quantified Boolean Formulas (QBF). Many different problems can be efficiently encoded as QBF instances, and there has been a great deal of recent interest and progress in solving such instances efficiently. Ideas from QBF have also started to migrate to CSP with the exploration of Quantified CSPs which offer an intriguing increase in representational power over traditional CSPs. Here we show that QBF instances can be simplified using techniques related to those used for preprocessing SAT. These simplifications can be performed in polynomial time, and are used to preprocess the instance prior to invoking a worst case exponential algorithm to solve it. We develop a method for preprocessing QBF instances that is empirically very effective. That is, the preprocessed formulas can be solved significantly faster, even when we account for the time required to perform the preprocessing. Our method significantly improves the efficiency of a range of state-of-the-art QBF solvers. Furthermore, our method is able to completely solve some instances just by preprocessing, including some instances that to our knowledge have never been solved before by any QBF solver.

1 Introduction

QBF is a powerful generalization of SAT in which the variables can be universally or existentially quantified (in SAT all variables are implicitly existentially quantified). While any NP problem can be encoded in SAT, QBF allows us to encode any PSPACE problem: QBF is PSPACE-complete. This increase in representational power also holds for finite domain CSPs, with quantified CSPs being able to represent PSPACE-complete problems not expressible as standard CSP problems [12,20].

This opens a much wider range of potential application areas for a QBF or QCSP solver, including problems from areas like automated planning (particularly conditional planning), non-monotonic reasoning, electronic design automation, scheduling, model checking and verification, see, e.g., [11,16]. However, the difficulty is that QBF and QCSP are in practice much harder problems to solve.

Current QBF solvers are typically limited to problems that are about 1-2 orders of magnitude smaller than the instances solvable by current SAT solvers (1000's of variables rather than 100,000's). A similar difference holds between current QCSP solvers and CSP solvers. Nevertheless, both QBF and QCSP solvers continue to improve. Furthermore, many problems have a much more compact encoding when quantifiers are available, so a quantified solver can still be useful even if it can only deal with much smaller instances than a traditional solver.

F. Benhamou (Ed.): CP 2006, LNCS 4204, pp. 514–529, 2006.
© Springer-Verlag Berlin Heidelberg 2006

In this paper we present a new technique for improving QBF solvers. Like many techniques used for QBF, ours is a modification of techniques already used in SAT. Namely we preprocess the input formula, without changing its meaning, so that it becomes easier to solve. As we demonstrate below our technique can be extremely effective, sometimes reducing the time it takes to solve a QBF instance by orders of magnitude. Although our technique is not immediately applicable to QCSP, it does provide insights into preprocessing that in future work could have a positive impact on the efficiency of QCSP solvers. We discuss some of the connections to QCSP in our future work section.

In the sequel we first present some necessary background, setting the context for our methods. We then present further details of our approach and state some results about its correctness. Then we provide empirical evidence of the effectiveness of our approach, and close with a discussion of future work and some conclusions.

2 QBF

A quantified boolean formula has the form $\boldsymbol{Q}.F$, where F is a propositional formula expressed in CNF and $\boldsymbol{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that no variable appear twice in $\boldsymbol{Q}$ and that the set of variables in F and $\boldsymbol{Q}$ be identical (i.e., F contains no free variables, and $\boldsymbol{Q}$ contains no extra or redundant variables).

A **quantifier block** qb of $\boldsymbol{Q}$ is a maximal contiguous subsequence of $\boldsymbol{Q}$ where every variable in qb has the same quantifier type. We order the quantifier blocks by their sequence of appearance in $\boldsymbol{Q}$: $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in $\boldsymbol{Q}$. Each variable x in F appears in some quantifier block $qb(x)$ and we say that $x \leq_q y$ if $qb(x) \leq qb(y)$ and $x <_q y$ if $qb(x) < qb(y)$. We also say that x is **universal** (**existential**) if its quantifier in $\boldsymbol{Q}$ is $\forall$ ($\exists$).

For example, $\exists e_1 e_2.\forall u_1 u_2.\exists e_3 e_4.(e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$ is a QBF with $\boldsymbol{Q} = \exists e_1 e_2.\forall u_1 u_2.\exists e_3 e_4$ and F equal to the two clauses $(e_1, \neg e_2, u_2, e_4)$ and $(\neg u_1, \neg e_3)$. The quantifier blocks in order are $\exists e_1 e_2$, $\forall u_1 u_2$, and $\exists e_3 e_4$, and we have, e.g., that, $e_1 <_q e_3$, $u_1 <_q e_4$, u_1 is universal, and e_4 is existential.

A QBF instance can be reduced by assigning values to some of its variables. The **reduction** of a formula $\boldsymbol{Q}.F$ by a literal ℓ (denoted by $\boldsymbol{Q}.F\big|_\ell$) is the new formula $\boldsymbol{Q}'.F'$ where F' is F with all clauses containing ℓ removed and the negation of ℓ, $\neg\ell$, removed from all remaining clauses, and $\boldsymbol{Q}'$ is $\boldsymbol{Q}$ with the variable of ℓ and its quantifier removed. For example, $\big(\forall xz.\exists y.(\neg y, x, z) \wedge (\neg x, y)\big)\big|_{\neg x} = \forall z.\exists y(\neg y, z)$.

Semantics. A SAT-model $\mathcal{M}_s$ of a CNF formula F is a truth assignment π to the variables of F that satisfies every clause in F. In contrast a QBF model (**QBF-model**) $\mathcal{M}_q$ of a quantified formula $\boldsymbol{Q}.F$ is a **tree** of truth assignments in which the root is the empty truth assignment, and every node n assigns a truth value to a variable of F not yet assigned by one of n's ancestors. The tree $\mathcal{M}_q$ is subject to the following conditions:

1. For every node n in $\mathcal{M}_q$, n has a sibling if and only if it assigns a truth value to a universal variable x. In this case it has exactly one sibling that assigns the opposite truth value to x. Nodes assigning existentials have no siblings.
2. Every path π in $\mathcal{M}_q$ (π is the sequence of truth assignments made from the root to a leaf of $\mathcal{M}_q$) must assign the variables in an order that respects $\leq_q$. That is, if n assigns x and one of n's ancestors assigns y then we must have that $y \leq_q x$.

3. Every path π in $\mathcal{M}_q$ must be a SAT-model of F. That is π must satisfy the body of $\boldsymbol{Q}.F$.

Thus a QBF-model has a path for every possible setting of the universal variables of $\boldsymbol{Q}$, and each of these paths is a SAT-model of F. We say that a QBF $\boldsymbol{Q}.F$ is QSAT iff it has a QBF-model, and that it is UNQSAT otherwise. The QBF problem is to determine whether or not $\boldsymbol{Q}.F$ is QSAT.

A more standard way of defining QSAT is the recursive definition: (1) $\forall x \boldsymbol{Q}.F$ is QSAT iff both $\boldsymbol{Q}.F|_x$ and $\boldsymbol{Q}.F|_{\neg x}$ are QSAT, and (2) $\exists x \boldsymbol{Q}.F$ is QSAT iff at least one of $\boldsymbol{Q}.F|_x$ and $\boldsymbol{Q}.F|_{\neg x}$ is QSAT. By removing the quantified variables one by one we arrive at either a QBF with an empty clause in its body F (which is not QSAT) or a QBF with an empty body F (which is QSAT). It is not difficult to prove by induction on the length of the quantifier sequence that the definition we provided above is equivalent to this definition.

The advantage of our "tree-of-models" definition is that it makes two key observations more apparent. These observations can be used to prove the correctness of our preprocessing technique.

A. If F' has the same SAT-models as F then $\boldsymbol{Q}.F$ will have the same QBF-models as $\boldsymbol{Q}.F'$.
Proof: $\mathcal{M}_q$ is a QBF-model of $\boldsymbol{Q}.F$ iff each path in $\mathcal{M}_q$ is a SAT-model of F iff each path is a SAT-model of F' iff $\mathcal{M}_q$ is a QBF-model of $\boldsymbol{Q}.F'$.
This observation allows us to transform F with any model preserving SAT transformation. Note that the transformation must be model preserving, i.e., it must preserve all SAT-models of F. Simply preserving whether or not F is satisfiable is not sufficient.

B. A QBF-model preserving (but not SAT-model preserving) transformation that can be performed on $\boldsymbol{Q}.F$ is **universal reduction**. A universal variable u is called a *tailing universal* in a clause c if for every existential variable $e \in c$ we have that $e <_q u$. The universal reduction [9] of a clause c is the process of removing all tailing universals from c. The universal reduction of a QBF formula $\boldsymbol{Q}.F$ is the process of applying universal reduction to all of the clauses of F. Universal reduction preserves the set of QBF-models.
Proof: Let $\boldsymbol{Q}.F'$ be the universal reduction of $\boldsymbol{Q}.F$.
$\mathcal{M}_q \models \boldsymbol{Q}.F \rightarrow \mathcal{M}_q \models \boldsymbol{Q}.F'$: Say that $v \in c$ is a tailing universal, then along any path π in any QBF-Model $\mathcal{M}_q$ of $\boldsymbol{Q}.F$, c must be satisfied by π prior to v being assigned a value. Say not, then since v is universal, the prefix of π that leads to the assignment of v must also be the prefix of another path π' that sets v to false: but then π' will falsify c because at this point c is a unit clause containing only the universal variable v. Therefore $\mathcal{M}_q$ cannot be a QBF-model of $\boldsymbol{Q}.F$. Hence every path π satisfies the universal reduction of c (and all other clauses in F), and thus $\mathcal{M}_q$ is also QBF-model of $\boldsymbol{Q}.F'$ where F' is F with the tailing universal v removed from c. This process can be repeated to remove all tailing universals from all clauses of F.
$\mathcal{M}_q \models \boldsymbol{Q}.F' \rightarrow \mathcal{M}_q \models \boldsymbol{Q}.F$: Since the clauses of F are longer (less restrictive) than those of F' every QBF-Model $\mathcal{M}_q$ of $\boldsymbol{Q}.F'$ must be also a QBF-Model of $\boldsymbol{Q}.F$.

We call two QBF formulas **Q-equivalent** iff they have exactly the same QBF-Models.

3 HyperBinary Resolution for SAT

The foundation of our polynomial time preprocessing technique is the SAT method of reasoning with binary clauses using hyper-resolution developed in [2,3]. This method reasons with CNF SAT theories using the following “HypBinRes” rule of inference:

> Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l) | l \in D\}$, infer the new clause $b = (c - D) \cup \{\ell\}$ if b is either binary or unary.

For example, from (a, b, c, d), $(h, \neg a)$, $(h, \neg c)$ and $(h, \neg d)$, we infer the new binary clause (h, b), similarly from (a, b, c) and $(b, \neg a)$ the rule generates (b, c). The HypBinRes rule covers the standard case of resolving two binary clauses (from (l_1, l_2) and $(\neg l_1, \ell)$ infer (ℓ, l_2)) and it can generate unit clauses (e.g., from (l_1, ℓ) and $(\neg l_1, \ell)$ we infer $(\ell, \ell) \equiv (\ell)$).

The advantage of HypBinRes inference is that it does not blow up the theory (it can only add binary or unary clauses to the theory) and it can discover a lot of new unit clauses. These unit clauses can then be used to simplify the formula by doing unit propagation which in turn might allow more applications of HypBinRes. Applying HypBinRes and unit propagation until closure (i.e., until nothing new can be inferred) uncovers *all* failed literals. That is, in the resulting reduced theory there will be no literal ℓ such that forcing ℓ to be true followed by unit propagation results in a contradiction. This and other results about HypBinRes are proved in the above references.

In addition to uncovering unit clauses we can use the binary clauses to perform equality reductions. In particular, if we have two clauses $(\neg x, y)$ and $(x, \neg y)$ we can replace all instances of y in the formula by x (and $\neg y$ by $\neg x$). This might result in some tautological clauses which can be removed, and some clauses which are reduced in length because of duplicate literals. This reduction might yield new binary or unary clauses which can then enable further HypBinRes inferences. Taken together HypBinRes and equality reduction can significantly reduce a SAT formula removing many of its variables and clauses [3].

4 Preprocessing QBF

Given a QBF $\boldsymbol{Q}.F$ we could apply HypBinRes, unit propagation, and equality reduction to F until closure. This would yield a new formula F', and the QBF $\boldsymbol{Q}'.F'$ where $\boldsymbol{Q}'$ is $\boldsymbol{Q}$ with all variables not in F' removed. Unfortunately, there are two problems with this approach. One is that the new QBF $\boldsymbol{Q}'.F'$ might not be Q-equivalent to $\boldsymbol{Q}.F$, so that this method of preprocessing is not sound. The other problem is that we miss out on some important additional inferences that can be achieved through universal reduction. We elaborate on these two issues and show how they can be overcome.

The reason why the straightforward application of HypBinRes, unit propagation and equality reduction to the body of a QBF is unsound, is that the resulting formula F' does not have exactly the same SAT models as F, as is required by condition $\mathbf{A}$ above. In particular, the models of F' do not make assignments to variables that have been removed by unit propagation and equality reduction. Hence, a QBF-model of $\boldsymbol{Q}'.F'$

might not be extendable to a QBF-model of $\boldsymbol{Q}.F$. For example, if unit propagation forced a universal variable in F, then $\boldsymbol{Q}'.F'$ might be QSAT, but $\boldsymbol{Q}.F$ is not (no QBF-model of $\boldsymbol{Q}.F$ can exist since the paths that set the forced universal to its opposite value will not be SAT-models of F). This situation occurs in the following example. Consider the QBF $\boldsymbol{Q}.F = \exists abc \forall x \exists yz (x, \neg y)(x, z)(\neg z, y)(a, b, c)$. We can see that $\boldsymbol{Q}.F$ is not QSAT since when x is false, $\neg y$ and z must be true, falsifying the clause $(\neg z, y)$. If we apply HypBinRes and unit propagation to F, we obtain $F' = (a, b, c)$, where the universal variable x has been unit propagated away. As anticipated, $\boldsymbol{Q}'.F' = \exists abc(a, b, c)$ is QSAT, so this reduction of F has not preserved the QSAT status of the original formula. This, however, is an easy problem to fix. Making unit propagation sound for QBF simply requires that we regard the unit propagation of a universal variable as equivalent to the derivation of the empty clause (i.e. false). This fact is well known and applied in all search-based QBF solvers.

Ensuring that equality reduction is sound for QBF is a bit more subtle. Consider a formula F in which we have the two clauses $(x, \neg y)$ and $(\neg x, y)$. Since every path in any QBF-model satisfies F, this means that along any path x and y must have the same truth value. However, in order to soundly replace all instances of one of these variables by the other in F, we must respect the quantifier ordering. In particular, if $x <_q y$ then we must replace y by x. It would be unsound to do the replacement in the other direction. For example, say that x appears in quantifier block 3 while y appears in quantifier block 5 with both x and y being existentially quantified. The above binary clauses will enforce the constraint that along any path of any QBF-model once x is assigned y must get the same value. In particular, y will be invariant as we change the assignments to the universal variables in quantifier block 4. This constraint will continue to hold if we replace y by x in all of the clauses of F. However, if we perform the opposite replacement, we would be able to make y vary as we vary the assignments to the universal variables of quantifier block 4: i.e., the opposite replacement would weaken the theory perhaps changing its QSAT status. The same reasoning holds if x is universal and y is existential. However, if y is universal, the two binary clauses imply that we will never have the freedom to assign y its two different truth values. That is, in this case the QBF is UNQSAT, and we can again treat this case as if the empty clause has been derived.

Therefore a sound version of equality reduction must respect the variable ordering. We call this ($<_q$ preferred) equality reduction. That is, if we detect that x and y are equivalent and $x <_q y$ then we always remove y from the theory replacing it by x. With this restriction on equality reduction we have the following result:

Proposition 1. *Let F' be the result of applying HypBinRes, unit propagation, and ($<_q$ preferred) equality reduction to F until closure. If F' has the same set of universal variables as F (i.e., no universal variable was removed by unit propagation or equality reduction), then the QBF-models of $\boldsymbol{Q}'.F'$ are in 1-1 correspondence with the QBF-models of $\boldsymbol{Q}.F$. In particular, $\boldsymbol{Q}.F$ is QSAT iff $\boldsymbol{Q}'.F'$ is QSAT. On the other hand, if F' has fewer universal variables than F then $\boldsymbol{Q}.F$ is UNQSAT.*

The idea behind the proof is that we can map any SAT-model of F' to a SAT-model of F by assigning all forced variables their forced value, and assigning all equality reduced variables a value derived from the variable they are equivalent to. That is, if

x was removed because it was equivalent to $\neg y$ (y), we assign x the opposite (same) value assigned to y in F''s SAT-model. In the other direction any SAT-model of F can be mapped to a SAT-model of F' by simply omitting the assignments of variables not in F'. Given this relationship between the SAT-models, we can then show that any QBF-model of F can be transformed to a unique QBF-model of F' (by splicing out the nodes that assign variables not in F') and vice versa (by splicing in nodes to assign the variables not in F'). This proves that the number of QBF-models of each formula are equal and that the transformation must be a 1-1 mapping.

This proposition tells us that we can use a SAT based reduction of F as a way of preprocessing a QBF, as long as we ensure that equality reduction respects the quantifier ordering and check for the removal of universals. This approach, however, does not fully utilize the power of universal reduction (condition **B** above). So instead we use a more powerful approach that is based on the following modification of HypBinRes that "folds" universal reduction into the inference rule. We call this rule "HypBinRes+UR":

> Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l) | l \in D\}$, infer the universal reduction of the clause $(c \setminus D) \cup \{\ell\}$ if this reduction is either binary or unary.

For example, from $c = (u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \neg e_7)$, $(e_2, \neg e_5)$ and $(e_2, \neg e_3)$ we infer the new binary clause (u_1, e_2) when $u_1 \leq_q e_2 \leq_q e_3 \leq_q u_4 \leq_q e_5 \leq_q u_6 \leq_q e_7$. Note that without universal reduction, HypBinRes would need 5 binary clauses in order to reduce c, while with universal reduction, 2 fewer binary clauses are required. This example also shows that HypBinRes+UR is able to derive clauses that HypBinRes cannot. Since clearly HypBinRes+UR can derive anything HypBinRes can, HypBinRes+UR is a more powerful rule of inference.

In addition to using universal reduction inside of HypBinRes we must also use it when unit propagation is used. For example, from the two clauses $(e_1, u_2, u_3, u_4, \neg e_5)$ and (e_5) (with $e_1 <_q u_i$) unit propagation by itself can only derive (e_1, u_2, u_3, u_4), but unit propagation with universal reduction can derive (e_1).

It turns out that in addition to gaining more inferential power, universal reduction also allows us to obtain the unconditionally sound preprocessing we would like to have.

Proposition 2. *Let F' be the result of applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to F until closure, where we always apply universal reduction before unit propagation. Then the QBF-models of $\boldsymbol{Q}'.F'$ are in 1-1 correspondence with the QBF-models of $\boldsymbol{Q}.F$.*

This result can be proved by showing that universal reduction generates the empty clause whenever a universal variable is to be unit propagated or removed via equality reduction. For example, for a universal u to be forced it must first appear in a unit clause (u), but then universal reduction would generate the empty clause (given that we apply universal reduction before unit propagation). Similarly, to make a universal variable u equivalent to an existential variable e with $e \leq_q u$ we would first have to generate the two binary clauses $(e, \neg u)$ and $(\neg e, u)$ which after universal reduction would yield (e) and $(\neg e)$ which after unit propagation would yield the empty clause. Thus the cases where Proposition 1 fails to preserve QBF-models are directly detected through the generation of an UNQSAT $\boldsymbol{Q}'.F'$. In this case we still preserve the QBF-models—neither formula has any.

Proposition 3. *Applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to $\boldsymbol{Q}.F$ until we reach closure can be done in time polynomial in the size of F.*

This result can be proved by making three observations: (1) F can never become larger than $|F|^2$ since we are only adding binary clauses, (2) there are at most a polynomial number of rule applications possible before closure since each rule either reduces a clause, removes a variable or adds a binary clause, and (3) at each stage detecting if another rule can be applied requires only time polynomial in the current size of the theory.

Our QBF preprocessor modifies $\boldsymbol{Q}.F$ exactly as described in Proposition 2. It applies HypBinRes+UR, unit propagation, universal reduction, and ($<_q$ preferred) equality reduction to F until it reaches closure. It then outputs the new formula $\boldsymbol{Q}'.F'$. Proposition 2 shows that this modification of the formula is sound. In particular, this preprocessing does not change the QSAT status of the formula.

To implement the preprocessor we adapted the algorithm presented in [3] which exploits a close connection between HypBinRes and unit propagation. In particular, this algorithm uses trial unit propagations to detect new HypBinRes inferences. The main changes required to make this algorithm work for QBF were adding universal reduction, modifying the unit propagator so that it performs universal reduction prior to any unit propagation step, and modifying equality reduction to ensure it respects the quantifier ordering.

To understand how trial unit propagation is used to detect HypBinRes+UR inferences, consider the example above of inferring (u_1, e_2) from $(u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \neg e_7)$, $(e_2, \neg e_5)$ and $(e_2, \neg e_3)$. If we perform a trial unit propagation of $\neg e_2$, dynamically performing universal reduction we obtain the unit clause (u_1). Because the trial propagation started with $\neg e_2$ this unit clause actually corresponds to the binary clause (u_1, e_2) (i.e., $\neg e_2 \rightarrow u_1$). The trial unit propagation has to keep track of the "root" of the propagation so that it does not erroneously apply universal reduction (every clause reduced during this process implicitly contains e_2).

5 Empirical Results

We implemented the described approach in the preprocessor **Prequel** [19]. To evaluate its performance we considered all of the non-random benchmark instances from QBFLib(2005) [13] (508 instances in total). We discarded the instances from the benchmark families von Neumann and Z since these are all very quickly solved by any state of the art QBF solver (less than 10 sec. for the entire suite of instances). We also discarded the instances coming from the benchmark families Jmc, and Jmc-squaring. None of these instances (with or without preprocessing) can be solved within our time bounds by any of the QBF solvers we tested. This left us with 468 remaining instances from 19 different benchmark families. We tested our approach on all of these instances.

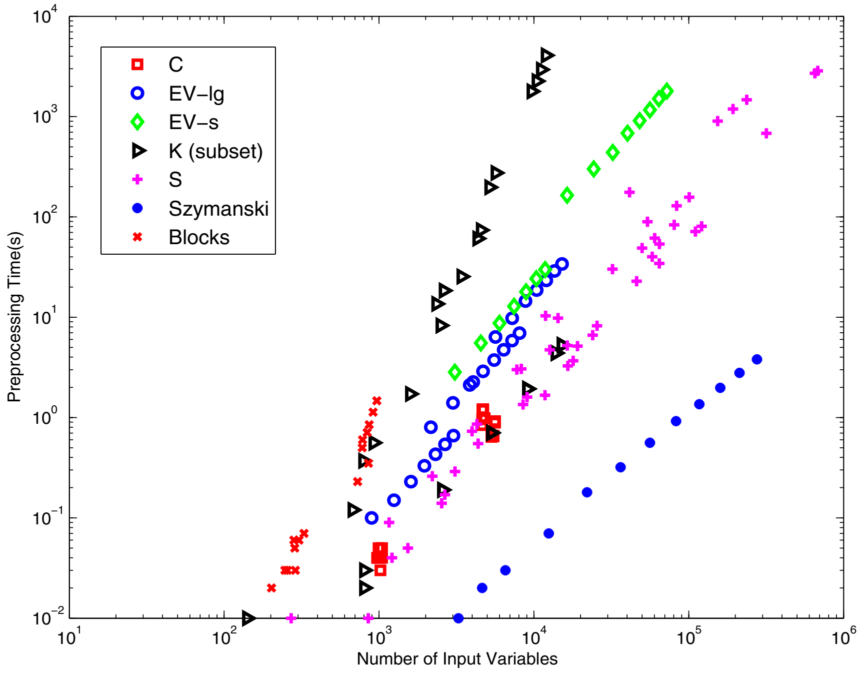


Fig. 1. Logarithmic scale comparison between the number of input variables and the preprocessing time in seconds on a selected set of benchmark families

All tests were run on a Pentium 4 3.60GHz CPU with 6GB of memory. The time limit for each run of any of the solvers or the preprocessor was set to 5,000 seconds.

5.1 Performance of the Preprocessor Prequel

We first examine the time required to preprocess the QBF formulas by looking at the runtime behaviour of Prequel on the given set of benchmark families. On the vast majority of benchmarks the preprocessing time is negligible. In particular, the preprocessing time for even the largest instances in the benchmarks Adder, Chain, Connect, Counter, FlipFlop, Lut, Mutex, Qshifter, Toilet, Tree, and Uclid is less than one second. For example, the instance Adder-16-s with $\approx$ 22,000 variables and $\approx$ 25,000 clauses is preprocessed in 0.3 seconds.

The benchmarks that require more effort to preprocess are C, EVPursade, S, Szymanski, and Blocks and a subset of the K benchmark:[1] k-branch-n, k-branch-p, k-lin-n, k-ph-n, and k-ph-p. To examine the runtime behaviour on the these benchmark families we plot the number of input variables of each instance against the time required for preprocessing (Figure 1), clustering all of the K-subfamilies into one group. Both axis of the plot are drawn in logarithmic scale.

Figure 1 shows that for all of these harder benchmarks the relationship between the number of input variables and preprocessing time is approximately linear on the

[1] This benchmark family is divided into sub-families.

Table 1. Summary of results reported in Tables 2 and 3. For each solver we show its number of solved instances among all tested benchmark families with and without preprocessing, the total CPU time (in seconds) required to solve the preprocessed and unpreprocessed instances taken over the "common" instances (instances solved in both preprocessed and unpreprocessed form), and the total CPU time required by the solvers to solve the "new" instances (instances that can only be solved in preprocessed form).

Solver	*Skizzo*		*Quantor*		*Quaffle*		*Qube*		*SQBF*	
	no-pre	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*
# Instances	311	**351**	262	**312**	226	**238**	213	**243**	205	**239**
Time on common instances	9,748	**9,595**	10,384	**2,244**	36,382	**20,188**	41,107	**23,196**	46,147	**25,554**
Time on new instances	-	12,756	-	16,829	-	9,579	-	9,707	-	2,421

loglog-plot. This is not surprising since Proposition 3 showed that the preprocessor runs in worst-case polynomial time: any polynomial function is linear in a loglog scale with the slope increasing with the degree of the polynomial. Fitting a linear function to each benchmark family enables a more detailed estimate of the runtime, since the slope of the fitted linear function determines the relationship between the number of input variables and preprocessing time. For instance, a slope of one indicates a linear runtime, a slope of two indicates quadratic behaviour, etc. Except for the benchmarks 'k-ph-n' and 'k-ph-p' the slope of the fitted linear function ranges between 1.3 (Szymanski) and 2.3 (Blocks) which indicates a linear to quadratic behaviour of the preprocessor. The two K-subfamilies 'k-ph-n' and 'k-ph-p' display worse behaviour, on them preprocessing time is almost cubic (slope of 2.9).

The graph also shows that on some of the larger problems the preprocessor can take thousands of seconds. However, this is not a practical limitation. In particular out of the 468 instances only 23 took more than 100 seconds to preprocess. Of these 18 could not be solved by any of our solvers, either in preprocessed form or unpreprocessed form. That is, preprocessing might well be cost effective on these 18 problems, but they are so hard that we have no way of evaluating this. Of the other 5 instances, which were solved by some solver, there were a total of 25 instance-solver solving attempts. Thirteen of these attempts resulted in success where a solver succeeded on either the preprocessed instance only or on the preprocessed and unpreprocessed instances (it was never the case that a solver failed on the preprocessed instance while succeeding on its unpreprocessed form). Among these 13 successful attempts, 38% were cases where only the preprocessed instance could be solved, an additional 24% were cases where the preprocessor yielded a net speedup, and only 38% were cases where the preprocessor yielded a net slowdown. So our conclusion is that except for a few instances, preprocessing is not a significant added computational burden.

5.2 Impact of Preprocessing

Now we examine how effective Prequel is. Is it able to improve the performance of state of the art QBF solvers, even when we consider the time it takes to run? To answer this question we studied the effect preprocessing has on the performance of five state of the art QBF solvers **Quaffle** [21] (version as of Feb. 2005), **Quantor** [8] (version as of 2004), **Qube** (release 1.3) [14], **Skizzo** (v0.82, r355) [4] and **SQBF** [17]. Quaffle, Qube and SQBF are based on search, whereas Quantor is based on variable elimination. Skizzo uses mainly a combination of variable elimination and search, but it also applies a variety of other kinds of reasoning on the symbolic and the ground representations of the instances.

A summary of our results is presented in Table 1. The second row of the table shows the total time required by each solver to solve the instances that could be solved in both preprocessed and unpreprocessed form (the "common instances"). The data demonstrates that preprocessing provides a speedup for every solver. Note that the times for the preprocessed instances *include* the time taken by the preprocessor. On these common instances Quantor was 4.6 times faster with preprocessing, while Quaffle, Qube and SQBF were all approximately 1.8 times faster with preprocessing. Skizzo is only slightly faster on the preprocessed benchmarks (that it could already solve). The first row of Table 1 shows the number of instances that can be solved within the 5000 sec. time bound. It demonstrates that in addition to speeding up the solvers on problems they can already solve, preprocessing also extends the reach of each solver, allowing it to solve problems that it could not solve before (within our time and memory bounds). In particular, the first row shows that the number of solved instances for each solver is significantly larger when preprocessing is applied. The increase in the number of solved instances is 13% for Skizzo, 19% for Quantor, 5% for Quaffle, 14% for Qube and 17% for SQBF. The time required by the solvers on these new instances is shown in row 3. For example, we see that SQBF was able to solve 34 new instances. None of these instances could previously be solved in 5,000 sec. each. That is, 170,000 CPU seconds were expended in 34 failed attempts. With preprocessing all of these instances could be solved in 2,421 sec. Similarly, Quantor expended 250,000 sec. in 50 failed attempts, which with preprocessing could all solved in 16,829 sec. Skizzo expended 200,000 sec. in 40 failed attempts which with preprocessing could all be solved in 12,756 seconds. Quaffle expended 60,000 sec. in 12 failed attempts, which with preprocessing could all be solved in 9,579 sec. And Qube expended 150,000 sec. in 30 failed attempts, which with preprocessing could all be solved in 9,707 seconds.

These results demonstrate quite convincingly that our preprocessor technique offers robust improvements to all of these different solvers, even though some of them are utilizing completely different solving techniques.

Tables 2 and 3 provide a more detailed breakdown of the data. Table 2 gives a family by family breakdown of the common instances (instances that can be solved in both preprocessed and unpreprocessed form). Specifically, the table shows for each benchmark family and solver (a) the percentage of instances that are solvable in both preprocessed and unpreprocessed form, (b) the total time required by the solvable instances when no preprocessing is used, and (c) the total time required with preprocessing (i.e., solving as well as preprocessing time).

Table 2 shows that the benefit of preprocessing varies among the benchmark families and, to a lesser extent, among the solvers. Nevertheless, the data demonstrates that among these benchmarks, preprocessing almost never causes a significant increase in the total time required to solve a set of instances. On the other hand, each solver has at least 2 benchmark families in which preprocessing yields more than an order of

Table 2. Benchmark family specific information about commonly solved instances. Shown are the percentage of instances that are solved in both preprocessed and unpreprocessed form and the total time in CPU seconds taken to solve these instances within each family with and without preprocessing. Best times shown in **bold**.

Benchmark (# instances)	*Skizzo*			*Quantor*			*Quaffle*			*Qube*			*SQBF*		
	Succ. %	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*
ADDER (16)	50%	954	**792**	25%	**24**	25	25%	1	1	13%	72	**27**	13%	3	**1**
adder (16)	44%	**455**	550	25%	29	**27**	42%	5	**4**	44%	**0**	1	38%	2,678	**2,229**
Blocks (16)	56%	108	**11**	100%	308	**79**	75%	1,284	**762**	69%	1774	**242**	75%	7,042	**1,486**
C (24)	25%	**1,070**	1,272	21%	140	**32**	21%	5,356	**14**	8%	3	**5**	17%	4	**0**
Chain (12)	100%	1	**0**	100%	0	0	67%	6,075	**0**	83%	4,990	**0**	58%	4,192	**0**
***Connect** (60)*	68%	802	**5**	67%	14	**7**	70%	253	**5**	75%	7,013	**7**	67%	**0**	5
Counter (24)	54%	1,036	**731**	50%	217	**141**	38%	5	5	33%	2	**1**	38	**9**	20
EVPursade (38)	29%	**1,450**	1,765	3%	**73**	82	26%	1,962	**1,960**	18%	4,402	**2,537**	32%	4,759	**4,508**
FlipFlop (10)	100%	6	**4**	100%	**3**	4	100%	**0**	4	100%	**1**	4	80%	5,027	**1**
K (107)	88%	**1,972**	2,228	63%	3,839	**39**	35%	21,675	**17,083**	37%	21,801	**19,203**	33%	5,563	**5,197**
Lut (5)	100%	9	9	100%	3	3	100%	1	1	100%	**3**	6	100%	1,247	**66**
Mutex (7)	100%	**0**	102	43%	**0**	1	29%	**43**	49	43%	**64**	71	43%	**1**	6
Qshifter (6)	100%	**8**	9	100%	**26**	29	17%	0	0	33%	29	29	33	**1,107**	2,103
S (52)	27%	**644**	1,886	25%	**910**	1,530	2%	0	0	4%	**401**	451	2%	0	0
Szymanski (12)	42%	1,147	**179**	25%	7	0	0%	0	0	8%	**0**	200	0%	0	0
TOILET (8)	100%	**1**	25	100%	4,135	**3**	75%	**61**	84	63%	496	**325**	100%	1,307	**621**
toilet (38)	100%	84	**50**	100%	684	**243**	97%	**115**	207	100%	**58**	90	97%	**395**	3,060
Tree (14)	100%	0	0	100%	0	0	100%	37	**9**	100%	**0**	1	93%	**1,051**	1,251

magnitude improvement in solving time. There are only two cases (Skizzo on Mutex, SQBF on the toilet benchmark) where preprocessing causes a slowdown that is as much as an order of magnitude (from 0 to 102 seconds and from 395 to 3,060 seconds).

Table 3 provides more information about the instances that were solvable only after preprocessing. In particular, it shows the percentage of each benchmark family that can be solved by each solver before and after preprocessing (for those families where this percentage changes). From this table we can see that for each solver there exist benchmark families where preprocessing increases the number of instances that can be solved. It is interesting to note that preprocessing improves different solvers on different families. That is, the effect of preprocessing is solver-specific. Nevertheless, preprocessing allows every solver to solve more instances. It can also be noted that the different solvers have distinct coverage, with or without preprocessing. That is, even when a solver is solving a larger percentage of a benchmark it can still be the case that it is failing to solve particular instances that are solved by another solver with a much lower success percentage on that benchmark. Preprocessing does not eliminate this variability.

Some instances are actually solved by the preprocessor itself. There are two benchmark families that are completely solved by preprocessing: FlipFlop and Connect. While the first family is rather easy to solve the second one is considered to be hard.

Table 3. Benchmark families where preprocessing changes the percentage of solved instances (within our 5,000 sec. time bound). The table shows the percentage of each families' instances that can be solved with and without preprocessing.

Benchmark	*Skizzo*		*Quantor*		*Quaffle*		*Qube*		*SQBF*	
	no-pre	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*
Blocks	69%	**88%**	100%	100%	75%	**88%**	69%	69%	75%	**81%**
C	25%	**29%**	21%	**30%**	21%	**25%**	8%	**21%**	17%	**25%**
Chain	100%	100%	100%	100%	67%	**100%**	83%	**100%**	58%	**100%**
Connect	68%	**100%**	67%	**100%**	70%	**100%**	75%	**100%**	58%	**100%**
FlipFlop	100%	100%	100%	100%	100%	100%	100%	100%	80%	**100%**
K	89%	**91%**	63%	**83%**	35%	**36%**	37%	**42%**	33%	**35%**
S	27%	**37%**	25%	**31%**	2%	**8%**	4%	**8%**	2%	**8%**
Szymanski	42%	**75%**	25%	**50%**	0%	0%	8%	**25%**	**8%**	0%
toilet	100%	100%	100%	100%	97%	**100%**	100%	100%	97%	97%
Uclid	0%	**67%**	0%	0%	0%	0%	0%	0%	0%	0%

In fact, $\approx 25\%$ of the Connect benchmarks could not be solved by any QBF solver in the 2005 QBF evaluation [15]. Our preprocessor solves the complete benchmark family in less than 10 seconds. In addition, a few benchmarks from the hard S benchmark family can be solved by the preprocessor. Again these instances could not be solved by any of the QBF solvers we tested within our time bounds. In total, the preprocessor can completely solve 18 instances that were unsolvable by any of the solvers we tested (in our time bounds). The Chain benchmark is another interesting case (its instances have 2 quantifier alternations $\exists\forall\exists$). The instances in this family are reduced to ordinary SAT instances by preprocessing. The preprocessor was able to eliminate all existential variables from the innermost quantifier block and consequently remove all universals by universal reduction. The resulting SAT instance is trivial to solve (it is smaller than the original QBF instance). In all of these cases the extended reasoning applied in the preprocessor exploits the structure of the instances very effectively. Note that the preprocessing cannot blow up the body of the QBF since it can only add binary clauses to the body. Thus, any time the preprocessor converts a QBF instance to a SAT instance, the SAT instance cannot be much larger that the original QBF.

There were only five cases where for a particular solver preprocessing changed a solvable instance to be unsolvable (Quaffle on one instance in the K benchmarks, SQBF on one instance in the Szymanski benchmarks, Skizzo on two instances in the Blocks benchmark and on one instance in the K benchmark). This is not apparent from Table 3 since both Quaffle and Skizzo can still solve more instances of the K and Blocks benchmarks respectively, with preprocessing than without. However, we can see that the percentage of solved instances for SQBF on the Szymanski benchmark falls to 0% after preprocessing. This simply represents the fact that SQBF can solve one instance of Szymanski before preprocessing and none after. That is, we have found very few cases when preprocessing is detrimental.

In total, these results indicate that preprocessing is very effective for each of the tested solvers across almost all of the benchmark families.

6 Related Work

In this section we review the similarities between our preprocessor and the methods applied in existing QBF solvers. We conclude that HypBinRes has not been previously lifted to the QBF setting, although equality reduction and binary clause reasoning have been used in some state-of-the-art QBF solvers. Our experimental results support this, since our preprocessor aids the performance and reach of even the solvers that employ binary clause reasoning and equality reduction.

Skizzo applies equality reduction as part of its symbolic reasoning phase [4]. [4] makes the claim that Skizzo's SHBR rule performs a symbolic version of hyper binary resolution. However, a close reading of the papers [4,6,5,7] suggests that in fact the SHBR rule is a strictly weaker form of inference than HypBinRes. SHBR traverses the binary implication graph of the theory, where each binary clause (x, y) corresponds to an edge $\neg x \rightarrow y$ in the graph. It detects when there is a path from a literal l to its negation $\neg l$, and in this case, unit propagates $\neg l$. This process will not achieve HypBinRes. Consider the following example where HypBinRes is applied to the theory $\{(a, b, c, d), (x, \neg a), (x, \neg b), (x, \neg c)\}$. HypBinRes is able to infer the binary clause (x, d). Yet the binary implication graph does not contain any path from a literal to its negation, so Skizzo's method will not infer any new clauses. In fact, the process of searching the implication graph is well known to be equivalent to ordinary resolution over binary clauses [1]. On the other hand, HypBinRes can infer anything that SHBR is able to since it captures binary clause resolution as a special case. Therefore SHBR is strictly weaker than HypBinRes. This conclusion is also supported by our experimental results, which show, e.g., that our preprocessor is able to completely solve the Connect Benchmark where as Skizzo is only able to solve 68% of these instances.

The variable elimination algorithm of Quantor also bears some resemblance to hyper binary resolution, in that variables are eliminated by performing all resolutions involving that variable in order to remove it from the theory. General resolution among n-ary clauses is a stronger rule of inference than HypBinRes, but it is difficult to use as a preprocessing technique due to its time and space complexity (however see [10]).

In this paper we have only discussed the static use of hyper binary resolution, i.e., its use prior to search. In [18] hyper binary resolution was used dynamically during search in a QBF solver and was found to also be useful in that context, but not as universally effective as its static use in a preprocessor.

7 Future Work

Additional techniques for preprocessing remain to be investigated. Based on the data we have gathered with our preprocessor, we can conclude that a very effective technique would be to run our preprocessor followed by running Quantor for a short period of time (10-20 seconds). This technique is capable of solving a surprising number of instances. As shown in Figure 2 the combination of the preprocessor and Quantor is in fact able to solve more instances than Skizzo [4]. Hence, by simply employing hyper resolution and variable elimination it is possible to gain an advantage over such sophisticated QBF solvers as Skizzo. Furthermore, this technique solves a number of instances that are particularly problematic for search based solvers. Figure 2 shows that this technique

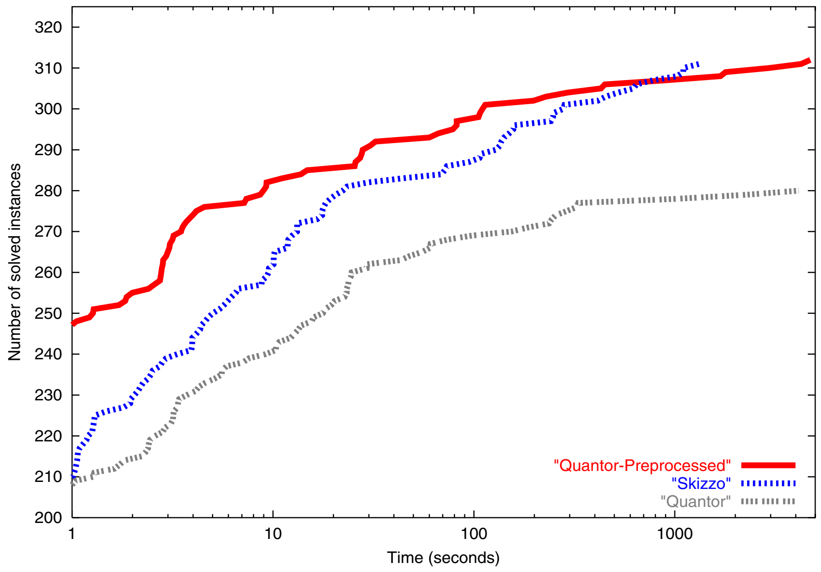


Fig. 2. Logarithmic time scale comparison of Quantor and Skizzo on the original and Quantor on the preprocessed benchmarks. Shown is the time in seconds versus the number of instances solved.

(the "Quantor-Preprocessed" line) can solve approximately 285 instances within 10 seconds. Yet if we continue to run Quantor for another 5000 seconds very few additional problems are solved (about 25 more instances). We have also found that search based solvers can solve a larger number of these "left-over" instances than Quantor.

This suggests the strategy of first running the preprocessor, then running Quantor, and then a search based solver if Quantor is unable to solve the instance quickly. Even more interesting would be to investigate obtaining the partially eliminated theory from Quantor after it has run for a few seconds, and then seeing if it could be further preprocessed or fed directly into a search based solver. The Skizzo solver [4] attempts to mix variable elimination with search in a related way, but it does not employ the extended preprocessing reasoning we have suggested here.

Another important direction for future work is to investigate how some of these ideas can be used to preprocess QCSP problems. Unfortunately although preprocessing is common in CSPs (achieving some level of local consistency prior to search), our particular technique of HypBinRes has no immediate analog in CSP. HypBinRes takes advantage of the fact that binary clauses form a tractable subtheory of SAT, however binary constraints are not a tractable subtheory in CSPs unless we also have binary valued domains. Nevertheless, what our work does indicate is that it might be worth investigating the usefulness of achieving higher levels of local consistency prior to search in QCSPs than might be sensible for standard CSPs. This is because QCSPs require more extensive search: all values of every universal variable have to be solved. So effort expended prior to search can be amortized over a larger search space. Note that HypBinRes+UR is a more powerful form of inference than what most of the QBF solvers are applying.

8 Conclusions

We have shown that preprocessing can be very effective for QBF and have presented substantial and significant empirical results to verify this claim. Nearly all of the publicly available instances are taken into account, and five different state of the art solvers are compared. The proposed method of preprocessing offers robust improvements across the different solvers among all tested benchmark families. The achieved improvement also includes almost 20 instances that to our knowledge have never been solved before.

References

1. B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithms for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
2. Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, 2002.
3. Fahiem Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Lecture Notes in Computer Science 2919*, pages 341–355, 2003.
4. M. Benedetti. skizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03, 2004.
5. M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proc. of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR04)*, number 3452 in LNCS. Springer, 2005.
6. M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In *Proc. of 9th International Joint Conference on Artificial Intelligence (IJCAI05)*, 2005.
7. M. Benedetti. Quantifier Trees for QBFs. In *Proc. of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT05)*, 2005.
8. A. Biere. Resolve and expand. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 238–246, 2004.
9. H. K. Büning, M. Karpinski, and A. Flügel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.
10. N. Een and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *In Proc. 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, number 3569 in LNCS. Springer, 2005.
11. Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving advanced reasoning tasks using quantified boolean formulas. In *AAAI/IAAI*, pages 417–422, 2000.
12. Ian P. Gent, Peter Nightingale, and Kostas Stergiou. Qcsp-solve: A solver for quantified constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 138–143, 2005.
13. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. http://www.qbflib.org/.
14. E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 364–369, 2001.
15. M. Narizzano and A. Tacchella. QBF evaluation 2005, 2005. http://www.qbflib.org/qbfeval/2005.
16. Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

17. H. Samulowitz and F. Bacchus. Using SAT in QBF. In *Principles and Practice of Constraint Programming*, pages 578–592. Springer-Verlag, New York, 2005. available at http://www.cs.toronto.edu/˜fbacchus/sat.html.
18. H. Samulowitz and F. Bacchus. Binary clause reasoning in QBF. to be published in SAT, 2006.
19. H. Samulowitz, Jessica Davies, and F. Bacchus. QBF Preprocessor Prequel, 2006. available at http://www.cs.toronto.edu/˜fbacchus/sat.html.
20. Kostas Stergiou. Repair-based methods for quantified csps. In *Principles and Practice of Constraint Programming*, pages 652–666, 2005.
21. L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In *Principles and Practice of Constraint Programming (CP2002)*, pages 185–199, 2002.

The Theory of Grammar Constraints

Meinolf Sellmann

Brown University
Department of Computer Science
115 Waterman Street, P.O. Box 1910
Providence, RI 02912
sello@cs.brown.edu

Abstract. By introducing the Regular Membership Constraint, Gilles Pesant pioneered the idea of basing constraints on formal languages. The paper presented here is highly motivated by this work, taking the obvious next step, namely to investigate constraints based on grammars higher up in the Chomsky hierarchy. We devise an arc-consistency algorithm for context-free grammars, investigate when logic combinations of grammar constraints are tractable, show how to exploit non-constant size grammars and reorderings of languages, and study where the boundaries run between regular, context-free, and context-sensitive grammar filtering.

Keywords: global constraints, regular grammar constraints, context-free grammar constraints, constraint filtering.

1 Introduction

With the introduction of the regular language membership constraint [9,10,2], a new field of study for filtering algorithms has opened. Given the great expressiveness of formal grammars and their (at least for someone with a background in computer science) intuitive usage, grammar constraints are extremely attractive modeling entities that subsume many existing definitions of specialized global constraints. Moreover, Pesant's implementation [8] of the regular grammar constraint has shown that this type of filtering can also be performed incrementally and generally so efficiently that it even rivals custom filtering algorithms for special regular grammar constraints like Stretch and Pattern [9,4].

In this paper, we theoretically investigate filtering problems that arise from grammar constraints. We answer questions like: Can we efficiently filter context-free grammar constraints? How can we achieve arc-consistency for conjunctions of regular grammar constraints? Given that we can allow non-constant grammars and reordered languages for the purposes of constraint filtering, what languages are suited for filtering based on regular and context-free grammar constraints? Are there languages that are suited for context-free, but not for regular grammar filtering?

Particularly, after recalling some essential basic concepts from the theory of formal languages in the next section, we devise an efficient arc-consistency algorithm that propagates context-free grammar constraints in Section 3. Then, in Section 4, we study how logic combinations of grammar constraints can be propagated efficiently. Finally, we investigate non-constant size grammars and reorderings of languages in Section 5.

2 Basic Concepts

We start our work by reviewing some well-known definitions from the theory of formal languages. For a full introduction, we refer the interested reader to [6]. All proofs that are omitted in this paper can also be found there.

Definition 1 (Alphabet and Words). *Given sets Z, Z_1, and Z_2, with Z_1Z_2 we denote the set of all* sequences *or* strings *$z = z_1z_2$ with $z_1 \in Z_1$ and $z_2 \in Z_2$, and we call Z_1Z_2 the* concatenation *of Z_1 and Z_2. Then, for all $n \in \mathbb{N}$ we denote with Z^n the set of all sequences $z = z_1z_2 \dots z_n$ with $z_i \in Z$ for all $1 \leq i \leq n$. We call z a* word *of length n, and Z is called an* alphabet *or set of* letters. *The empty word has length 0 and is denoted by ε. It is the only member of Z^0. We denote the set of all words over the alphabet Z by $Z^* := \bigcup_{n \in \mathbb{N}} Z^n$. In case that we wish to exclude the empty word, we write $Z^+ := \bigcup_{n \geq 1} Z^n$.*

Definition 2 (Grammar). *A* grammar *is a four-tuple $G = (\Sigma, N, P, S_0)$ where Σ is the alphabet, N is a finite set of* non-terminals, *$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ is the set of* productions, *and $S_0 \in N$ is the start non-terminal. We will always assume that $N \cap \Sigma = \emptyset$.*

Remark 1. We will use the following convention: Capital letters A, B, C, D, and E denote non-terminals, lower case letters a, b, c, d, and e denote letters in Σ, Y and Z denote symbols that can either be letters or non-terminals, u, v, w, x, y, and z denote strings of letters, and α, β, and γ denote strings of letters and non-terminals. Moreover, productions (α, β) in P can also be written as $\alpha \to \beta$.

Definition 3 (Derivation and Language)

- *Given a grammar $G = (\Sigma, N, P, S_0)$, we write $\alpha\beta_1\gamma \underset{G}{\Rightarrow} \alpha\beta_2\gamma$ iff there exists a production $\beta_1 \to \beta_2 \in P$. We write $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$ iff there exists a sequence of strings $\alpha_2, \dots, \alpha_{m-1}$ such that $\alpha_i \underset{G}{\Rightarrow} \alpha_{i+1}$ for all $1 \leq i < m$. Then, we say that α_m can be* derived *from α_1.*
- *We define the* language given by *G to be $L_G := \{w \in \Sigma^* \mid S_0 \overset{*}{\underset{G}{\Rightarrow}} w\}$.*

Definition 2 gives a very general form of grammars which is known to be Turing machine equivalent. Consequently, reasoning about languages given by general grammars is infeasible. For example, the word problem for grammars as defined above is undecidable.

Definition 4 (Word Problem). *Given a grammar $G = (\Sigma, N, P, S_0)$ and a word $w \in \Sigma^*$, the* word problem *consists in answering the question whether $w \in L_G$.*

Therefore, in the theory of formal languages, more restricted forms of grammars have been defined. Noam Chomsky introduced a hierarchy of decreasingly complex sets of languages [5]. In this hierarchy, the grammars given in Definition 2 are called Type-0 grammars. In the following, we define the Chomsky hierarchy of formal languages.

Definition 5 (Context-Sensitive, Context-Free, and Regular Grammars)

- *Given a grammar $G = (\Sigma, N, P, S_0)$ such that for all productions $\alpha \rightarrow \beta \in P$ we have that β is at least as long as α, then we say that the grammar G and the language L_G are* context-sensitive. *In Chomsky's hierarchy, these grammars are known as Type-1 grammars.*
- *Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (N \cup \Sigma)^*$, we say that the grammar G and the language L_G are* context-free. *In Chomsky's hierarchy, these grammars are known as Type-2 grammars.*
- *Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (\Sigma^* N \cup \Sigma^*)$, we say that G and the language L_G are* right-linear. *In Chomsky's hierarchy, these grammars are known as Type-3 grammars.*

Remark 2. The word problem becomes easier as the grammars become more and more restricted: For context-sensitive grammars, the problem is already decidable, but unfortunately PSPACE-complete. For context-free languages, the word problem can be answered in polynomial time. For Type-3 languages, the word problem can even be decided in time linear in the length of the given word.

For all grammars mentioned above there exists an equivalent definition based on some sort of automaton that accepts the respective language. As mentioned earlier, for Type-0 grammars, that automaton is the Turing machine. For context-sensitive languages it is a Turing machine with a linearly space-bounded tape. For context-free languages, it is the so-called push-down automaton (in essence a Turing machine with a stack rather than a tape). And for right-linear languages, it is the finite automaton (which can be viewed as a Turing machine with only one read-only input tape on which it cannot move backwards). Depending on what one tries to prove about a certain class of languages, it is convenient to be able to switch back and forth between different representations (i.e. grammars or automata). In this work, when reasoning about context-free languages, it will be most convenient to use the grammar representation. For right-linear languages, however, it is often more convenient to use the representation based on finite automata:

Definition 6 (Finite Automaton). *Given a finite set Σ, a* finite automaton *A is defined as a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of* states*, Σ denotes the* alphabet *of our language, $\delta \subseteq Q \times \Sigma \times Q$ defines the* transition function*, q_0 is the* start state*, and F is the set of* final states. *A finite automaton is called* deterministic *iff $(q, a, p_1), (q, a, p_2) \in \delta$ implies that $p_1 = p_2$.*

Definition 7 (Accepted Language). *The language defined by a finite automaton A is the set $L_A := \{w = (w_1, \dots w_n) \in \Sigma^* \mid \exists\ (p_0, \dots, p_n) \in Q^n\ \forall\ 1 \leq i \leq n : (p_{i-1}, w_i, p_i) \in \delta$ and $p_0 = q_0, p_n \in F\}$. L_A is called a* regular language.

Lemma 1. *For every right-linear grammar G there exists a finite automaton A such that $L_A = L_G$, and vice versa.*

Consequently, we can use the terms right-linear and regular synonymously.

3 Context-Free Grammar Constraints

Within constraint programming it would be convenient to use formal languages to describe certain features that we would like our solutions to exhibit. It is worth noting here that any constraint and conjunction of constraints really defines a formal language by itself when we view the instantiations of variables $X_1, \ldots, X_n$ with domains $D_1, \ldots, D_n$ as forming a word in $D_1 D_2 \ldots D_n$. Conversely, if we want a solution to belong to a certain formal language in this view, then we need appropriate constraints and constraint filtering algorithms that will allow us to express and solve such constraint programs efficiently. We formalize the idea by defining grammar constraints.

Definition 8 (Grammar Constraint). *For a given grammar $G = (\Sigma, N, P, S_0)$ and variables $X_1, \ldots, X_n$ with domains $D_1 := D(X_1), \ldots, D_n := D(X_n) \subseteq \Sigma$, we say that $Grammar_G(X_1, \ldots, X_n)$ is true for an instantiation $X_1 \leftarrow w_1, \ldots, X_n \leftarrow w_n$ iff it holds that $w = w_1 \ldots w_n \in L_G \cap D_1 \times \cdots \times D_n$.*

Gilles Pesant pioneered the idea to exploit formal grammars for constraint programming by considering regular languages [9,10]. Based on the review of our knowledge of formal languages in the previous section, we can now ask whether we can also develop efficient filtering algorithms for grammar constraints of higher-orders. Clearly, for Type-0 grammars, this is not possible, since the word problem is already undecidable. For context-sensitive languages, the word problem is PSPACE complete, which means that even checking the corresponding grammar constraint is computationally intractable.

However, for context-free languages deciding whether a given word belongs to the language can be done in polynomial time. Context-free grammar constraints come in particularly handy when we need to look for a recursive sequence of nested objects. Consider for instance the puzzle of forming a mathematical term based on two occurrences of the numbers 3 and 8, operators +, -, *, /, and brackets (,), such that the term evaluates to 24. The generalized problem is NP-hard, but when formulating the problem as a constraint program, with the help of a context-free grammar constraint we can easily express the syntactic correctness of the term formed. Or, closer to the real-world, consider the task of organizing a group of workers into a number of teams of unspecified size, each team with one team leader and one project manager who is the head of all team leaders. This organizational structure can be captured easily by a combination of an AllDifferent and a context-free grammar constraint. Therefore, in this section we will develop an algorithm that propagates context-free grammar constraints.

3.1 Parsing Context-Free Grammars

One of the most famous algorithms for parsing context-free grammars is the algorithm by Cocke, Younger, and Kasami (CYK). It takes as input a word $w \in \Sigma^n$ and a context-free grammar $G = (\Sigma, N, P, S_0)$ in some special form and decides in time $O(n^3|P|)$ whether it holds that $w \in L_G$. The algorithm is based on the dynamic programming principle. In order to keep the recursion equation under control, the algorithm needs to assume that all productions are length-bounded on the right-hand side.

Definition 9 (Chomsky Normal Form). *A context-free grammar $G = (\Sigma, N, P, S_0)$ is said to be in* Chomsky Normal Form *iff for all productions $A \rightarrow \alpha \in P$ we have that $\alpha \in \Sigma^1 \cup N^2$.*

Lemma 2. *Every context free grammar G such that $\varepsilon \notin L_G$ can be transformed into a grammar H such that $L_G = L_H$ and H is in Chomsky Normal Form.*

The proof of this lemma is given in [6]. It is important to note that the proof is constructive but that the resulting grammar H may be exponential in size of G, which is really due to the necessity to remove all productions $A \rightarrow \varepsilon$. When we view the grammar size as constant (i.e. if the size of the grammar is independent of the word-length as it is commonly assumed in the theory of formal languages), then this is not an issue. As a matter of fact, in most references one will simply read that CYK could solve the word problem for any context-free language in cubic time. For now, let us assume that indeed all grammars given can be treated as having constant-size, and that our asymptotic analysis only takes into account the increasing word lengths. We will come back to this point later in Section 4 when we discuss logic combinations of grammar constraints, and in Section 5 when we discuss the possibility of non-constant grammars and reorderings.

Now, given a word $w \in \Sigma^n$, let us denote the sub-sequence $w_i w_{i+1} \dots w_{i+j-1}$ by w_{ij}. Based on a grammar $G = (\Sigma, N, P, S_0)$ in Chomsky Normal Form, CYK determines iteratively the set of all non-terminals from where we can derive w_{ij}, i.e. $S_{ij} := \{A \in N \mid A \underset{G}{\overset{*}{\Rightarrow}} w_{ij}\}$ for all $1 \leq i \leq n$ and $1 \leq j \leq n - i$. It is easy to initialize the sets S_{i1} just based on w_i and all productions $A \rightarrow w_i \in P$. Then, for j from 2 to n and i from 1 to $n - j + 1$, we have that

$$S_{ij} = \bigcup_{k=1}^{j-1} \{A \mid A \rightarrow BC \in P \text{ with } B \in S_{ik} \text{ and } C \in S_{i+k,j-k}\}. \tag{1}$$

Then, $w \in L_G$ iff $S_0 \in S_{1n}$. From the recursion equation it is simple to derive that CYK can be implemented to run in time $O(n^3|P|) = O(n^3)$ when we treat the size of the grammar as a constant.

3.2 Example

Assume we are given the following context-free, normal-form grammar $G = (\{], [\},$-$\{A, B, C, S_0\}, \{S_0 \rightarrow AC, S_0 \rightarrow S_0S_0, S_0 \rightarrow BC, B \rightarrow AS_0, A \rightarrow [\ , C \rightarrow]\ \}, S_0)$ that gives the language L_G of all correctly bracketed expressions (like, for example, "[[][]]" or "[][[]]"). Given the word "[][[]]", CYK first sets $S_{11} = S_{31} = S_{41} = \{A\}$, and $S_{21} = S_{51} = S_{61} = \{C\}$. Then it determines the non-terminals from which we can derive sub-sequences of length 2: $S_{12} = S_{42} = \{S_0\}$ and $S_{22} = S_{32} = S_{52} = \emptyset$. The only other non-empty sets that CYK finds in iterations regarding longer sub-sequences are $S_{34} = \{S_0\}$ and $S_{16} = \{S_0\}$. Consequently, since $S_0 \in S_{16}$, CYK decides (correctly) that $[][[]] \in L_G$.

Algorithm 1. CFCG Filtering Algorithm

1. We run the dynamic program based on recursion equation 1 with initial sets $S_{i1} := \{A \mid A \to v \in P, v \in D_i\}$.
2. We define the directed graph $Q = (V, E)$ with node set $V := \{v_{ijA} \mid A \in S_{ij}\}$ and arc set $E := E_1 \cup E_2$ with $E_1 := \{(v_{ijA}, v_{ikB}) \mid \exists\ C \in S_{i+k,j-k} :\ A \to BC \in P\}$ and $E_2 := \{(v_{ijA}, v_{i+k,j-k,C}) \mid \exists\ B \in S_{ik} :\ A \to BC \in P\}$ (see Figure 1).
3. Now, we remove all nodes and arcs from Q that cannot be reached from v_{1nS_0} and denote the resulting graph by $Q' := (V', E')$.
4. We define $S'_{ij} := \{A \mid v_{ijA} \in V'\} \subseteq S_{ij}$, and set $D'_i := \{v \mid \exists\ A \in S'_{i1} :\ A \to v \in P\}$.

3.3 Context-Free Grammar Filtering

We denote a given grammar constraint $Grammar_G(X_1, \ldots, X_n)$ over a context-free grammar G in Chomsky Normal Form by $CFGC_G(X_1, \ldots, X_n)$. Obviously, we can use CYK to determine whether $CFGC_G(X_1, \ldots, X_n)$ is satisfied for a full instantiation of the variables, i.e. we could use the parser for generate-and-test purposes. In the following, we show how we can augment CYK to a filtering algorithm that achieves generalized arc-consistency for $CFGC$.

First, we observe that we can check the satisfiability of the constraint by making just a very minor adjustment to CYK. Given the domains of the variables, we can decide whether there exists a word $w \in D_1 \ldots D_n$ such that $w \in L_G$ simply by adding all non-terminals A to S_{i1} for which there exists a production $A \to v \in P$ with $v \in D_i$. From the correctness of CYK it follows trivially that the constraint is satisfiable iff $S_0 \in S_{1n}$. The runtime of this algorithm is the same as that for CYK.

As usual, whenever we have a polynomial-time algorithm that can decide the satisfiability of a constraint, we know already that achieving arc-consistency is also computationally tractable. A brute force approach could simply probe values by setting $D_i := \{v\}$, for every $1 \leq i \leq n$ and every $v \in D_i$, and checking whether the constraint is still satisfiable or not. This method would result in a runtime in $O(n^4 D|P|)$, where $D \leq |\Sigma|$ is the size of the largest domain D_i.

We will now show that we can achieve a much improved filtering time. The core idea is once more to exploit Mike Trick's method of filtering dynamic programs [11]. Roughly speaking, when applied to our CYK-constraint checker, Trick's method simply reverses the recursion process after it has assured that the constraint is satisfiable so as to see which non-terminals in the sets S_{i1} can actually be used in the derivation of any word $w \in L_G \cap (D_1 \ldots D_n)$. The methodology is formalized in Algorithm 1.

Lemma 3. *In Algorithm 1:*

1. *It holds that $A \in S_{ij}$ iff there exists a word $w_i \ldots w_{i+j-1} \in D_i \ldots D_{i+j-1}$ such that $A \underset{G}{\overset{*}{\Rightarrow}} w_i \ldots w_{i+j-1}$.*
2. *It holds that $B \in S'_{ik}$ iff there exists a word $w \in L_G \cap (D_1 \ldots D_n)$ such that $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1 \ldots w_{i-1}\ B\ w_{i+k} \ldots w_n$.*

Proof. 1. We induce over j. For $j = 1$, the claim holds by definition of S_{i1}. Now assume $j > 1$ and that the claim is true for all S_{ik} with $1 \leq k < j$. Now, by definition of S_{ij}, $A \in S_{ij}$ iff there exists a $1 \leq k < j$ and a production $A \to BC \in P$ such that $B \in S_{ik}$ and $C \in S_{i+k,j-k}$. Thus, $A \in S_{ij}$ iff there exist $w_{ik} \in D_i \dots D_{i+k-1}$ and $w_{i+k,j-k} \in D_{i+k} \dots D_{i+j-1}$ such that $A \underset{G}{\overset{*}{\Rightarrow}} w_{ik}w_{i+k,j-k}$.

2. We induce over k, starting with $k = n$ and decreasing to $k = 1$. For $k = n$, ${S'}_{1k} = {S'}_{1n} \subseteq \{S_0\}$, and it is trivially true that $S_0 \underset{G}{\overset{*}{\Rightarrow}} S_0$. Now let us assume the claim holds for all ${S'}_{ij}$ with $k < j \leq n$. Choose any $B \in {S'}_{ik}$. According to the definition of ${S'}_{ik}$ there exists a path from v_{1nS_0} to v_{ikB}. Let $(v_{ijA}, v_{ikB}) \in E_1$ be the last arc on any one such path (the case when the last arc is in E_2 follows analogously). By the definition of E_1 there exists a production $A \to BC \in P$ with $C \in S_{i+k,j-k}$. By induction hypothesis, we know that there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+j} \dots w_n$. Thus, $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1 \dots w_{i-1}\ BC\ w_{i+j} \dots w_n$. And therefore, with (1) and $C \in S_{i+k,j-k}$, there exists a word $w_{i+k} \dots w_{i+j-1} \in D_{i+k} \dots D_{i+j-1}$ such that $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1 \dots w_{i-1}\ B\ w_{i+k} \dots w_{i+j-1}\ w_{i+j} \dots w_n$. Since we can also apply (1) to non-terminal B, we have proven the claim.

□

Theorem 1. *Algorithm 1 achieves generalized arc-consistency for the $CFGC$.*

Proof. We show that $v \notin {D'}_i$ iff for all words $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$ it holds that $v \neq w_i$.

$\Rightarrow$ (Correctness) Let $v \notin {D'}_i$ and $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$. Due to $w \in L_G$ there must exist a derivation $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+1} \dots w_n \underset{G}{\Rightarrow} w_1 \dots w_{i-1} w_i w_{i+1} \dots w_n$ for some $A \in N$ with $A \to w_i \in P$. According to Lemma 3, $A \in {S'}_{i1}$, and thus $w_i \in {D'}_i$, which implies $v \neq w_i$ as $v \notin {D'}_i$.

$\Leftarrow$ (Effectiveness) Now let $v \in {D'}_i \subseteq D_i$. According to the definition of ${D'}_i$, there exists some $A \in {S'}_{i1}$ with $A \to v \in P$. With Lemma 3 we know that then there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+1} \dots w_n$. Thus, it holds that $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1 \dots w_{i-1}\ v\ w_{i+1} \dots w_n \in L_G \cap (D_1 \dots D_n)$.

□

We now have a filtering algorithm that achieves generalized arc-consistency for context-free grammar constraints. Since the computational effort is dominated by carrying out the recursion equation, Algorithm 1 runs asymptotically in the same time as CYK. In essence, this implies that checking one complete assignment via CYK is as costly as performing full arc-consistency filtering for $CFGC$. Clearly, achieving arc-consistency for a grammar constraint is at least as hard as parsing. Now, there exist faster parsing algorithms for context-free grammars. For example, the fastest known algorithm was developed by Valiant and parses context-free grammars in time $O(n^{2.8})$. While this is only moderately faster than the $O(n^3)$ that CYK requires, there also exist special purpose parsers for non-ambiguous context-free grammars (i.e. grammars where each word in the language has exactly one parse tree) that run in $O(n^2)$. Now, it is known that there exist inherently ambiguous context-free languages, so these parsers lack some

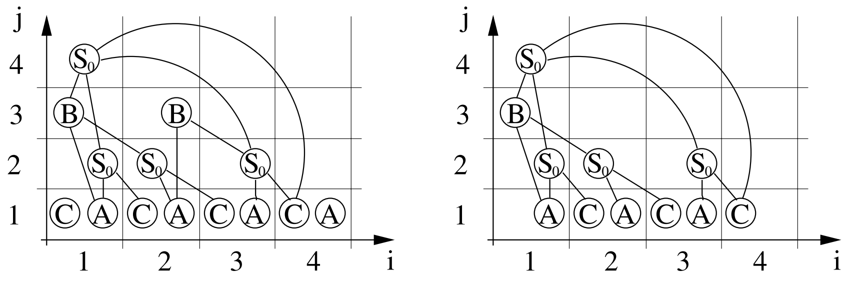


Fig. 1. Context-Free Filtering: Assume we are given the context-free grammar from section 3.2 again. A rectangle with coordinates (i, j) contains one node v_{ijA} for each non-terminal A in the set S_{ij}. All arcs are considered to be directed from top to bottom. The left picture shows the situation after step (2). S_0 is in S_{14}, therefore the constraint is satisfiable. The right picture illustrates the shrunken graph with sets S'_{ij} after all parts have been removed that cannot be reached from node v_{14S_0}. We see that the value ']' will be removed from D_1 and '[' from D_4.

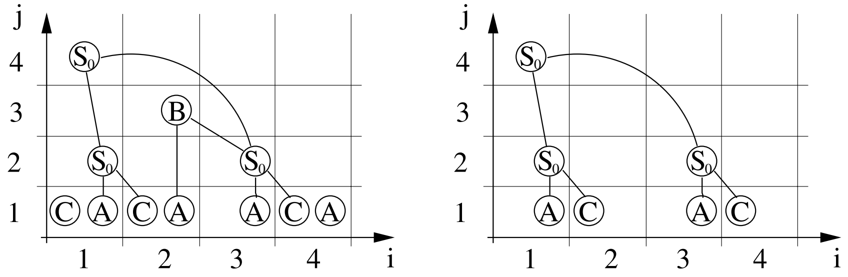


Fig. 2. We show how the algorithm works when the initial domain of X_3 is $D_3 = \{[\}$. The left picture shows sets S_{ij} and the right the sets S'_{ij}. We see that the constraint filtering algorithm determines the only word in $L_G \cap D_1 \dots D_4$ is "[][]".

generality. However, in case that a user specifies a grammar that is non-ambiguous it would actually be nice to have a filtering algorithm that runs in quadratic rather than cubic time. It is a matter of further research to find out whether grammar constraint propagation can be done faster for non-ambiguous context-free grammars.

4 Logic Combinations of Grammar Constraints

We define regular grammar constraints analogously to $CFGC$, but as in [10] we base it on automata rather than right-linear grammars:

Definition 10 (Regular Grammar Constraint). *Given a finite automaton A and a right-linear grammar G with $L_A = L_G$, we set*

$$RGC_A(X_1, \dots, X_n) := Grammar_G(X_1, \dots, X_n).$$

Efficient arc-consistency algorithms for *RGC*s have been developed in [9,10]. Now equipped with efficient filtering algorithms for regular and context-free grammar constraints, in the spirit of [7,1,3] we focus on certain questions that arise when a problem is modeled by logic combinations of these constraints. An important aspect when investigating logical combinations of grammar constraints is under what operations the given class of languages is closed. For example, when given a conjunction of regular grammar constraints, the question arises whether the conjunction of the constraints could not be expressed as one global *RGC*. This question can be answered affirmatively since the class of regular languages is known to be closed under intersection. In the following we summarize some relevant, well-known results for formal languages (see for instance [6]).

Lemma 4. *For every regular language L_{A^1} based on the finite automaton A^1 there exists a deterministic finite automaton A^2 such that $L_{A^1} = L_{A^2}$.*

Proof. When $Q^1 = \{q_0, \dots, q_{n-1}\}$, we set $A^2 := (Q^2, \Sigma, \delta^2, q_0^2, F^2)$ with $Q^2 := 2^{Q^1}$, $q_0^2 = \{q_0^1\}$, $\delta^2 := \{(P, a, R) \mid R = \{r \in Q^1 \mid \exists\, p \in P : \ (p, a, r) \in \delta^1\}\}$, and $F^2 := \{P \subseteq Q^1 \mid \exists\, p \in P \cap F^1\}$. With this construction, it is easy to see that $L_{A^1} = L_{A^2}$. □

We note that the proof above gives a construction that can change the properties of the language representation, just like we had noted it earlier for context-free grammars that we had transformed into Chomsky Normal Form first before we could apply CYK for parsing and filtering. And just like we were faced with an exponential blow-up of the representation when bringing context-free grammars into normal-form, we see the same again when transforming a non-deterministic finite automaton of a regular language into a deterministic one.

Theorem 2. *Regular languages are closed under the following operations:*

- *Union*
- *Intersection*
- *Complement*

Proof. Given two regular languages L_{A^1} and L_{A^2} with respective finite automata $A^1 = (Q^1, \Sigma, \delta^1, q_0^1, F^1)$ and $A^2 = (Q^2, \Sigma, \delta^2, q_0^2, F^2)$, without loss of generality, we may assume that the sets Q^1 and Q^2 are disjoint and do not contain symbol q_0^3.

- We define $Q^3 := Q^1 \cup Q^2 \cup \{q_0^3\}$, $\delta^3 := \delta^1 \cup \delta^2 \cup \{(q_0^3, a, q) \mid (q_0^1, a, q) \in \delta^1 \text{ or } (q_0^2, a, q) \in \delta^2)\}$, and $F^3 := F^1 \cup F^2$. Then, it is straight-forward to see that the automaton $A^3 := (Q^3, \Sigma, \delta^3, q_0^3, F^3)$ defines $L_{A^1} \cup L_{A^2}$.
- We define $Q^3 := Q^1 \times Q^2$, $\delta^3 := \{((q^1, q^2), a, (p^1, p^2) \mid \exists (q^1, a, p^1) \in \delta^1, (q^2, a, p^2) \in \delta^2\}$, and $F^3 := F^1 \times F^2$. The automaton $A^3 := (Q^3, \Sigma, \delta^3, (q_0^1, q_0^2), F^3)$ defines $L_{A^1} \cap L_{A^2}$.
- According to Lemma 4, we may assume that A^1 is a deterministic automaton. Then, $(Q^1, \Sigma, \delta^1, q_0^1, Q^1 \setminus F^1)$ defines $L_{A^1}^C$. □

The results above suggest that any logic combination (disjunction, conjunction, and negation) of RGCs can be expressed as one global RGC. While this is true in principle, from a computational point of view, the size of the resulting automaton needs to be taken into account. In terms of disjunctions of RGCs, all that we need to observe is that the algorithm developed in [9] actually works with non-deterministic automata as well. In the following, denote by m an upper bound on the number of states in all automata involved, and denote the size of the alphabet Σ by D. We obtain our first result for disjunctions of regular grammar constraints:

Lemma 5. *Given RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve arc-consistency for the global constraint $\bigvee_i R_i$ in time $O((km + k)nD) = O(nDk)$ for automata with constant state-size m.*

If all that we need to consider are disjunctions of RGCs, then the result above is subsumed by the well known technique of achieving arc-consistency for disjunctive constraints which simply consists in removing, for each variable domain, the intersection of all values removed by the individual constraints. However, when considering conjunctions over disjunctions the result above is interesting as it allows us to treat a disjunctive constraint over RGCs as one new RGC of slightly larger size.

Now, regarding conjunctions of RGCs, we find the following result:

Lemma 6. *Given RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve arc-consistency for the global constraint $\bigwedge_i R_i$ in time $O(nDm^k)$.*

Finally, for the complement of a regular constraint, we have:

Lemma 7. *Given an RGC R* based on a deterministic automaton*, we can achieve arc-consistency for the constraint $\neg R$ in time $O(nDm) = O(nD)$ for an automaton with constant state-size.*

Proof. Lemmas 5- 7 are an immediate consequence of the results in [9] and the constructive proof of Theorem 2. □

Note that the lemma above only covers RGCs for which we know a deterministic finite automaton. However, when negating a disjunction of regular grammar constraints, the automaton to be negated is non-deterministic. Fortunately, this problem can be entirely avoided: When the initial automata associated with the elementary constraints of a logic combination of regular grammar constraints are deterministic, we can apply the rule of DeMorgan so as to only have to apply negations to the original constraints rather than the non-deterministic disjunctions or conjunctions thereof. With this method, we have:

Corollary 1. *For any logic combination (disjunction, conjunction, and negation) of deterministic RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve generalized arc-consistency in time $O(nDm^k)$.*

Regarding logic combinations of context-free grammar constraints, unfortunately we find that this class of languages is not closed under intersection and complement, and the mere disjunction of context-free grammar constraints is not interesting given the standard methods for handling disjunctions. We do know, however, that context-free languages are closed under intersection with regular languages. It is a subject of further research to assess how big the resulting grammars can become.

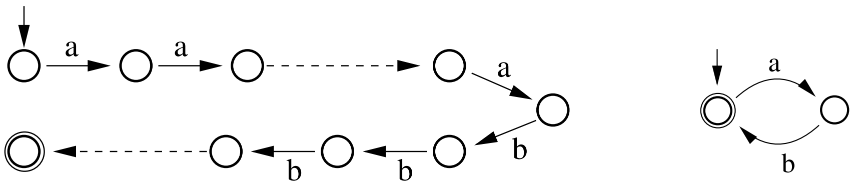


Fig. 3. Regular grammar filtering for $\{a^n b^n\}$. The left figure shows a linear-size automaton, the right an automaton that accepts a reordering of the language.

5 Limits of the Expressiveness of Grammar Constraints

So far we have been very careful to mention explicitly how the size of the state-space of a given automaton or how the size of the set of non-terminals of a grammar influences the running time of our filtering algorithms. From the theory of formal languages' view-point, this is rather unusual, since here the interest lies purely in the asymptotic runtime with respect to the word-length. For the purposes of constraint programming, however, a grammar may very well be generated on the fly and may depend on the word-length, whenever this can be done efficiently. This fact makes grammar constraints even more expressive and powerful tools from the modeling perspective. Consider for instance the context-free language $L = \{a^n b^n\}$ that is well-known not to be regular. Note that, within a constraint program, the length of the word is known — simply by considering the number of variables that define the scope of the grammar constraint. Now, by allowing the automaton to have $2n + 1$ states, we can express that the first n variables shall take the value a and the second n variables shall take the value b by means of a regular grammar constraint. Of course, larger automata also result in more time that is needed for propagation. However, as long as the grammar is polynomially bounded in the word-length, we can still guarantee a polynomial filtering time.

The second modification that we can safely allow is the reordering of variables. In the example above, assume the first n variables are $X_1, \ldots, X_n$ and the second n variables are $Y_1, \ldots, Y_n$. Then, instead of building an automaton with $2n + 1$ states that is linked to $(X_1, \ldots, X_n, Y_1, \ldots, Y_n)$, we could also build an automaton with just two states and link it to $(X_1, Y_1, X_2, Y_2, \ldots, X_n, Y_n)$ (see Figure 3). The same ideas can also be applied to $\{a^n b^n c^n\}$ which is not even context-free but context-sensitive. The one thing that we really need to be careful about is that, when we want to exploit our earlier results on the combination of grammar constraints, we need to make sure that the ordering requirements specified in the respective theorems are met (see for instance Lemmas 5 and 6).

While these ideas can be exploited to model some required properties of solutions by means of grammar constraints, they make the theoretical analysis of which properties can or cannot be modeled by those constraints rather difficult. Where do the boundaries run between languages that are suited for regular or context-free grammar filtering? The introductory example, as uninteresting as it is from a filtering point of view, showed already that the theoretical tools that have been developed to assess that a certain language cannot be expressed by a grammar on a lower level in the Chomsky hierarchy fail. The well-known pumping lemmas for regular and context-free grammars for instance rely on the fact that grammars be constant in size. As soon as we allow reordering and/or non-constant size grammars, they do not apply anymore.

To be more formal: what we really need to consider for propagation purposes is not an entire infinite set of words that form a language, but just a slice of words of a given length. I.e., given a language L what we need to consider is just $L_{|n} := L \cap \Sigma^n$. Since $L_{|n}$ is a finite set, it really is a regular language. In that regard, our previous finding that $\{a^n b^n\}$ for fixed n can be modeled as regular language is not surprising. The interesting aspect is that we can model $\{a^n b^n\}$ by a regular grammar of size *linear in* n, or even of *constant size* when reordering the variables appropriately.

Definition 11 (Suitedness for Grammar Filtering). *Given a language L over the alphabet Σ, we say that L is* suited for regular (or context-free) grammar filtering *iff there exist constants $k, n \in \mathbb{N}$ such that there exists a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and a finite automaton A (or normal-form context-free grammar G) such that both σ and A (G) can be constructed in time $O(n^k)$ with $\sigma(L_{|n}) = \sigma(L \cap \Sigma^n) := \{w_{\sigma(1)} \ldots w_{\sigma(n)} \mid w_1 \ldots w_n \in L\} = L_A$ ($\sigma(L_{|n}) = L_G$).*

Remark 3. Note that the previous definition implies that the size of the automaton (grammar) constructed is in $O(n^k)$. Note further that, if the given language is regular (context-free), then it is also suited for regular (context-free) grammar filtering.

Now, we have the terminology at hand to express that some properties cannot be modeled efficiently by regular or context-free grammar constraints. We start out by proving the following useful Lemma:

Lemma 8. *Denote with $N = \{S_0, \ldots, S_r\}$ a set of non-terminal symbols and $G = (\Sigma, N, P, S_0)$ a context-free grammar in Chomsky-Normal-Form. Then, for every word $w \in L_G$ of length n, there must exist $t, u, v \in \Sigma^*$ and a non-terminal symbol $S_i \in N$ such that $S_0 \underset{G}{\overset{*}{\Rightarrow}} tS_iv$, $S_i \underset{G}{\overset{*}{\Rightarrow}} u$, $w = tuv$, and $n/4 \leq |u| \leq n/2$.*

Proof. Since $w \in L_G$, there exists a derivation $S_0 \underset{G}{\overset{*}{\Rightarrow}} w$ in G. We set $h_1 := 0$. Assume the first production used in the derivation of w is $S_{h_1} \to S_{k_1} S_{k_2}$ for some $0 \leq k_1, k_2 \leq r$. Then, there exist words $u_1, u_2 \in \Sigma^*$ such that $w = u_1 u_2$, $S_{k_1} \underset{G}{\overset{*}{\Rightarrow}} u_1$, and $S_{k_2} \underset{G}{\overset{*}{\Rightarrow}} u_2$. Now, either u_1 or u_2 fall into the length interval claimed by the lemma, or one of them is longer than $n/2$. In the first case, we are done, the respective non-terminal has the claimed properties. Otherwise, if $|u_1| < |u_2|$ we set $h_2 := k_2$, else $h_2 := k_1$. Now, we repeat the argument that we just made for S_{h_1} for the non-terminal S_{h_2} that derives to the longer subsequence of w. At some point, we are bound to hit a production $S_{h_m} \to S_{k_m} S_{k_{m+1}}$ where S_{h_m} still derives to a subsequence of length greater than $n/2$, but both $S_{k_m}, S_{k_{m+1}}$ derive to subsequences that are at most $n/2$ letters long. The longer of the two is bound to have length greater than $n/4$, and the respective non-terminal has the desired properties. □

Now consider the language

$$L_{\text{AllDiff}} := \{w \in \mathbb{N}^* \mid \forall\, 1 \leq k \leq |w| : \; \exists\, 1 \leq i \leq |w| : \; w_i = k\}.$$

Since the word problem for L_{AllDiff} can be decided in linear space, L_{AllDiff} is (at most) context-sensitive.

Theorem 3. *L_{AllDiff} is not suited for context-free grammar filtering.*

Proof. We observe that reordering the variables linked to the constraint has no effect on the language itself, i.e. we have that $\sigma(L_{\text{AllDiff}|n}) = L_{\text{AllDiff}|n}$ for all permutations σ. Now assume that, for all $n \in \mathbb{N}$, we can construct a normal-form context-free grammar $G = (\{1, \dots, n\}, \{S_0, \dots, S_r\}, P, S_0)$ that generates $L_{\text{AllDiff}|n}$. We will show that the minimum size for G is exponential in n. Due to Lemma 8, for every word $w \in L_{\text{AllDiff}|n}$ there exist $t, u, v \in \{1, \dots, n\}^*$ and a non-terminal symbol S_i such that $S_0 \underset{G}{\overset{*}{\Rightarrow}} tS_iv$, $S_i \underset{G}{\overset{*}{\Rightarrow}} u$, $w = tuv$, and $n/4 \le |u| \le n/2$. Now, let us count for how many words non-terminal S_i can be used in the derivation. Since from S_i we can derive u, all terminal symbols that are in u must appear in one block in any word that can use S_i for its derivation. This means that there can be at most $(n - |u|)(n - |u|)!(|u|)! \le \frac{3n}{4}(\frac{n}{2}!)^2$ such words. Consequently, since there exist $n!$ many words in the language, the number of non-terminals is bounded from below by

$$r \quad \ge \quad \frac{n!}{\frac{3n}{4}(\frac{n}{2}!)^2} \quad = \quad \frac{4(n-1)!}{3(\frac{n}{2}!)^2} \quad \approx \quad \frac{4\sqrt{2}}{3\sqrt{\pi}}\frac{2^n}{n^{3/2}} \quad \in \quad \omega(1.5^n).$$

□

Now, the interesting question arises whether there exist languages at all that are fit for context-free, but not for regular grammar filtering? If this wasn't the case, then the algorithm developed in Section 3 would be utterly useless. What makes the analysis of suitedness so complicated is the fact that the modeler has the freedom to change the ordering of variables that are linked to the grammar constraint — which essentially allows him or her to change the language almost ad gusto. We have seen an example for this earlier where we proposed that a^nb^n could be modeled as $(ab)^n$.

Theorem 4. *The set of languages that are suited for context-free grammar filtering is a strict superset of the set of languages that are suited for regular grammar filtering.*

Proof. Consider the language $L = \{ww^R\#vv^R \mid v, w \in \{0,1\}^*\} \subseteq \{0,1,\#\}^*$ (where x^R denotes the reverse of a word x). Obviously, L is context-free with the grammar $(\{0,1,\#\}, \{S_0, S_1\}, \{S_0 \to S_1\#S_1,\ S_1 \to 0S_10,\ S_1 \to 1S_11,\ S_1 \to \varepsilon\}, S_0)$. Consequently L is suited for context-free grammar filtering.

Note that, when the position $2k+1$ of the sole occurrence of the letter $\#$ is fixed, for every position i containing a letter 0 or 1, there exists a *partner position* $p^k(i)$ so that both corresponding variables are forced to take the same value. Crucial to our following analysis is the fact that, in every word $x \in L$ of length $|x| = n = 2l+1$, every even (odd) position is linked in this way exactly with *every* odd (even) position for some placement of $\#$. Formally, we have that $\{p^k(i) \mid 0 \le k \le l\} = \{1, 3, 5, \dots, n\}$ ($\{p^k(i) \mid 0 \le k \le l\} = \{2, 4, 6, \dots, 2l\}$) when i is even (odd).

Now, assume that, for every odd $n = 2l+1$, there exists a finite automaton that accepts some reordering of $L \cap \{0,1,\#\}^n$ under variable permutation $\sigma : \{1, \dots, n\} \to \{1, \dots, n\}$. For a given position $2k+1$ of $\#$ (in the original ordering), by $dist_\sigma^k := \sum_{i=1,3,\dots,2l+1} |\sigma(i) - \sigma(p^k(i))|$ we denote the total distance of the pairs after the reordering through σ. Then, the average total distance after reordering through σ is

$$\begin{aligned}\tfrac{1}{l+1}\textstyle\sum_{0\leq k\leq l} dist_\sigma^k &= \tfrac{1}{l+1}\textstyle\sum_{0\leq k\leq l}\sum_{i=1,3,\dots,2l+1} |\sigma(i)-\sigma(p^k(i))|\\ &= \tfrac{1}{l+1}\textstyle\sum_{i=1,3,\dots,2l+1}\sum_{0\leq k\leq l} |\sigma(i)-\sigma(p^k(i))|.\end{aligned}$$

Now, since we know that every odd i has l even partners, even for an ordering σ that places all partner positions in the immediate neighborhood of i, we have that

$$\sum_{0\leq k\leq l} |\sigma(i)-\sigma(p^k(i))| \geq 2 \sum_{s=1,\dots,\lfloor l/2\rfloor} s = (\lfloor l/2\rfloor + 1)\lfloor l/2\rfloor.$$

Thus, for sufficiently large l, the average total distance under σ is

$$\begin{aligned}\tfrac{1}{l+1}\textstyle\sum_{0\leq k\leq l} dist_\sigma^k &\geq \tfrac{1}{l+1}\textstyle\sum_{i=1,3,\dots,2l+1}(\lfloor l/2\rfloor+1)\lfloor l/2\rfloor\\ &\geq \tfrac{l}{l+1}(\lfloor l/2\rfloor+1)\lfloor l/2\rfloor\\ &\geq l^2/8.\end{aligned}$$

Consequently, for any given reordering σ, there must exist a position $2k+1$ for the letter # such that the total distance of all pairs of linked positions is at least the average, which in turn is greater or equal to $l^2/8$. Therefore, since the maximum distance is $2l$, there must exist at least $l/16$ pairs that are at least $l/8$ positions apart after reordering through σ. It follows that there exists an $1 \leq r \leq n$ such that there are at least $l/128$ positions $i \leq r$ such that $p^k(i) > r$. Consequently, after reading r inputs, the finite automaton that accepts the reordering of $L \cap \{0, 1, \#\}^n$ needs to be able to reach at least $2^{l/128}$ different states. It is therefore not polynomial in size. It follows: L is not suited for regular grammar filtering. □

As a final remark, it is interesting that $\{ww^R\#vv^R\}_{|2l+1} = \bigcup_{k=1}^{l}\{ww^R\#vv^R \mid$ - $|w| = k, |v| = l-k\}$, and $\{ww^R\#vv^R \mid |w| = k, |v| = l-k\}$ is actually suited for regular grammar filtering when each of the sets is reordered appropriately. Now, Lemma 5 cannot be applied, since the different constraints use different reorderings of the variables. However, we could apply standard disjunctive constraint filtering. Ultimately, context-free grammar filtering only appears to become unavoidable for unrestricted concatenations of non-regular grammars such as $\{w_1w_1^R\#w_2w_2^R\#\dots w_kw_k^R\}$, or our example grammar that generates correctly bracketed expressions, or the language of syntactically correct mathematical expressions, to name just a few.

6 Conclusions

We investigated the idea of basing constraints on formal languages. Particularly, we devised an efficient arc-consistency algorithm for grammar constraints based on context-free grammars in Chomsky Normal Form. We studied logic combinations of grammar constraints and showed where the boundaries run between regular, context-free, and context-sensitive grammar constraints when allowing non-constant grammars and reorderings of variables. Our hope is that grammar constraints can serve as powerful modeling entities for constraint programming in the future, and that our theory can help to better understand and tackle the computational problems that arise in the context of grammar constraint filtering.

References

1. F. Bacchus and T. Walsh. Propagating Logical Combinations of Constraints. *International Joint Conference on Artificial Intelligence*, pp. 35–40, 2005.
2. N. Beldiceanu, M. Carlsson, T. Petit. Deriving Filtering Algorithms from Constraint Checkers. *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, LNCS 3258, 2004.
3. C. Bessière and J-C. Régin. Local consistency on conjunctions of constraints. *Workshop on Non-Binary Constraints*, 1998.
4. S. Bourdais, P. Galinier, G. Pesant. HIBISCUS: A Constraint Programming Application to Staff Scheduling in Health Care. *International Conference on Principles and Practice of Constraint Programming*, LNCS 2833:153–167, 2003.
5. N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory* 2:113–124, 1956.
6. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979.
7. O. Lhomme. Arc-consistency Filtering Algorithms for Logical Combinations of Constraints. *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*, 2004.
8. L-M. Rousseau, G. Pesant, W-J. van Hoeve. On Global Warming: Flow Based Soft Global Constraint. *Informs*, p. 27, 2005.
9. G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. *International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 3258:482–495, 2004.
10. G. Pesant. A Regular Language Membership Constraint for Sequences of Variables *International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pp. 110–119, 2003.
11. M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 113–124, 2001.

Constraint Programming Models for Graceful Graphs

Barbara M. Smith

Cork Constraint Computation Centre, University College Cork, Ireland
b.smith@4c.ucc.ie

Abstract. The problem of finding a graceful labelling of a graph, or proving that the graph is not graceful, has previously been modelled as a CSP. A new and much faster CSP model of the problem is presented, with several new results for graphs whose gracefulness was previously unknown. Several classes of graph that are conjectured to be graceful only for small instances are investigated: after a certain size, it appears that for some of these classes the search to prove that there is no graceful labelling is essentially the same for each successive instance. The possibility of constructing a proof of the conjecture based on the search is discussed.

1 Introduction

The study of graceful labellings of graphs is an active research area in graph theory. Few general results are known and constraint programming can be a useful tool in investigating new classes of graph. An existing CSP model used in previous studies is shown to be far slower than a model derived from a proof that cliques K_n are not graceful for $n > 4$. The new model is applied to instances from several classes of graph, finding some new graceful labellings and showing that some graphs, whose status was previously unknown, are not graceful. For some classes of graph, the results lead to the conjecture that only small instances of the class are graceful. The paper discusses whether it would be possible to base a proof of such a conjecture on a trace of the search to show that an instance of the class is not graceful.

2 Graceful Graphs

A labelling f of the nodes of a graph with q edges is *graceful* if f assigns each node a unique label from $\{0, 1, ..., q\}$ and when each edge xy is labelled with $|f(x) - f(y)|$, the edge labels are all different. (Hence, the edge labels are a permutation of 1, 2, ..., q.)

Figure 1 shows an example, with 25 edges and 10 nodes. The labels on the edges consist of the numbers 1 to 25, while the node labels are all different and taken from $\{0, ..., 25\}$.

Gallian [6] gives a survey of graceful graphs, i.e. graphs with a graceful labelling, and lists the graphs whose status is known; his survey is frequently updated to include new results. Graceful labellings were first defined by Rosa in 1967, although the name was introduced by Golomb [8] in 1972. Gallian lists a number of applications of labelled graphs; however, the study of graceful graphs has become an active area of research

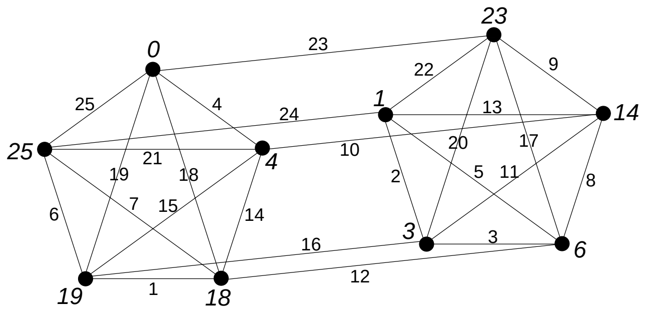


Fig. 1. A graceful labelling of the graph $K_5 \times P_2$

in its own right. The survey lists several classes of graph for which it is known that every instance is graceful, for instance the wheel graph W_n, consisting of a cycle C_n and an additional node joined to every node of the cycle. However, the only general result given by Gallian is that if every node has even degree and the number of edges is congruent to 1 or 2 (mod 4) then the graph is not graceful. For example, the cycles C_{4n+1} and C_{4n+2} are not graceful, although C_{4n} and C_{4n+3} are graceful for all n. There is a long-standing conjecture that all trees are graceful and although this has been proved for several classes of tree (including paths), and for all trees with at most 27 nodes, the general case remains unproven.

Given a graph whose gracefulness is so far unknown, in general there is no way to tell whether it is graceful or not, except by trying to label it. Constraint programming is thus a useful tool to use in investigating graceful graphs.

3 A CP Model

Constraint programming has already been applied to finding graceful labellings in a few cases; for instance, the all-interval series problem (problem 007 in CSPLib, at http://www.csplib.org) is equivalent to finding a graceful labelling of a path. Lustig & Puget [10] found a graceful labelling of the graph shown in Figure 2, which was not previously known to be graceful. Petrie and Smith [11] used graceful graphs to investigate different symmetry breaking methods in constraint programming.

In order to model the problem of finding a graceful labelling as a CSP, the nodes of the graph are numbered from 1 to n, where n is the number of nodes. Figure 2 shows this numbering of the nodes, as well as (in brackets) the label attached to each node in one of the graceful labellings.

The CSP model used in [10,11] has two sets of variables: a variable for each node, $x_1, x_2, ..., x_n$ each with domain $\{0, 1, ..., , q\}$ and a variable for each edge, $d_1, d_2, ..., d_q$, each with domain $\{1, 2, ..., , q\}$. The value assigned to x_i is the label attached to node i, and the value of d_k is the label attached to the edge k.

The constraints of the problem are: if edge k joins nodes i and j then $d_k = |x_i - x_j|$, for $k = 1, 2, ..., q$; $x_1, x_2, ..., x_n$ are all different; $d_1, d_2, ..., d_q$ are all different (and form a permutation).

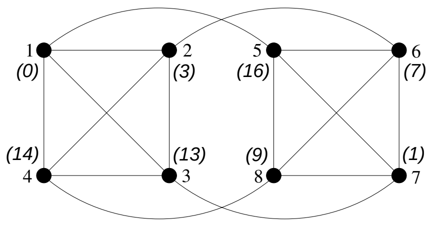


Fig. 2. The graph $K_4 \times P_2$

There are two kinds of symmetry in the problem of finding a graceful labelling of a graph: first, there may be symmetry in the graph. For instance, if the graph is a clique, any permutation of the node labels in a graceful labelling is also graceful, and if the graph is a path, P_n, the node labels can be reversed. The second type of symmetry is that we can replace the value v of every node variable x_i by its complement $q - v$. We can also combine each graph symmetry with the complement symmetry. For instance, the graceful labelling (0, 3, 1, 2) of P_4 has three symmetric equivalents: reversal (2, 1, 3, 0); complement (3, 0, 2, 1); and reversal + complement (1, 2, 0, 3).

If the graph is symmetric, it is essential to eliminate all or most of the symmetry in the CSP in order to avoid wasted search, especially if all graceful labellings are required or the graph is not graceful. In terms of the CSP model described above, the symmetry of the graph leads to variable symmetries in the CSP affecting the node variables $x_1, x_2, ..., x_n$; each symmetry maps an assignment of a value l to a variable x_i to an assignment of the same value to a variable x_j. Following the procedure introduced by Crawford *et al.* [5], variable symmetries can be eliminated by adding a lexicographic ordering constraint for each symmetry. Although in general this may not be practicable, Puget [12] showed that when there is an allDifferent constraint on the variables, the symmetry can be eliminated by at most $n - 1$ binary constraints; he used the CSP model of graceful graphs described in this section as an example.

The symmetries of the graph in Figure 2 are, firstly, any permutation of the 4-cliques which acts on both in the same way, for instance, transposing nodes 1 and 2 and simultaneously nodes 5 and 6. Secondly, the labels of the first clique (nodes 1, 2, 3, 4) can be interchanged with the labels of the corresponding nodes (nodes 5, 6, 7, 8) in the second. A possible set of constraints to eliminate the graph symmetry is $x_1 < x_2$, $x_2 < x_3$, $x_3 < x_4$ to exclude permutations within the cliques and $x_1 < x_5$, $x_1 < x_6$, $x_1 < x_7$, $x_1 < x_8$ to exclude swapping the cliques and permuting both. Since the constraints imply that $x_1 = 0$, this constraint together with $x_2 < x_3$, $x_3 < x_4$ is sufficient.

It is not easy to devise a simple constraint that will eliminate the complement symmetry in this example. It is simpler if the graph has a node that is not symmetrically equivalent to any other node; the constraint that the label of this node must be $\leq q/2$ will eliminate the complement symmetry, at least if q is odd. Eliminating the complement symmetry will be returned to later; for now, this symmetry will be ignored.

As an alternative to using symmetry-breaking constraints, a dynamic symmetry-breaking method such as Symmetry Breaking During Search (SBDS) [7] can be used; again, this will be discussed further in the context of an alternative CSP model.

The model just described, which will be called the *node model* in the rest of the paper, is adequate to find all graceful labellings of the graph in Figure 2, and allowed us to investigate slightly larger graphs as well, e.g. the related graphs $K_5 \times P_2$ and $K_6 \times P_2$. $K_5 \times P_2$ is graceful, and has a unique graceful labelling, apart from symmetric equivalents, shown in Figure 1, whereas $K_6 \times P_2$ is not graceful; these new results are now listed in Gallian's survey [6]. Results that we obtained with this model for other classes of graph are summarised in [13]. However, the search effort and run time increase rapidly with problem size, so that it is very time-consuming, as shown below, to prove that $K_7 \times P_2$, with 14 nodes and 49 edges, is not graceful; $K_8 \times P_2$ is out of reach within a reasonable time. In the next sections, a better model of the problem is introduced.

4 Gracefulness of Cliques

Beutner & Harborth [3] present a search algorithm which they use to investigate graceful labellings of "nearly complete" graphs. To illustrate their method, they present Golomb's proof that K_n is not graceful for $n > 4$ [8]:

Any gracefully labelled graph must have an edge labelled q and it must link nodes labelled 0 and q. Then we consider the edge labelled $q - 1$ (which has to exist unless the graph is just a path of length 2). It must link either 0 and $q - 1$ or 1 and q. So we must have a path $0 \leftrightarrow q \leftrightarrow 1$ or $q - 1 \leftrightarrow 0 \leftrightarrow q$. These two possibilities are symmetrically equivalent under the complement symmetry, so we can break this symmetry by arbitrarily choosing one, say $0 \leftrightarrow q \leftrightarrow 1$; hence, we need a third node labelled 1.

This gives a complete graceful labelling of K_3. To extend the labelling to K_n, $n > 3$, we next consider the edge labelled $q - 2$. This must link either 0 and $q - 2$ *or* 1 and $q - 1$ *or* 2 and q, so we need a new node labelled $q - 2$, $q - 1$ or 2. But there is already an edge labelled 1 (from 0 to 1) and both $q-1$ (from $q-1$ to q) and 2 (from 2 to 1) give another, which is not allowed. Hence, there must be nodes labelled 0, 1, $q - 2$ and q. This gives the graceful labelling of K_4, with $q = 6$, shown in Figure 3; the construction shows that it is unique, apart from the complement labelling.

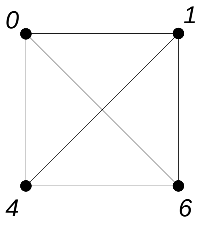


Fig. 3. The graceful labelling of K_4

In larger cliques, the next edge label to consider is $q - 4$ (there is already an edge labelled $q - 3$ from 1 to $q - 2$). The fifth node added to the clique must be labelled so that this edge label is created. The possible ways to construct an edge label $q - 4$ (in any graph) are: 0 and $q - 4$ *or* 1 and $q - 3$ *or* 2 and $q - 2$ *or* 3 and $q - 1$ *or* 4 and q. However, labelling the fifth node with any of $q - 4$, 2 and 3 gives a new edge labelled

2 ($q-4$ to $q-2$, 2 to 0, 3 to 1), so the fifth node must be labelled 4. But this gives an edge labelled $q-6$ from $q-2$ to 4; we also have an edge labelled 4, from 0 to 4, and for $n=5$, $q-6=4$. So K_5 cannot be graceful.

For $n \geq 6$, introducing a fifth node labelled 4 is allowable, so that the nodes so far are labelled 0, 1, 4, $q-2$ and q. The largest edge label not already present is $q-5$. This edge label must come from nodes labelled *either* 0 and $q-5$ *or* 1 and $q-4$ *or* 2 and $q-3$ *or* 3 and $q-2$ *or* 4 and $q-1$ *or* 5 and q. Nodes labelled $q-4$, 2 and 3 have already been ruled out. $q-5$ will give a 2nd edge labelled 3 (from $q-5$ to $q-2$); 5 will give a 2nd edge labelled 4, from 5 to 1. So we cannot have an edge labelled $q-6$ and hence K_n is not graceful for $n > 5$.

Beutner & Harborth use a backtracking search procedure, based on the approach in the proof, to find graceful labellings or prove that there are none. They focus on the edge labels, considering each label in turn, from q (the number of edges) downwards. Each edge label has to be used, and appear somewhere in the graph, and they construct a labelling, or prove that there is no labelling, by trying to decide, for each edge label in turn, which node labels will appear at either end of the edge. The classes they consider are nearly complete graphs, with specific kinds of subgraph removed from a clique; for instance, they show that $K_n - e$, i.e. an n-clique with one edge removed, is graceful only for $n \leq 5$ and that any graph derived from K_n by removing 2 or 3 edges is not graceful for $n > 6$.

5 A CSP Model Based on Edge Labels

The proof that K_n is not graceful for $n \geq 5$ can equally be adapted to modelling the problem of finding graceful labellings of a graph as a CSP. The model has a variable e_i for each edge label i, which indicates how this edge label will be formed. The value assigned to e_i is the smaller node label forming the edge labelled i, i.e. if $e_i = j$, the edge labelled i joins the nodes labelled j and $j+i$. Hence, the domain of e_i is $\{0, 1, ..., q-i\}$. (Essentially, an edge label is associated with a *pair* of node labels, but since if the edge label and the smaller node label are known, the other node label is also known, we need only associate an edge label with one node label.) Note that the domain of e_q is $\{0\}$. The domain of e_{q-1} is initially $\{0, 1\}$ but this can be reduced either to $\{1\}$ or to $\{0\}$ arbitrarily, to break the complement symmetry, as in the proof.

Once it is decided how to construct an edge label from two node labels, those node labels must be attached to specific nodes in the graph. (This is not necessary in the proof, because all nodes in a clique are equivalent.) The variable l_j represents the node to which the label j is attached, provided that this node label is used at all in the labelling. To allow for the fact that a label may not be used, the domains of these variables contain a dummy value, say $n+1$.

The link between the two sets of variables is achieved by the constraints: $e_i = j$ iff the values of l_j and l_{j+i} correspond to adjacent nodes ($1 \leq i \leq q$; $0 \leq j \leq q-i$). In Solver 6.0, this is expressed as a table constraint, using a ternary predicate which uses the adjacency matrix of the graph.

The node label variables must all have different values (unless they are assigned the dummy value). There is no need for a constraint to ensure that each edge has a different label, since an edge label is represented by a variable and so must have exactly one value, i.e. labels exactly one edge.

This new model will be called the *edge-label* model in the rest of the paper.

The complement symmetry can be eliminated by setting $e_{q-1} = 1$, as already described. (The alternative choice, $e_{q-1} = 0$, makes little difference.) However, the graph symmetry, if any, has still to be dealt with. In the edge-label model, it appears as value symmetries affecting the node-label variables. If a graph symmetry σ maps the node numbered i to the node numbered $\sigma(i)$, then it maps the assignment $l_j = i$ to the assignment $l_j = \sigma(i)$, whereas in the node model, it maps $x_i = j$ into $x_{\sigma(i)} = j$. Law and Lee [9] point out that value symmetry can be transformed into variable symmetry acting on the dual variables and then is amenable to the Crawford *et al.* procedure for deriving symmetry-breaking constraints.

Hence, a simple way to derive symmetry-breaking constraints in the edge-label model is to re-introduce the variables $x_1, ..., x_n$ from the node model. These are the duals of the node label variables $l_1, ..., l_q$. The two sets of variables can be linked by the channelling constraints: $x_i = j$ iff $l_j = i$ for $i = 1, ..., n$ and $j = 0, ..., q$, or equivalently by the global *inverse* constraint [1]. As a side-effect, the channelling constraints are sufficient to ensure that every possible node label is assigned to a different node, or else is not used, so that no explicit allDifferent constraint is required.

In the next section, where the two CSP models are compared experimentally, SBDS is also used to break the symmetry in the edge-label model; for some classes of graph, this is a better choice.

The search strategy is designed to mimic the process described in the proof of section 4, with an extra step in which specific nodes in the graph are labelled. The edge labels are considered in descending order; for each one in turn, the search first decides which node labels will make that edge label, and then decides where in the graph to put those node labels. Hence, both the edge label variables $e_1, e_2, ..., e_q$ and the node label variables $l_0, l_1, ..., l_q$ are search variables. The variable ordering strategy finds the largest edge label i that has not yet been attached to a specific edge in the graph. The next variable assigned is e_i, if that has not been assigned; if e_i has been assigned a value j, then the next variable is l_j, if that is not yet assigned, or l_{i+j}. (If all three variables have been assigned, the label i has been associated with a specific edge.)

In the next section, the performance of the edge-label model will be compared with that of the node model.

6 $K_n \times P_2$ Graphs

In this section, an experimental comparison of the two models is presented, using graphs of the class $K_n \times P_2$; these have two copies of a clique K_n with corresponding nodes of successive cliques also forming the nodes of a path P_2. The graph $K_4 \times P_2$ from this class was used in section 3 to illustrate the node model.

As before, the graph symmetry in the node model can be eliminated by the constraints $x_1 = 0; x_2 < x_3 < ... < x_n$. The complement symmetry can now be broken in the same way as in the edge-label model, by using the fact that the edge label $q - 1$ either joins nodes labelled 1 and q or nodes 0 and $q - 1$ and that these are symmetric under the complement symmetry. In the $K_n \times P_2$ graphs, it is simple to add a constraint that there is a node labelled $q-1$ adjacent to the node labelled 0, since the other symmetry-breaking constraints ensure that node 1 is the node labelled 0.

The allDifferent constraint on the node variables is treated as a set of binary $\neq$ constraints, whereas generalized arc consistency is enforced on the allDifferent constraint on the edge variables. They are treated differently because the values assigned to the edge variables form a permutation and hence the allDifferent constraint is much tighter than that on the node variables, which gives little scope for domain pruning.

In order to give a small search tree (even at the expense of increasing the run-time) generalized arc consistency is enforced on the ternary constraints $d_k = |x_i - x_j|$. A set of implied constraints are also added: for any triple of nodes i, j, k, where i is adjacent to both j and k, $2x_i \neq x_j + x_k$.

Smallest-domain variable ordering is used with the node model for this class of graph; the earlier studies [11] showed that this is not always a good choice for these problems, compared to lexicographic variable ordering, but in this case it is better. In both models, the value ordering chooses the smallest available value in the domain.

In the edge-label model, as described in the last section, the graph symmetry can be broken by the symmetry-breaking constraints on the node variables $x_1, x_2, ..., x_n$. For this class of problems, as will be seen, it is also worthwhile to use SBDS: the symmetry permuting the nodes within each cliques can be broken using SBDS and the symmetry between cliques by the constraint that the smallest node label in the first clique is less than the smallest in the second clique, i.e. $\min(x_1, x_2, ..., x_n) < \min(x_{n+1}, x_{n+2}, ..., x_{2n})$. This constraint still allows the nodes within the cliques to be freely permuted and so is compatible with SBDS breaking the symmetry within cliques. This symmetry is easy to deal with in SBDS: since it allows all permutations of the nodes in a clique (while simultaneously permuting the corresponding nodes in the other clique), it is only necessary to input the transpositions of pairs of nodes. For instance, one such transposition swaps the assignments to x_i and x_j $(1 \leq i, j \leq n)$ and also to x_{i+n} and x_{j+n}.

The models were implemented in ILOG Solver 6.0, and all experiments reported in this paper were run on a 1.7GHz Pentium M PC. The results for the two models, and both symmetry-breaking methods used in the edge-label model, are given in Table 1.

Clearly, the new model is far better than the original model for these problems. With hindsight, the domains of the node variables in the original model are too large to allow efficient search (and they increase rapidly with the size of the problem). In the edge-label model, the initial domain sizes of the edge-label variables, in the order in which they are assigned, are always $1, 2, ..., q$. In practice, domain pruning ensures that by the time an edge label variable is reached during the search, there are very few possible values left in its domain $-$ often no more than two.

For this class of graph, symmetry breaking constraints give significantly worse performance than SBDS, with the edge-label model. It is known that symmetry-breaking constraints can conflict with variable ordering heuristics. In the edge-label model, the symmetry-breaking constraints are posted on the node variables, but these are not search variables. The order in which the node variables are assigned is hard to predict, and it is difficult to see how the symmetry-breaking constraints can be tailored to be compatible

Table 1. Comparison of CSP models for finding all graceful labellings of $K_n \times P_2$ graphs, using Solver 6.0 on a 1.7GHz Pentium M PC

		Node model (with symmetry-breaking constraints)		Edge label model: Symmetry-breaking constraints		Edge label model: Breaking symmetry using SBDS	
Graph	Solutions	backtracks	time (sec.)	backtracks	time (sec.)	backtracks	time (sec.)
$K_3 \times P_2$	4	37	0.01	18	0.01	20	0.03
$K_4 \times P_2$	15	499	0.15	198	0.21	172	0.21
$K_5 \times P_2$	1	9,350	5.55	1,252	6.28	866	3.84
$K_6 \times P_2$	0	192,360	220.95	4,151	109.50	1,861	42.53
$K_7 \times P_2$	0	3,674,573	8,433.96	10,635	739.53	2,440	169.81
$K_8 \times P_2$	0	-	-	23,048	3,301.41	2,553	351.32
$K_9 \times P_2$	0	-	-	-	-	2,561	715.21
$K_{10} \times P_2$	0	-	-	-	-	2,561	1,314.71

with it. Because of the way that the two cliques are linked, it seems that early in the search labels are assigned in the second clique in a way that creates conflicts with the symmetry-breaking constraints when the corresponding nodes in the first clique are labelled later in search; hence, this leads the search into backtracking to try to resolve these conflicts. In the rest of the paper, other classes of graph containing multiple cliques are considered, but in these cases, the cliques are much less interlinked, and symmetry-breaking constraints do not encounter the same difficulties. Since symmetry-breaking constraints are easy to post, break all the symmetries in these problems, and offer the potential to prune variable domains early, they are a better choice than SBDS for these graphs. However, it is worth remembering that conflict with the variable ordering can sometimes cause them to do very poorly.

For $n \geq 9$, it appears from Table 1 that the size of the search tree explored by the edge-label model with SBDS no longer increases with n. This has also been observed in other classes of graph and will be discussed further in the next section. The runtime continues to increase with n because increasing the size of the problem means more variables and larger domain sizes, and hence constraint propagation is more time-consuming.

Cliques and nearly complete graphs with n nodes are graceful only for small values of n. Gallian lists a few other families of graphs involving cliques that are known not to be graceful; for instance, 'windmill' graphs $K_n^{(m)}$ consisting of m copies of K_n with a common vertex are not graceful for $m = 2$. Intuitively, these graphs have too many edges compared to the number of nodes to be graceful, or else this eventually happens as n increases. In the light of this and the results in Table 1, it seems reasonable to conjecture that $K_n \times P_2$ is not graceful for $n > 5$. In the next section, other graph families based on cliques will be considered which exhibit similar behaviour.

7 Sets of Overlapping Cliques

It is clearly possible to use constraint programming to show that some graphs are graceful and that some graphs are not. What would be required to show that a *class* of graph is not graceful (perhaps from some point onwards)?

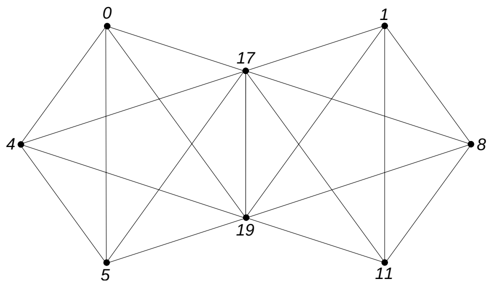


Fig. 4. The graph $B(5,2,2)$ with its unique graceful labelling (apart from symmetric equivalents)

Table 2. Search effort with the new model to find all graceful labellings of graphs $B(n,2,2)$ or prove that there are none

Graph	Solutions	backtracks	time (sec.)
$B(3,2,2)$	4	1	0.01
$B(4,2,2)$	4	22	0.01
$B(5,2,2)$	1	80	0.14
$B(6,2,2)$	0	116	1.72
$B(7,2,2)$	0	124	6.88
$B(8,2,2)$	0	124	15.26
$B(9,2,2)$	0	124	29.78

If we consider cliques K_n and the proof given earlier that they are graceful only for $n \leq 4$, the proof demonstrates that when $n \geq 6$, it is impossible to construct an edge labelled $q-5$, where q is the number of edges, given that the edge labels $q, q-1, .., q-4$ already exist. (The proof in the case $n = 5$ is slightly shorter, since in that case the label $q-4$ cannot be constructed.) If we applied this proof on an instance-by-instance basis, then when $n = 6$, it would be impossible to construct the edge labelled 11 (q=15); when n = 7, it would be impossible to construct the edge labelled 16 (q = 21); and so on. However, the proof for each instance would be essentially the same.

The search strategy used with the new model is based on the structure of Golomb's proof. This suggests that perhaps a proof that some class of graphs is not graceful from some point onwards could be constructed from the searches for individual instances that prove that each instance has no solution. For this to be possible, a first requirement is that the search is essentially the same for each instance, from some instance onwards on. And a first indication that the searches are essentially the same is that the number of backtracks is the same. In this section, this idea is explored, using a class of graphs that, like the $K_n \times P_2$ graphs of the last section, contain multiple copies of a clique.

Gallian [6] summarises what is known about the gracefulness or otherwise of the class of graph denoted by $B(n,r,m)$, consisting of m copies of K_n with a K_r in common ($n \geq r$). The case $r = 1$ is the same as the windmill graphs $K_n^{(m)}$. For $r > 1$, the only graceful graphs in this class listed in Gallian's survey are: $n = 3; r = 2; m \geq 1$; $n = 4; r = 2; m \geq 1$; $n = 4; r = 3; m \geq 1$.

Using the edge-label model, it is easy to add to the list of graceful instances: for example, the graph $B(5, 2, 2)$ is graceful, with a unique graceful labelling (except for symmetric equivalents) shown in Figure 4. It can also be shown that $B(n, 2, 2)$ is not graceful for $n = 6, 7, 8, 9$, which according to Gallian's survey was not previously known. These results by themselves suggest that $B(n, 2, 2)$ is not graceful for $n \geq 6$, and indeed, a result by Bermond and Farhi [2] shows that $B(n, 2, 2)$ is not graceful for $n \geq 9$. Results for the model applied to these instances is shown in Table 2; the model is exactly as used for Table 1, except that only symmetry-breaking constraints are used and not SBDS.

It is notable that the number of backtracks is constant for n= 7, 8, 9; this suggests that the same search tree is being explored for these instances. The number of assignments made during the search to each edge variable e_{q-i} is also the same in all three searches, making allowance for the fact that q is different in each case. If it can be shown the searches are in some sense equivalent for these three instances, and it can be assumed that they will continue to be equivalent for larger values of n, then a trace of the search could form the outline of a new proof, based on constraint programming, that these graphs are not graceful for $n \geq 7$.

In ILOG Solver, the user can define trace functions to execute after various events during the search, for instance when a value is about to be removed from the domain of a variable, when a value has just been removed from the domain of a variable, or when a variable has been set. These functions can be used to extract an instance-independent account of the search from the individual searches for the instances $n = 7, 8, 9$. The first step is to output any reference to an edge or node label $> q/2$ in terms of q rather than a specific value. For instance, since $q = 41, 55, 71$ when $n = 7, 8, 9$, the values 40, 54 and 70 respectively are all output as $q - 1$.

Most of the constraint propagation events occurring during the search are not relevant to its eventual failure. To tailor the trace output to try to produce a clear account of the search, the relevant events were defined.

For this class of graph, there are three symmetrically distinct sets of nodes: those in the first clique that are not in the common edge; those in the second clique that are not in the common edge; and the two nodes in the common edge. (Although the two cliques are indistinguishable in the graph, they are distinguished in the CSP by the symmetry-breaking constraints.) The three sets of nodes are denoted by K_n^a, K_n^b and K_2 respectively in the trace output.

The only relevant events affecting the node label variable n_i are: the node label must be used (because it is required to form an edge label); it cannot be used (because it would create a duplicate edge label); it must or must not be assigned to a node in one of the three sets (denoted for instance as $n_i \in K_n^a$ in the trace output).

The only events relating to edge labels that are output are those in which the value of an edge-label variable is set; this is output as $e_i = (j, j + i)$. This can happen either as the result of a search choice, or because two node labels have been assigned to adjacent nodes and created this edge label, or because all other values have been removed from the domain of the variable. The last event is the most interesting because it can lead to further constraint propagation. A value j is removed from the domain of an edge variable e_i either because one or both of the node labels j and $j + i$ cannot be used;

or because one of the node labels is already assigned to a node in K_n^a or K_n^b and the other label cannot be assigned to an adjacent node; or because one node label is already assigned to a node in K_n^a and the other to a node in K_n^b.

As well as using the Solver trace facilities to output details of the relevant constraint propagation events, a summary of the state of the search is output whenever the variable ordering heuristic chooses the next variable to assign. The variable assignments made so far are listed, as well as any node label variables that must be used or cannot be in one of the sets of nodes. The search follows the Solver default of making binary choices, either assigning a value to a variable ($var = val$) or not assigning that value to that variable ($var \neq val$). On each branch, the relevant choice defining the branch is output.

The trace information is output as Latex; a small example is shown in Table 3. This fragment was obtained by solving the subproblem created by labelling the nodes in the common edge 0 and q and insisting that the edge labelled $q-2$ joins nodes labelled 2 and q (in the instance $n = 7$). It takes only three backtracks to prove that the subproblem has no solution, compared to 124 for the complete problem. The original output has been further reduced by about half by removing all the output relating to propagation between successive choice point on the same branch: the summary of the state of the search when each choice is made presents the same information more clearly. This leaves the output relating to the constraint propagation on each branch between the last choice point and the point where the search backtracks. For clarity, a few propagation events that were not relevant to the failure have also been removed and a brief explanation of each failure (in italics) is given; these last two changes have obviously required manual intervention, but have been done in order to explore what would be needed to explain the failure reasonably comprehensibly.

In the complete search traces for the three instances, the choices made and so the search tree explored are identical (apart from the value of q). The smallest edge label $> q/2$ whose value is set during the search is $q - 20$ and the largest edge label $< q/2$ is 12; the smallest edge label set at a choice point is $q-9$, but smaller edge labels are set by constraint propagation. The traces do differ very slightly from instance to instance because events can be added to the propagation queue in a different order.

The trace given in Table 3 does not unfortunately give as clear an explanation of each failure as would be desirable. One difficulty is that the list of events preceding a failure does not necessarily appear in a logical order from the point of view of explaining it. Nevertheless, the information that is output before each failure is sufficient to be able to understand (with a little effort) why the previous choice, given the state of the search at that point, led to a failure.

It is worth noting that the proof given by Beutner and Harborth [3] that $K_n - e$ is graceful only if $n \leq 5$ is based on their search procedure but is given in a very compressed format (which reduces the proof given in section 4 to just five lines). The proof gives the structure of the search tree, with the choices available at each choice point. It lists the assignments made to node labels during the search (whether by choice or as the result of another assignment) and the largest edge label not yet used. It differs significantly from Table 3 in that when the next largest edge label cannot be constructed, given the choices already made, it is simply recorded that the search backtracks at that point, without any further explanation. Although it is possible to reconstruct the reason

Table 3. Search to demonstrate that there is no labelling of $B(n, 2, 2)$ with the common edge labelled 0 and q and the edge label $q - 2$ formed from node labels 2 and q

```
Edge labels set: $e_1 = (0, 1)$; $e_2 = (0, 2)$; $e_3 = (q - 3, q)$;
$e_{q-3} = (0, q - 3)$; $e_{q-2} = (2, q)$; $e_{q-1} = (1, q)$; $e_q = (0, q)$;
Node labels set: $n_0 \in K_2$; $n_1 \in K_n^a$; $n_2 \in K_n^b$ ; $n_q \in K_2$;
Node labels partly set: $n_4 \notin K_n^b$; $n_5 \notin K_n^b$;
Node labels not used: 3; 4; $q - 2$; $q - 1$;
Choose $n_{q-3} \in K_n^a$
    Edge labels set: $e_1 = (0, 1)$; $e_2 = (0, 2)$; $e_3 = (q - 3, q)$;
    $e_{q-4} = (1, q - 3)$; $e_{q-3} = (0, q - 3)$; $e_{q-2} = (2, q)$; $e_{q-1} = (1, q)$; $e_q = (0, q)$;
    Node labels set: $n_0 \in K_2$; $n_1 \in K_n^a$; $n_2 \in K_n^b$ ; $n_{q-3} \in K_n^a$; $n_q \in K_2$;
    Node labels partly set: $n_5 \notin K_n^b$; $n_{q-6} \notin K_n^a$; $n_{q-5} \notin K_n^a$
    Node labels not used: 3; 4; $q - 4$; $q - 2$; $q - 1$;
    Choose $e_{q-5} = (0, q - 5)$
        $n_5 \notin K_n^a$; $n_5$ is not used
        $n_{q-5}$ must be used; $n_{q-5} \in K_n^b$
        $e_{q-7} = (2, q - 5)$
        $e_5 = (q - 5, q)$
        $n_{q-6}$ is not used
        $n_{q-8}$ is not used
        $n_{q-7}$ is not used
        $e_{q-6} = (6, q)$
        $n_7$ is not used
        $n_6 \notin K_n^a$
        $e_{q-8} = (8, q)$
        $n_8$ must be used; $n_8 \in K_n^a$
        $n_6$ must be used; $n_6 \in K_n^b$
        $e_{q-9} = (0, q - 9)$
        $n_{q-9}$ must be used
        $e_6 = (0, 6)$
        $e_4 = (2, 6)$
        *fail:* $n_{q-9} \in K_n^a$ *gives a 2nd edge labelled 6 from* $(q - 9, q - 3)$
        $n_{q-9} \in K_n^b$ *gives a 2nd edge labelled 4 from* $(q - 9, q - 5)$ $\diamond$
    Choose $e_{q-5} \neq (0, q - 5)$
        $e_{q-5} = (5, q)$
        $n_5$ must be used; $n_5 \in K_n^a$
        $n_{q-5}$ is not used
        $e_4 = (1, 5)$
        $n_6$ is not used
        $n_{q-6}$ is not used
        $e_{q-6} = (0, q - 6)$
        *fail:* $q - 6$ *is not used* $\diamond$
Choose $n_{q-3} \notin K_n^a$
    $n_{q-3} \in K_n^b$
    $e_{q-5} = (2, q - 3)$
    $n_{q-4}$ must be used; $n_{q-4} \in K_n^a$
    *fail:* $(q - 4, 1)$ *gives a 2nd edge labelled* $q - 5$ $\diamond$
```

for the failure from the previous choices, it requires significant work. Their proof, based on a search with 13 backtracks, occupies just 27 lines. Using the same format, Table 3 would occupy about six lines, and most of the trace information would be lost.

Bundy [4] discusses the acceptability (or rather the unacceptability) of large computer-generated proofs in mathematics. A proof that $B(n, 2, 2)$ is not graceful for $n > 7$ based on a search which requires 124 backtracks and in the style of Table 3 would occupy about 50 pages. However, cutting out explanatory detail does not seem to be an adequate answer. Bundy suggests breaking a long automated proof into small lemmas, where possible. In this case, we could think of each failure during the search as requiring a lemma that the choices made along this branch of the search tree cannot lead to a graceful labelling. Hence, Table 3 could be rewritten as three lemmas, one for each failure, embedded within a structure reflecting the search tree.

8 Larger Classes

The edge-label model has been used to investigate other graphs in the general $B(n, r, m)$ class. The graph $B(4, 3, 2)$, i.e. two copies of K_4 with a common triangle, was already known to be graceful; the result in [2] shows that $B(n, 3, m)$ is not graceful for $n \geq 11$ if $m \geq 2$. With the edge-label model, it can be shown that $B(n, 3, 2)$ is also not graceful for $n = 6, 7, 8, 9, 10$. The search to prove this for the last three instances takes 143 backtracks in each case, suggesting that, as before, these searches are essentially identical. $B(4, 3, 3)$ was already known to be graceful. $B(5, 3, 3)$ is also graceful, with 5 possible distinct labellings, of which one is shown in Figure 5; this is a new result. For larger values of n, $B(n, 3, 3)$ is not graceful for $n = 6$ and 7. The search effort to prove that an instance has no graceful labelling is still increasing rapidly with n at these problem sizes: for $n = 7$, the search takes 5,944 backtracks. One might expect that the search effort may exhibit similar behaviour to that shown in Table 2, i.e. that at some point the search effort reaches a plateau and appears not to increase further.

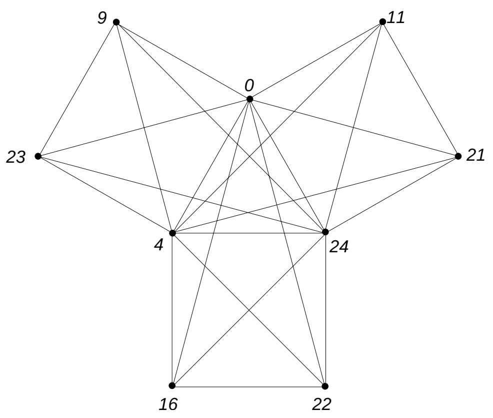


Fig. 5. The graph $B(5, 3, 3)$ with one of its graceful labellings

However, even if an acceptable proof can be based on a search which requires 100 or so backtracks, it seems doubtful that this can be done for a search requiring thousands.

9 Conclusions

A CSP model for finding a graceful labelling of a graph or proving that it is not graceful has been presented, based on constructing the edge labels in descending order. This model is far more efficient than the previous model based on assigning a label to each node. This demonstrates again the importance of having a good model of the problem to be solved, and also that what seems a natural CSP model may not prove to be a good model. The new model in this case was developed not by reformulating the existing model, but by adopting an approach from the graph theory literature [8]; non-CP approaches to solving a problem may be a good source of modelling ideas in other cases.

The search strategy used with the edge-label model has already been used by Beutner and Harborth [3] in a special-purpose algorithm for finding graceful labellings. The main advantage of using this strategy in a constraint programming framework is the constraint propagation it provides; in [3], checking that a new node label does not create duplicate edge labels is only done after the node label is assigned, whereas in solving the CSP, it can be determined in advance that the node label is not consistent with the assignments already made. Furthermore, a trace of the constraint propagation provides reasons for a branch of the search to fail and thus can be used to explain the search; as discussed in section 3 it may be possible in some cases to use trace information as the basis for a proof that a class of graphs is ungraceful.

The edge-label model has so far been applied to classes of graph which contain multiple copies of the clique K_n. Some new results have been obtained; graceful labellings have been found for $B(5, 2, 2)$ and $B(5, 3, 3)$, which were not previously known to be graceful, and it has been shown for the first time that several instances of the same classes are not graceful.

Future work will consider further how to construct a proof, that the graphs $B(n, 2, 2)$ and similar graphs are not graceful beyond a certain size, from the searches to prove that CSP instances have no solution. Although Bermond and Farhi [2] already dealt with the class $B(n, r, m)$, there are related graphs, such as those consisting of two unequal-sized cliques with a common edge, that are not covered by their result, but that could be amenable to a search-based proof. Currently, we are investigating a single CSP that is a relaxation of every CSP instance representing the graceful labelling of $B(n, 2, 2)$ with $n \geq 7$, and also covers the case where the cliques are not the same size. If the relaxation has no solution, then none of the instances has a solution either. This work will be described in a later paper.

The conjecture that graphs $K_n \times P_2$ are not graceful for $n > 5$ seems harder to prove. A key point in presenting the search trace given in Table 3 is that in $B(n, r, m)$ graphs, the nodes fall into three sets. The nodes within each set are indistinguishable, and this limits the number of different search events affecting each node label that need be considered. In $K_n \times P_2$ graphs, although all the nodes are initially indistinguishable, they become distinct as nodes are labelled and other nodes have different relationships to the labelled nodes; hence the number of different cases to consider is far greater.

Nevertheless, the fact that the search tree appears not to increase in size for $n \geq 9$ suggests that in theory a proof might be constructed from the search.

Other classes of graph constructed from multiple copies of cliques will also be considered; from the evidence reported in this paper, it seems likely that these will also be graceful only for small instances. Further elucidation of this type of graph would be a useful contribution to the study of graceful graphs.

Acknowledgments

This material is based in part on works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075. I am grateful to Ian Gent for helpful comments. Thanks are also due to colleagues at 4C, and especially to Nic Wilson.

References

1. N. Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. Technical Report Technical Report T2000/01, SICS, 2000.
2. J. C. Bermond and G. Farhi. Sur un problème combinatoire d'antennes en radioastronomie II. *Annals of Discrete Mathematics*, 12:49–53, 1982.
3. D. Beutner and H. Harborth. Graceful labelings of nearly complete graphs. *Results in Mathematics*, 41:34–39, 2002.
4. A. Bundy. A Very Mathematical Dilemma. *Computer Journal*, 2006. (In press).
5. J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-Breaking Predicates for Search Problems. In *Proceedings KR'96*, pages 149–159, Nov. 1996.
6. J. A. Gallian. A Dynamic Survey of Graph Labeling. *The Electronic Journal of Combinatorics*, (DS6), 2005. 9th edition.
7. I. P. Gent and B. M. Smith. Symmetry Breaking During Search in Constraint Programming. In W. Horn, editor, *Proceedings ECAI'2000, the European Conference on Artificial Intelligence*, pages 599–603, 2000.
8. S. W. Golomb. How to number a graph. In R. C. Read, editor, *Graph Theory and Computing*, pages 23–37. Academic Press, 1972.
9. Y. C. Law and J. H. M. Lee. Symmetry Breaking Constraints for Value Symmetries in Constraint Satisfaction. *Constraints*, 11:221–267, 2006.
10. I. J. Lustig and J.-F. Puget. Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming. *INTERFACES*, 31(6):29–53, 2001.
11. K. E. Petrie and B. M. Smith. Symmetry Breaking in Graceful Graphs. Technical Report APES-56-2003, APES Research Group, 2003. Available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.
12. J.-F. Puget. Breaking symmetries in all different problems. In *Proceedings of IJCAI'05*, pages 272–277, 2005.
13. B. Smith and K. Petrie. Graceful Graphs: Results from Constraint Programming. http://4c.ucc.ie/~bms/Graceful/, 2003.

A Simple Distribution-Free Approach to the Max *k*-Armed Bandit Problem

Matthew J. Streeter[1] and Stephen F. Smith[2]

[1] Computer Science Department and
Center for the Neural Basis of Cognition
[2] The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{matts, sfs}@cs.cmu.edu

Abstract. The max k-armed bandit problem is a recently-introduced online optimization problem with practical applications to heuristic search. Given a set of k slot machines, each yielding payoff from a fixed (but unknown) distribution, we wish to allocate trials to the machines so as to maximize the maximum payoff received over a series of n trials. Previous work on the max k-armed bandit problem has assumed that payoffs are drawn from *generalized extreme value* (GEV) distributions. In this paper we present a simple algorithm, based on an algorithm for the classical k-armed bandit problem, that solves the max k-armed bandit problem effectively without making strong distributional assumptions. We demonstrate the effectiveness of our approach by applying it to the task of selecting among priority dispatching rules for the resource-constrained project scheduling problem with maximal time lags (RCPSP/max).

1 Introduction

In the classical k-armed bandit problem one is faced with a set of k slot machines, each having an arm that, when pulled, yields a payoff drawn independently at random from a fixed (but unknown) distribution. The goal is to allocate trials to the arms so as to maximize the cumulative payoff received over a series of n trials. Solving the problem entails striking a balance between exploration (determining which arm yields the highest mean payoff) and exploitation (repeatedly pulling this arm).

In the max k-armed bandit problem, the goal is to maximize the *maximum* (rather than cumulative) payoff. This version of the problem arises in practice when tackling combinatorial optimization problems for which a number of randomized search heuristics exist: given k heuristics, each yielding a stochastic outcome when applied to some particular problem instance, we wish to allocate trials to the heuristics so as to maximize the maximum payoff (e.g., the maximum number of clauses satisfied by any sampled variable assignment, the minimum makespan of any sampled schedule). Cicirello and Smith (2005) show that a max k-armed bandit approach yields good performance on the resource-constrained project scheduling problem with maximum time lags (RCPSP/max).

1.1 Motivations

When solving the classical k-armed bandit problem, one can provide meaningful performance guarantees subject only to the assumption that payoffs are drawn from a bounded interval, for example $[0, 1]$. In the max k-armed bandit problem stronger distributional assumptions are necessary, as illustrated by the following example.

Example 1. There are two arms. One returns payoff $\frac{1}{2}$ with probability 0.999, and payoff 1 with probability 0.001; the other returns payoff $\frac{1}{2}$ with probability 0.995 and payoff 1 with probability 0.005. It is not known which arm is which.

Given a budget of n pulls of the two arms described in Example 1, a variety of techniques are available for (approximately) maximizing the cumulative payoff received. However, any attempt to maximize the *maximum* payoff received over n trials is hopeless. No information is gained about any of the arms until a payoff of 1 is obtained, at which point the maximum payoff cannot be improved.

Previous work on the max k-armed bandit problem has assumed that payoffs are drawn from *generalized extreme value* (GEV) distributions. A random variable Z has a GEV distribution if

$$\mathbb{P}[Z \leq z] = \exp\left(-\left(1 + \frac{\xi(z-\mu)}{\sigma}\right)^{-\frac{1}{\xi}}\right)$$

for some constants μ, $\sigma > 0$, and ξ.

The assumption that payoffs are drawn from a GEV is justified by the Extremal Types Theorem [6], which singles out the GEV as the limiting distribution of the maximum of a large number of independent identically distributed (i.i.d.) random variables. Roughly speaking, one can think of the Extremal Types Theorem as an analogue of the Central Limit Theorem. Just as the Central Limit Theorem states that the sum of a large number of i.i.d. random variables converges in distribution to a Gaussian, the Extremal Types Theorem states that the maximum of a large number of i.i.d. random variables converges in distribution to a GEV. Despite this asymptotic guarantee, we will see in §4 that the GEV is often not even an approximately accurate model of the payoff distributions encountered in practice.

In this work, we do not assume that the payoff distributions belong to any specific parametric family. In fact, we will not make any formal assumptions at all about the payoff distributions, although (as shown in Example 1) our approach cannot be expected to work well if the distributions are chosen adversarially. Roughly speaking, our approach will work best when the following two criteria are satisfied.

1. There is a (relatively low) threshold $t_{critical}$ such that, for all $t > t_{critical}$, the arm that is most likely to yield a payoff $> t$ is the same as the arm most likely to yield a payoff $> t_{critical}$. Call this arm i^*.

2. As t increases beyond $t_{critical}$, there is a growing gap between the probability that arm i^* yields a payoff $> t$ and the corresponding probability for other arms. Specifically, if we let $p_i(t)$ denote the probability that the i^{th} arm returns a payoff $> t$, the ratio $\frac{p_{i^*}(t)}{p_i(t)}$ should increase as a function of t for $t > t_{critical}$, for any $i \neq i^*$.

Figure 1 illustrates a set of two payoff distributions that satisfy these assumptions.

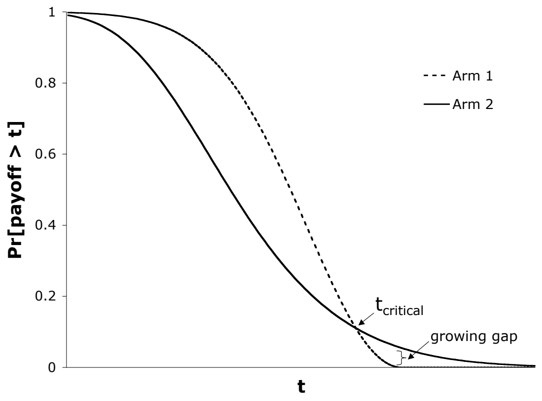


Fig. 1. A max k-armed bandit instance on which Threshold Ascent should perform well

1.2 Contributions

The primary contributions of this paper are as follows.

1. We present a new algorithm, Chernoff Interval Estimation, for the classical k-armed bandit problem and prove a bound on its regret. Our algorithm is extremely simple and has performance guarantees competitive with the state of the art.
2. Building on Chernoff Interval Estimation, we develop a new algorithm, Threshold Ascent, for solving the max k-armed bandit problem. Our algorithm is designed to work well as long as the two mild distributional assumptions described in §1.1 are satisfied.
3. We evaluate Threshold Ascent experimentally by using it to select among randomized priority dispatching rules for the RCPSP/max. We find that Threshold Ascent (a) performs better than any of the priority rules perform in isolation, and (b) outperforms the recent QD-BEACON max k-armed bandit algorithm of Cicirello and Smith [4,5].

1.3 Related Work

The classical k-armed bandit problem was first studied by Robbins [11] and has since been the subject of numerous papers; see Berry and Fristedt [3] and

Kaelbling [7] for overviews. We give a more detailed discussion of related work on the classical k-armed bandit problem as part of our discussion of Chernoff Interval Estimation in §2.2

The max k-armed bandit problem was introduced by Cicirello and Smith [4,5], whose experiments with randomized priority dispatching rules for the RCPSP/max form the basis of our experimental evaluation in §4. Cicirello and Smith show that their max k-armed bandit problem yields performance on the RCPSP/max that is competitive with the state of the art. The design of Cicirello and Smith's heuristic is motivated by an analysis of the special case in which each arm's payoff distribution is a GEV distribution with shape parameter $\xi = 0$. Streeter and Smith [13] provide a theoretical treatment of the max k-armed bandit problem under the assumption that each payoff distribution is a GEV.

2 Chernoff Interval Estimation

In this section we present and analyze a simple algorithm, Chernoff Interval Estimation, for the classical k-armed bandit problem. In §3 we use this approach as the basis for Threshold Ascent, an algorithm for the max k-armed bandit problem.

In the classical k-armed bandit problem one is faced with a set of k arms. The i^{th} arm, when pulled, returns a payoff drawn independently at random from a fixed (but unknown) distribution. All payoffs are real numbers between 0 and 1. We denote by μ_i the expected payoff obtained from a single pull of arm i, and define $\mu^* = \max_{1 \leq i \leq k} \mu_i$. We consider the finite-time version of the problem, in which our goal is to maximize the cumulative payoff received using a fixed budget of n pulls. The *regret* of an algorithm (on a particular instance of the classical k-armed bandit problem) is the difference between the cumulative payoff the algorithm would have received by pulling the single best arm n times and the cumulative payoff the algorithm actually received.

```
Procedure **ChernoffIntervalEstimation**($n$, $\delta$):
  1. Initialize $x_i \leftarrow 0$, $n_i \leftarrow 0$ $\forall i \in \{1, 2, \ldots, k\}$.
  2. Repeat $n$ times:
     (a) $\hat{i} \leftarrow \arg\max_i U(\bar{\mu}_i, n_i)$, where $\bar{\mu}_i = \frac{x_i}{n_i}$ and

           $U(\mu, n) = \begin{cases} \mu + \frac{\alpha + \sqrt{2n\mu\alpha + \alpha^2}}{n} & \text{if } n > 0 \\ \infty & \text{otherwise} \end{cases}$

         where $\alpha = \ln\left(\frac{2nk}{\delta}\right)$.
     (b) Pull arm $\hat{i}$, receive payoff $R$, set $x_{\hat{i}} \leftarrow x_{\hat{i}} + R$, and set
         $n_i \leftarrow n_i + 1$.
```

Chernoff Interval Estimation is simply the well-known interval estimation algorithm [7,8] with confidence intervals derived using Chernoff's inequality.

Although various interval estimation algorithms have been analyzed in the literature and a variety of guarantees have been proved, both (a) our use of Chernoff's inequality in an interval estimation algorithm and (b) our analysis appear to be novel. In particular, when the mean payoff returned by each arm is small (relative to the maximum possible payoff) our algorithm has much better performance than the recent algorithm of [1], which is identical to our algorithm except that confidence intervals are derived using Hoeffding's inequality. We give further discussion of related work in §2.2.

2.1 Analysis

In this section we put a bound on the expected regret of Chernoff Interval Estimation. Our analysis proceeds as follows. Lemma 1 shows that (with a certain minimum probability) the value $U(\bar{\mu}_i, n_i)$ is always an upper bound on μ_i. Lemma 2 then places a bound on the number of times the algorithm will sample an arm whose mean payoff is suboptimal. Theorem 1 puts these results together to obtain a bound on the algorithm's expected regret.

We will make use of the following inequality.

Chernoff's inequality. *Let $X = \sum_{i=1}^{n} X_i$ be the sum of n independent identically distributed random variables with $X_i \in [0,1]$ and $\mu = \mathbb{E}[X_i]$. Then for $\beta > 0$,*

$$\mathbb{P}\left[\frac{X}{n} < (1-\beta)\mu\right] < \exp\left(-\frac{n\mu\beta^2}{2}\right)$$

and

$$\mathbb{P}\left[\frac{X}{n} > (1+\beta)\mu\right] < \exp\left(-\frac{n\mu\beta^2}{3}\right) .$$

We will also use the following easily-verified algebraic fact.

Fact 1. *If $U = U(\mu, n)$ then*

$$Un\left(1 - \frac{\mu}{U}\right)^2 = 2\alpha .$$

Lemma 1. *During a run of ChernoffIntervalEstimation(n, δ) it holds with probability at least $1 - \frac{\delta}{2}$ that for all arms $i \in \{1, 2, \ldots, k\}$ and for all n repetitions of the loop, $U(\bar{\mu}_i, n_i) \geq \mu_i$.*

Proof. It suffices to show that for any arm i and any particular repetition of the loop, $\mathbb{P}[U(\bar{\mu}_i, n_i) < \mu_i] < \frac{\delta}{2nk}$. Consider some particular fixed values of μ_i, α, and n_i, and let μ_c be the largest solution to the equation

$$U(\mu_c, n_i) = \mu_i \tag{1}$$

By inspection, $U(\mu_c, n_i)$ is strictly increasing as a function of μ_c. Thus $U(\bar{\mu}_i, n_i) < \mu_i$ if and only if $\bar{\mu}_i < \mu_c$, so $\mathbb{P}[U(\bar{\mu}_i, n_i) < \mu_i] = \mathbb{P}[\bar{\mu}_i < \mu_c]$. Thus

$$\begin{aligned}
\mathbb{P}\left[U(\bar{\mu}_i, n_i) < \mu_i\right] &= \mathbb{P}\left[\bar{\mu}_i < \mu_c\right] \\
&= \mathbb{P}\left[\bar{\mu}_i < \mu_i\left(1 - \left(1 - \frac{\mu_c}{\mu_i}\right)\right)\right] \\
&< \exp\left(-\frac{\mu_i n_i}{2}\left(1 - \frac{\mu_c}{\mu_i}\right)^2\right) \\
&= \exp\left(-\alpha\right) \\
&= \frac{\delta}{2nk}
\end{aligned}$$

where on the third line we have used Chernoff's inequality, and on the fourth line we have used Fact 1 in conjunction with (1). □

Lemma 2. *During a run of ChernoffIntervalEstimation*(n, δ) *it holds with probability at least* $1 - \delta$ *that each suboptimal arm* i *(i.e., each arm* i *with* $\mu_i < \mu^*$*) is pulled at most* $\frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y_i})^2}$ *times, where* $y_i = \frac{\mu_i}{\mu^*}$.

Proof. Let i^* be some optimal arm (i.e., $\mu_{i^*} = \mu^*$) and assume that $U(\bar{\mu}_{i^*}, n_{i^*}) \geq \mu^*$ for all n repetitions of the loop. By Lemma 1, this assumption is valid with probability at least $1 - \frac{\delta}{2}$. Consider some particular suboptimal arm i. By inspection, we will stop sampling arm i once $U(\bar{\mu}_i, n_i) < \mu^*$. So it suffices to show that if $n_i \geq \frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y_i})^2}$, then $U(\bar{\mu}_i, n_i) < \mu^*$ with probability at least $1 - \frac{\delta}{2k}$ (then the probability that any of our assumptions fail is at most $\frac{\delta}{2} + k\frac{\delta}{2k} = \delta$). To show this we will prove two claims.

Claim. If $n_i \geq \frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y_i})^2}$, then with probability at least $1 - \frac{\delta}{2k}$, $\bar{\mu}_i < \sqrt{y_i^{-1}}\mu_i$.

Proof (of Claim 1).

$$\begin{aligned}
\mathbb{P}\left[\bar{\mu}_i > \sqrt{y_i^{-1}}\mu_i\right] &= \mathbb{P}\left[\bar{\mu}_i > \left(1 + \frac{1 - \sqrt{y_i}}{\sqrt{y_i}}\right)\mu_i\right] \\
&< \exp\left(-\frac{n_i \mu_i}{3}\frac{(1 - \sqrt{y_i})^2}{y_i}\right) \\
&= \exp\left(-\frac{n_i \mu^*}{3}(1 - \sqrt{y_i})^2\right) \\
&< \exp\left(-\alpha\right) \\
&= \frac{\delta}{2nk} < \frac{\delta}{2k} \ .
\end{aligned}$$

□

Claim. If $n_i \geq \frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y_i})^2}$ and $\bar{\mu}_i < \sqrt{y_i^{-1}}\mu_i$, then $U(\bar{\mu}_i, n_i) < \mu^*$.

Proof (of Claim 2). Let $U_i = U(\bar{\mu}_i, n_i)$, and suppose for contradiction that $U_i \geq \mu^*$. Then by Fact 1,

$$n_i = \frac{2\alpha}{U_i}\left(1 - \frac{\bar{\mu}_i}{U_i}\right)^{-2} .$$

The right hand side increases as a function of $\bar{\mu}_i$ (assuming $\bar{\mu}_i < U_i$, which is true by definition). So if $\bar{\mu}_i < \sqrt{y_i^{-1}}\mu_i$ then replacing $\bar{\mu}_i$ with $\sqrt{y_i^{-1}}\mu_i$ only increases the value of the right hand side. Similarly, the right hand side decreases as a function of U_i, so if $U_i \geq \mu^*$ then replacing replacing U_i with μ^* only increases the value of the right hand side. Thus

$$n_i < \frac{2\alpha}{\mu^*}\left(1 - \frac{\sqrt{y_i^{-1}}\mu_i}{\mu^*}\right)^{-2} = \frac{2\alpha}{\mu^*}\left(1 - \sqrt{y_i}\right)^{-2}$$

which is a contradiction. □

Putting Claims 1 and 2 together, once $n_i \geq \frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y_i})^2}$ it holds with probability at least $1 - \frac{\delta}{2k}$ that $U(\bar{\mu}_i, n_i) < \mu^*$, so arm i will no longer be pulled. □

The following theorem shows that when n is large (and the parameter δ is small), the total payoff obtained by Chernoff Interval Estimation over n trials is almost as high as what would be obtained by pulling the single best arm for all n trials.

Theorem 1. *The expected regret incurred by ChernoffIntervalEstimation*(n, δ) *is at most*

$$(1-\delta)2\sqrt{3\mu^* n(k-1)\alpha} + \delta\mu^* n$$

where $\alpha = \ln\left(\frac{2nk}{\delta}\right)$.

Proof. We confine our attention to the special case $k = 2$. The proof for general k is similar.

First, note that the conclusion of Lemma 2 fails to hold with probability at most δ. Because expected regret cannot exceed $\mu^* n$, this scenario contributes at most $\delta\mu^* n$ to overall expected regret. Thus it remains to show that, conditioned on the event that the conclusion of Lemma 2 holds, expected regret is at most $2\sqrt{3\mu^* n(k-1)\alpha}$.

Assume $\mu^* = \mu_1 > \mu_2$ and let $y = \frac{\mu_2}{\mu^*}$. By Lemma 2, we sample arm 2 at most $\min\{n, \frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y})^2}\}$ times. Each sample of arm 2 incurs expected regret $\mu^* - \mu_2 = \mu^*(1-y)$. Thus expected total regret is at most

$$\mu^*(1-y)\min\left\{n, \frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y})^2}\right\} . \tag{2}$$

Using the fact that $y < 1$,

$$\begin{aligned}\frac{1-y}{(1-\sqrt{y})^2} &= \frac{1-y}{(1-\sqrt{y})^2} \cdot \frac{(1+\sqrt{y})^2}{(1+\sqrt{y})^2} \\ &= \frac{(1+\sqrt{y})^2}{1-y} \\ &< \frac{4}{1-y} \ .\end{aligned}$$

Plugging this value into (2), the expected total regret is at most

$$\min\left\{\mu^* \Delta n, \frac{12\alpha}{\Delta}\right\}$$

where $\Delta = 1 - y$. Setting these two expressions equal gives $\bar{\Delta} = 2\sqrt{\frac{3\alpha}{n\mu^*}}$ as the value of Δ that maximizes expected regret. Thus the expected regret is at most $\mu^* \bar{\Delta} n = 2\sqrt{3\mu^* n\alpha} = 2\sqrt{3\mu^* n(k-1)\alpha}$.

□

2.2 Discussion and Related Work

Types of Regret Bounds. In comparing the regret bound of Theorem 1 to previous work, we must distinguish between two different types of regret bounds. The first type of bound describes the asymptotic behavior of regret (as $n \to \infty$) on a *fixed* problem instance (i.e., with all k payoff distributions held constant). In this framework, a lower bound of $\Omega(\ln(n))$ has been proved, and algorithms exist that achieve regret $O(\ln(n))$ [1]. Though we do not prove it here, Chernoff Interval Estimation achieves $O(\ln(n))$ regret in this framework when δ is set appropriately.

The second type of bound concerns the maximum, over all possible instances, of the expected regret incurred by the algorithm when run on that instance for n pulls. In this setting, a lower bound of $\Omega(\sqrt{kn})$ has been proved [2]. It is this second form of bound that Theorem 1 provides. In what follows, we will only consider bounds of this second form.

The Classical k-Armed Bandit Problem. We are not aware of any work on the classical k-armed bandit problem that offers a better regret bound (of the second form) than the one proved in Theorem 1. Auer et al. [1] analyze an algorithm that is identical to ours except that the confidence intervals are derived from Hoeffding's inequality rather than Chernoff's inequality. An analysis analogous to the one in this paper shows that their algorithm has worst-case regret $O(\sqrt{nk\ln(n)})$ when the instance is chosen adversarially as a function of n. Plugging $\delta = \frac{1}{n^2}$ into Theorem 1 gives a bound of $O(\sqrt{n\mu^* k\ln(n)})$, which is never any worse than the latter bound (because $\mu^* \leq 1$) and is much better when μ^* is small.

The Nonstochastic Multiarmed Bandit Problem. In a different paper, Auer et al. [2] consider a variant of the classical k-armed bandit problem in which the sequence of payoffs returned by each arm is determined adversarially in advance. For this more difficult problem, they present an algorithm called **Exp3.1** with expected regret

$$8\sqrt{(e-1)G_{\max}k\ln(k)} + 8(e-1)k + 2k\ln(k)$$

where $G_{\max}$ is the maximum, over all k arms, of the total payoff that would be obtained by pulling that arm for all n trials. If we plug in $G_{\max} = \mu^* n$, this bound is sometimes better than the one given by Theorem 1 and sometimes not, depending on the values of n, k, and μ^*, as well as the choice of the parameter δ.

3 Threshold Ascent

To solve the max k-armed bandit problem, we use Chernoff Interval Estimation to maximize the number of payoffs that exceed a threshold T that varies over time. Initially, we set T to zero. Whenever s or more payoffs $> T$ have been received so far, we increment T. We refer to the resulting algorithm as Threshold Ascent. The code for Threshold Ascent is given below. For simplicity, we assume that all payoffs are integer multiples of some known constant Δ.

```
Procedure **ThresholdAscent**($s$, $n$, $\delta$):
  1. Initialize $T \leftarrow 0$ and $n_i^R = 0$, $\forall i \in \{1,2,\ldots,k\}, R \in \{0,\Delta,2\Delta,\ldots,1-\Delta,1\}$.
  2. Repeat $n$ times:
     (a) While $\sum_{i=1}^{k} S_i(T) \geq s$ do:

                $T \leftarrow T + \Delta$

         where $S_i(t) = \sum_{R>t} n_i^R$ is the number of payoffs $> t$ received so far from arm $i$.
     (b) $\hat{i} \leftarrow \arg\max_i U\left(\frac{S_i(T)}{n_i}, n_i\right)$, where $n_i = \sum_R n_i^R$ is the number of times arm $i$ has been pulled and

                $U(\mu,n) = \begin{cases} \mu + \frac{\alpha+\sqrt{2n\mu\alpha+\alpha^2}}{n} & \text{if } n > 0 \\ \infty & \text{otherwise} \end{cases}$

         where $\alpha = \ln\left(\frac{2nk}{\delta}\right)$.
     (c) Pull arm $\hat{i}$, receive payoff $R$, and set $n_i^R \leftarrow n_i^R + 1$.
```

The parameter s controls the tradeoff between exploration and exploitation. To understand this tradeoff, it is helpful to consider two extreme cases.

Case $s = 1$. ThresholdAscent$(1, n, \delta)$ is equivalent to round-robin sampling. When $s = 1$, the threshold T is incremented whenever a payoff $> T$ is obtained. Thus the value $\frac{S_i(T)}{n_i}$ calculated in 2 (b) is always 0, so the value of $U\left(\frac{S_i(T)}{n_i}, n_i\right)$ is determined strictly by n_i. Because U is a decreasing function of n_i, the algorithm simply samples whatever arm has been sampled the smallest number of times so far.

Case $s = \infty$. ThresholdAscent(∞, n, δ) is equivalent to ChernoffIntervalEstimation (n, δ) running on a k-armed bandit instance where payoffs $> T$ are mapped to 1 and payoffs $\leq T$ are mapped to 0.

4 Evaluation on the RCPSP/max

Following Cicirello and Smith [4,5], we evaluate our algorithm for the max k-armed bandit problem by using it to select among randomized priority dispatching rules for the resource-constrained project scheduling problem with maximal time lags (RCPSP/max). Cicirello and Smith's work showed that a max k-armed bandit approach yields good performance on benchmark instances of this problem.

Briefly, in the RCPSP/max one must assign start times to each of a number of activities in such a way that certain temporal and resource constraints are satisfied. Such an assignment of start times is called a *feasible schedule*. The goal is to find a feasible schedule whose makespan is as small as possible, where makespan is defined as the maximum completion time of any activity.

Even without maximal time lags (which make the problem more difficult), the RCPSP is NP-hard and is "one of the most intractable problems in operations research" [9]. When maximal time lags are included, the feasibility problem (i.e., deciding whether a feasible schedule exists) is also NP-hard.

4.1 The RCPSP/max

Formally, an instance of the RCPSP/max is a tuple $\mathcal{I} = (\mathcal{A}, R, \mathcal{T})$, where $\mathcal{A}$ is a set of activities, R is a vector of resource capacities, and $\mathcal{T}$ is a list of temporal constraints. Each activity $a_i \in \mathcal{A}$ has a *processing time* p_i, and a resource demand $r_{i,k}$ for each $k \in \{1, 2, \ldots, |R|\}$. Each temporal constraint $T \in \mathcal{T}$ is a triple $T = (i, j, \delta)$, where i and j are activity indices and δ is an integer. The constraint $T = (i, j, \delta)$ indicates that activity a_j cannot start until δ time units after activity a_i has started.

A schedule S assigns a *start time* $S(a)$ to each activity $a \in \mathcal{A}$. S is feasible if

$$S(a_j) - S(a_i) \geq \delta \quad \forall (i, j, \delta) \in \mathcal{T}$$

(i.e., all temporal constraints are satisfied), and

$$\sum_{a_i \in A(S,t)} r_{i,k} \leq R_k \quad \forall\, t \geq 0, k \in \{1, 2, \ldots, |R|\}$$

where $A(S,t) = \{a_i \in \mathcal{A} \mid S(a_i) \leq t < S(a_i) + p_i\}$ the set of activities that are in progress at time t. The latter equation ensures that no resource capacity is ever exceeded.

4.2 Randomized Priority Dispatching Rules

A priority dispatching rule for the RCPSP/max is a procedure that assigns start times to activities one at a time, in a greedy fashion. The order in which start times are assigned is determined by a rule that assigns priorities to each activity. As noted above, it is NP-hard to generate a feasible schedule for the RCPSP/max. Priority dispatching rules are therefore augmented to perform a limited amount of backtracking in order to increase the odds of producing a feasible schedule. For more details, see [10].

Cicirello and Smith describe experiments with randomized priority dispatching rules, in which the next activity to schedule is chosen from a probability distribution, with the probability assigned to an activity being proportional to its priority. Cicirello and Smith consider the five randomized priority dispatching rules in the set $\mathcal{H} = \{LPF, LST, MST, MTS, RSM\}$. See Cicirello and Smith [4,5] for a complete description of these heuristics. We use the same five heuristics as Cicirello and Smith, with two modifications: (1) we added a form of intelligent backtracking to the procedure of [10] in order to increase the odds of generating a feasible schedule and (2) we modified the RSM heuristic to improve its performance.

4.3 Instances

We evaluate our approach on a set $\mathcal{I}$ of 169 RCPSP/max instances from the ProGen/max library [12]. These instances were selected as follows. We first ran the heuristic LPF (the heuristic identified by Cicirello and Smith as having the best performance) 10,000 times on all 540 instances from the `TESTSETC` data set. For many of these instances, LPF found a (provably) optimal schedule on a large proportion of the runs. We considered any instance in which the best makespan found by LPF was found with frequency > 0.01 to be "easy" and discarded it from the data set. What remained was a set $\mathcal{I}$ of 169 "hard" RCPSP/max instances.

For each RCPSP/max instance $I \in \mathcal{I}$, we ran each heuristic $h \in \mathcal{H}$ 10,000 times, storing the results in a file. Using this data, we created a set $\mathcal{K}$ of 169 five-armed bandit problems (each of the five heuristics $h \in \mathcal{H}$ represents an arm). After the data were collected, makespans were converted to payoffs by multiplying each makespan by -1 and scaling them to lie in the interval $[0, 1]$.

4.4 Payoff Distributions in the RCPSP/max

To motivate the use of a distribution-free approach to the max k-armed bandit problem, we examine the payoff distributions generated by randomized priority dispatching rules for the RCPSP/max. For a number of instances $I \in \mathcal{I}$, we plotted the payoff distribution functions for each heuristic $h \in \mathcal{H}$. For each distribution, we fitted a GEV to the empirical data using maximum likelihood estimation of the parameters μ, σ, and ξ, as recommended by Coles [6].

Our experience was that the GEV sometimes provides a good fit to the empirical cumulative distribution function but sometimes provides a very poor fit. Figure 2 shows the empirical distribution and the GEV fit to the payoff distribution of LPF on instances PSP129 and PSP121. For the instance PSP129, the GEV accurately models the entire distribution, including the right tail. For the instance PSP121, however, the GEV fit severely overestimates the probability mass in the right tail. Indeed, the distribution in Figure 2 (B) is so erratic that no parametric family of distributions can be expected to be a good model of its behavior. In such cases a distribution-free approach is preferable.

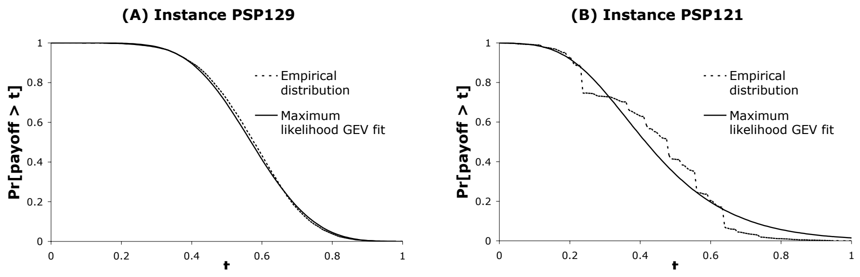


Fig. 2. Empirical cumulative distribution function of the LPF heuristic for two RCPSP/max instances. (A) depicts an instance for which the GEV provides a good fit; (B) depicts an instance for which the GEV provides a poor fit.

4.5 An Illustrative Run

Before presenting our results, we illustrate the typical behavior of Threshold Ascent by showing how it performs on the instance PSP124. For this and all subsequent experiments, we run Threshold Ascent with parameters $n = 10,000$, $s = 100$, and $\delta = 0.01$.

Figure 3 (A) depicts the payoff distributions for each of the five arms. As can be seen, LPF has the best performance on PSP124. MST has zero probability of generating a payoff > 0.8, while LST and RMS have zero probability of generating a payoff > 0.9. MTS gives competitive performance up to a payoff of $t \approx 0.9$, after which point the probability of obtaining a payoff $> t$ suddenly drops to zero.

Figure 3 (B) shows the number of pulls allocated by Threshold Ascent to each of the five arms as a function of the number of pulls performed so far. As can be seen, Threshold Ascent is a somewhat conservative strategy, allocating a fair number of pulls to heuristics that might seem "obviously" suboptimal to a human observer. Nevertheless, Threshold Ascent spends the majority of its time sampling the single best heuristic (LPF).

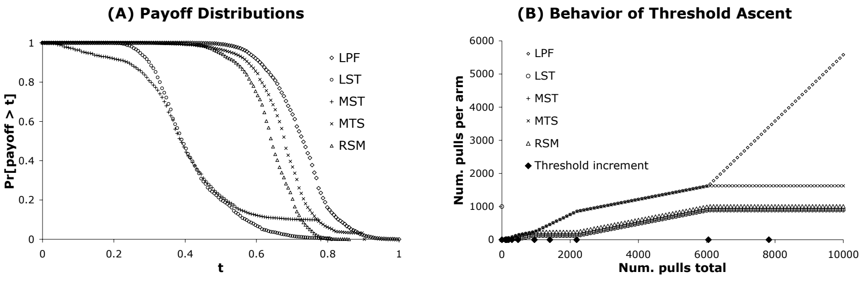


Fig. 3. Behavior of Threshold Ascent on instance PSP124. (A) shows the payoff distributions; (B) shows the number of pulls allocated to each arm.

4.6 Results

For each instance $K \in \mathcal{K}$, we ran three max k-armed bandit algorithms, each with a budget of $n = 10,000$ pulls: Threshold Ascent with parameters $n = 10,000$, $s = 100$, and $\delta = 0.01$, the QD-BEACON algorithm of Cicirello and Smith [5], and an algorithm that simply sampled the arms in a round-robin fashion. Cicirello and Smith describe three versions of QD-BEACON; we use the one based on the GEV distribution. For each instance $K \in \mathcal{K}$, we define the *regret* of an algorithm as the difference between the minimum makespan (which corresponds to the maximum payoff) sampled by the algorithm and the minimum makespan sampled by any of the five heuristics (on any of the 10,000 stored runs of each of the five heuristics). For each of the three algorithms, we also recorded the number of instances for which the algorithm generated a feasible schedule. Table 1 summarizes the performance of these three algorithms, as well as the performance of each of the five heuristics in isolation.

Table 1. Performance of eight heuristics on 169 RCPSP/max instances

Heuristic	Σ **Regret**	$\mathbb{P}$[Regret = 0]	**Num. Feasible**
Threshold Ascent	188	0.722	166
Round-robin sampling	345	0.556	166
LPF	355	0.675	164
MTS	402	0.657	166
QD-BEACON	609	0.538	165
RSM	2130	0.166	155
LST	3199	0.095	164
MST	4509	0.107	164

Of the eight max k-armed bandit strategies we evaluated (Threshold Ascent, QD-BEACON, round-robin sampling, and the five pure strategies), Threshold Ascent has the least regret and achieves zero regret on the largest number of instances. Additionally, Threshold Ascent generated a feasible schedule for the

166 (out of 169) instances for which any of the five heuristics was able to generate a feasible schedule (for three instances, none of the five randomized priority rules generated a feasible schedule after 10,000 runs).

4.7 Discussion

Two of the findings summarized in Table 1 may seem counterintuitive: the fact that round-robin performs better than any single heuristic, and the fact that QD-BEACON performs worse than round-robin. We now examine each of these findings in more detail.

Why Round-Robin Sampling Performs Well. In the classical k-armed bandit problem, round-robin sampling can never outperform the best pure strategy (where a pure strategy is one that samples the same arm the entire time), either on a single instance or across multiple instances. In the max k-armed bandit problem, however, the situation is different, as the following example illustrates.

Example 2. Suppose we have 2 heuristics, and we run them each for n trials on a set of I instances. On half the instances, heuristic A returns payoff 0 with probability 0.9 and returns payoff 1 with probability 0.1, while heuristic B returns payoff 0 with probability 1. On the other half of the instances, the roles of heuristics A and B are reversed.

If n is large, round-robin sampling will yield total regret ≈ 0, while either of the two heuristics will have regret $\approx \frac{1}{2}I$. By allocating pulls equally to each arm, round-robin sampling is guaranteed to sample the best heuristic at least $\frac{n}{k}$ times, and if n is large this number of samples may be enough to exploit the tail behavior of the best heuristic.

Understanding QD-BEACON. QD-BEACON is designed to converge to a single arm at a doubly-exponential rate. That is, the number of pulls allocated to the (presumed) optimal arm increases doubly-exponentially relative to the number of pulls allocated to presumed suboptimal arms. In our experience, QD-BEACON usually converges to a single arm after at most 10-20 pulls from each arm. This rapid convergence can lead to large regret if the presumed best arm is actually suboptimal.

5 Conclusions

We presented an algorithm, Chernoff Interval Estimation, for solving the classical k-armed bandit problem, and proved that it has good performance guarantees when the mean payoff returned by each arm is small relative to the maximum possible payoff. Building on Chernoff Interval Estimation we presented an algorithm, Threshold Ascent, that solves the max k-armed bandit problem without making strong assumptions about the payoff distributions. We demonstrated the effectiveness of Threshold Ascent on the problem of selecting among randomized priority dispatching rules for the RCPSP/max.

Acknowledgment. This work was sponsored in part by the National Science Foundation under contract #9900298 and the CMU Robotics Institute.

References

1. Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002a.
2. Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The non-stochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002b.
3. Donald. A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, London, 1986.
4. Vincent A. Cicirello and Stephen F. Smith. Heuristic selection for stochastic search optimization: Modeling solution quality by extreme value theory. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 197–211, 2004.
5. Vincent A. Cicirello and Stephen F. Smith. The max k-armed bandit: A new model of exploration applied to search heuristic selection. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1355–1361, 2005.
6. Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer-Verlag, London, 2001.
7. Leslie P. Kaelbling. *Learning in Embedded Systems*. The MIT Press, Cambridge, MA, 1993.
8. Tze Leung Lai. Adaptive treatment allocation and the multi-armed bandit problem. *The Annals of Statistics*, 15(3):1091–1114, 1987.
9. Rolf H. Möhring, Andreas S. Schulz, Frederik Stork, and Marc Uetz. Solving project scheduling problems by minimum cut computations. *Management Science*, 49(3):330–350, 2003.
10. Klaus Neumann, Christoph Schwindt, and Jürgen Zimmerman. *Project Scheduling with Time Windows and Scarce Resources*. Springer-Verlag, 2002.
11. Herbert Robbins. Some aspects of sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535, 1952.
12. C. Schwindt. Generation of resource–constrained project scheduling problems with minimal and maximal time lags. Technical Report WIOR-489, Universität Karlsruhe, 1996.
13. Matthew J. Streeter and Stephen F. Smith. An asymptotically optimal algorithm for the max k-armed bandit problem. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, 2006.

Generating Propagators for Finite Set Constraints

Guido Tack[1], Christian Schulte[2], and Gert Smolka[1]

[1] PS Lab, Saarland University, Saarbrücken, Germany
{tack, smolka}@ps.uni-sb.de
[2] ECS, ICT, KTH - Royal Institute of Technology, Sweden
cschulte@kth.se

Abstract. Ideally, programming propagators as implementations of constraints should be an entirely declarative specification process for a large class of constraints: a high-level declarative specification is automatically translated into an efficient propagator. This paper introduces the use of existential monadic second-order logic as declarative specification language for finite set propagators. The approach taken in the paper is to automatically derive projection propagators (involving a single variable only) implementing constraints described by formulas. By this, the paper transfers the ideas of indexicals to finite set constraints while considerably increasing the level of abstraction available with indexicals. The paper proves soundness and completeness of the derived propagators and presents a run-time analysis, including techniques for efficiently executing projectors for *n*-ary constraints.

1 Introduction

Implementing constraints as propagators is an essential yet challenging task in developing and extending constraint programming systems. It is essential as the system's propagators define its efficiency, correctness, maintainability, and usability. It is challenging as most systems can only be programmed at a painfully low level of abstraction, requiring the use of languages such as C++, Java, or Prolog as well as intimate knowledge of complex programming interfaces. Some approaches that address this problem are indexicals (discussed below), high-level descriptions for the range and roots constraint [5], and deriving filtering algorithms from constraint checkers [3].

For finite domain constraints, indexicals have been introduced as a high-level programming language for projectors, a restricted form of propagators that only allow projections for a single variable [13,6,8]. While indexicals simplify programming propagators considerably, they have shortcomings with respect to level of abstraction and expressiveness. Indexicals still can not be automatically obtained from purely declarative specifications: multiple implementations for the same constraint are required, such as for different levels of consistency and entailment checking. Indexicals are only expressive enough for binary or ternary constraints. For *n*-ary constraints the very idea to decompose the propagator into projectors sacrifices efficiency. For example, decomposing a linear equation constraint involving n variables into n projectors increases the run-time from $O(n)$ to $O(n^2)$.

Worse still, for more involved constraint domains such as finite sets the burden on the programmer increases. Here programmers do not only need to reason on a by-value-base but on a by-set-base (typically involving sets for lower and upper bounds). This additional complexity has been addressed by recent work of Ågren, Flener, and Pearson [2] in the context of constraint-based local search with finite set variables. The authors use formulas from an existential monadic second-order logic (∃MSO) to give descriptions of penalty functions to guide local search. While using ∃MSO as a high-level language, formulas are treated in a purely syntactic fashion: different penalty functions are obtained from syntactically different yet logically equivalent formulas.

Main contribution. This paper presents an exercise in applied software engineering. We provide a high-level, abstract specification language for finite set constraints, and an automatic translation into an efficient implementation.

The paper introduces the use of ∃MSO as a purely declarative specification language for automatically obtaining propagators for finite set constraints. As an example, consider the binary union constraint $x = y \cup z$. In the fragment of ∃MSO we define, this constraint can be written $\forall v. v \in x \leftrightarrow v \in y \vee v \in z$, where v ranges over individual values from the universe, and x, y, and z are the finite set variables. Formulas allow us to specify constraints in a *purely declarative* way.

From a constraint expressed as a formula, we derive a set of *projectors*, one for each of the variables. For the binary union example, we get $p_x = (y \cup z \subseteq x \subseteq y \cup z)$, $p_y = (x \setminus z \subseteq y \subseteq x)$, $p_z = (x \setminus y \subseteq z \subseteq x)$. Projectors transfer the idea of finite domain indexicals to finite sets. We show that the generated set of projectors is sound and complete for the constraint it was derived from.

Projectors are an *executable specification*. Just like indexicals, projectors can be implemented efficiently. We have implemented an interpreter and compiler for projectors for the Gecode library [20]. We can specify constraints purely declaratively as formulas, and then generate an implementation that is provably sound and complete.

Plan of the paper. After presenting a framework for studying propagators in Section 2, this paper is organized as follows:

- The use of ∃MSO as a declarative specification language for finite set constraints is introduced (Section 3).
- Projectors for finite set constraints are derived from range expressions over finite set variables (Section 4). This transfers the ideas of indexicals to finite set constraints.
- A correspondence between formulas in ∃MSO and finite set projectors is established (Section 5), where equivalent formulas lead to equivalent projectors. We prove that the derived projectors are correct and as strong as possible.
- Negated and reified constraints are obtained from entailment-checking projectors (Section 6).
- The specification language is shown to be also suitable for generating ROBDD-based propagators (Section 7).
- Techniques for evaluating projectors derived from range expressions are discussed, including common subexpression elimination (Section 8).
- We analyze the run-time of projectors and generalize range expressions to propagators involving arbitrary numbers of set variables (Section 9).

Section 10 concludes the paper.

2 Constraints and Propagators

This section introduces the central notions used in the rest of the paper. The presentation builds on the extended constraint systems of Benhamou [4].

Variables and constraints. We assume a finite set of variables *Var* and a finite set of values *Val*. Constraints are characterized by assignments $\alpha \in Asn$ mapping variables to values: $Asn = Var \to Val$. A constraint $c \in Con$ is a set of fulfilling assignments, $Con = \mathscr{P}(Asn)$. Basing constraints on assignments (defined for all variables *Var*) rather than tuples or partial assignments (defined for a subset of variables $X \subseteq Var$) simplifies further presentation. Note that a set of tuples or partial assignments for a set of variables X can be extended easily to a set of assignments by mapping all variables from $Var \setminus X$ to all possible values.

Domains and stores. Propagation is performed by propagators over constraint stores (or just stores) where stores are constructed from domain approximations as follows. A domain $d \in Dom$ contains values a variable can take, $Dom = \mathscr{P}(Val)$. A domain approximation A for *Dom* is a subset of *Dom* that is closed under intersection and that contains at least those domains needed to perform constraint propagation, namely $\emptyset$, *Val*, and all singletons $\{v\}$ (called *unit approximate domains* in [4]). We call elements of A approximate domains.

A store $S \in Store$ is a tuple of approximate domains. The set of stores is a Cartesian product $Store = A^n$, where $n = |Var|$ and A is a domain approximation for *Dom*.

Note that a store S can be identified with a mapping $S \in Var \to A$ or with a set of assignments $S \in \mathscr{P}(Asn)$. This allows us to treat stores as constraints when there is no risk of confusion. In particular, for any assignment α, $\{\alpha\}$ is a store.

A store S_1 is *stronger* than a store S_2, if $S_1 \subseteq S_2$. By $(c)_{Store}$ we refer to the strongest store including all values of a constraint, defined as $\min\{S \in Store | c \subseteq S\}$. The minimum exists as stores are closed under intersection. Note that this is a meaningful definition as stores only allow Cartesian products of approximate domains. Now, for a constraint c and a store S, $(c \cap S)_{Store}$ refers to removing all values from S not supported by the constraint c (with respect to the approximative nature of stores). For a constraint c, we define $c.x = \{v | \exists \alpha \in c : \alpha(x) = v\}$ as the projection on the variable x. For a store S, $S.x$ is the x-component of the tuple S.

Constraint satisfaction problems. A constraint satisfaction problem is a pair $(C,S) \in \mathscr{P}(Con) \times Store$ of a set of constraints C and a store S. A solution of a constraint satisfaction problem (C,S) is an assignment α such that $\alpha \in S$ and $\alpha \in \bigcap_{c \in C} c$.

Propagators. Propagators serve here as implementations of constraints. They are sometimes also referred to as constraint narrowing operators or filter functions. A propagator is a function $p \in Store \to Store$ that is contracting ($p(S) \subseteq S$) and monotone ($S' \subseteq S \Rightarrow p(S') \subseteq p(S)$). Note that propagators are not required to be idempotent.

A propagator is sound for a constraint c iff for all assignments α, we have $c \cap \{\alpha\} = p(\{\alpha\})$. This implies that for any store S, we have $(c \cap S)_{Store} \subseteq p(S)$. Thus, p is sound for c if it does not remove solutions and can decide if an assignment fulfills c or not.

A propagator is complete for a constraint c iff for all stores S, we have $(c \cap S)_{Store} = p(S)$. A complete propagator thus removes all assignments from S that are locally inconsistent with c. Note that a complete propagator for a constraint c establishes domain-consistency for c with respect to the domain approximation.

Fixpoints as propagation results. With set inclusion lifted point-wise to Cartesian products, stores form a complete partial order. Hence, one can show that the greatest mutual fixpoint of a set of propagators P for a store S exists. We write the fixpoint as $\bigsqcap_P S$.

In the following we will be interested in the soundness and completeness of sets of propagators. A set of propagators P is sound for a constraint c iff $\forall \alpha : c \cap \{\alpha\} = \bigsqcap_P \{\alpha\}$. Likewise, P is complete for c iff $\forall S : (c \cap S)_{Store} = \bigsqcap_P S$.

Note that we specify *what* is computed by constraint propagation and not *how*. Approaches how to perform constraint propagation can be found in [4,1,18].

Projectors. Projection propagators (or projectors, for short) behave as the identity function for all but one variable. Thus, for a projector on x, we have $\forall S\ \forall y \neq x : (p(S)).y = S.y$. To simplify presentation, we write a projector on x as $p_x \in Store \rightarrow A$.

3 A Specification Language for Finite Set Constraints

This section introduces a high-level, declarative language that allows to specify finite set constraints intensionally. Finite set constraints talk about variables ranging over finite sets of a fixed, finite universe: $Val = \mathscr{P}(\mathscr{U})$. To specify finite set constraints, we use a fragment of existential monadic second order logic (∃MSO).

In the following sections, we use ∃MSO for proving properties of projectors and the constraints they implement. Furthermore, we derive sound and complete projectors from formulas.

3.1 A Logic for Finite Set Constraints

In our framework, constraints are represented extensionally as collections of assignments. To be able to reason about them, we use formulas of a fragment of second-order logic as an intensional representation.

Finite set constraints can be elegantly stated in existential monadic second-order logic (∃MSO), see [2] for a discussion in the context of local search. Second-order variables take the role of finite set variables from *Var*, and first-order variables range over individual values from the universe $\mathscr{U}$.

We use the fragment defined by the grammar in Table 1. It has the usual Boolean connectives, a first-order universal quantifier and a second-order existential quantifier, all with their standard interpretation. As the second-order variables represent finite sets, we write $v \in x$ instead of $x(v)$ and abbreviate $\neg v \in x$ as $v \notin x$. Furthermore, we use implication ($\rightarrow$) and equivalence ($\leftrightarrow$) as the usual abbreviations.

Formulas as constraints. The extension of a formula φ is the set of models of φ. In the case of monadic second-order formulas without free first-order variables, a model corresponds directly to an assignment. The extension of φ is hence the set of assignments that satisfy φ.

Table 1. Syntax of a fragment of ∃MSO

S	$::= \exists x \ S \mid F$	*second-order quantification*
F	$::= \forall v.B \mid F \wedge F$	*first-order quantification*
B	$::= B \wedge B \mid B \vee B \mid \neg B \mid v \in x \mid \perp$	*basic formulas*

A set of satisfying (or fulfilling) assignments is exactly the definition of a constraint. Thus, for a formula φ without free first-order variables, we write $[\varphi]$ for its extension, and we use it like a constraint. Table 2 shows some examples of finite set constraints expressed as ∃MSO formulas.

Some important constraints, like disequality of sets, cannot be expressed using this logic, as they require a first-order existential quantifier. Extending the logic with existential quantification is discussed in Section 6.

Table 2. Finite set constraints expressed in ∃MSO

$x \subseteq y$	$[\forall v.v \in x \rightarrow v \in y]$	*subset*
$x = y$	$[\forall v.v \in x \leftrightarrow v \in y]$	*equality*
$x = y \cup z$	$[\forall v.v \in x \leftrightarrow v \in y \vee v \in z]$	*union*
$x = y \cap z$	$[\forall v.v \in x \leftrightarrow v \in y \wedge v \in z]$	*intersection*
$x \parallel y$	$[\forall v.v \notin x \vee v \notin y]$	*disjointness*

4 Finite Integer Set Projectors

This section introduces a high-level programming language for implementing projectors for finite set constraints. First, we define domain approximations for finite set constraints – this yields a concrete instance of the framework defined in Section 2. Then we describe range expressions as introduced by Müller [15], which carry over the ideas behind finite domain indexicals to finite set projectors.

4.1 Domain Approximations for Sets

With $Val = \mathscr{P}(\mathscr{U})$ for set variables, a full representation of the domain of a variable can be exponential in size. This makes domain approximations especially important for set variables.

Most constraint programming systems use *convex sets* as an approximation (introduced in [16], formalized in [10]). A convex set d is a set of sets that can be described by a greatest lower bound $\mathrm{glb}(d)$ and a least upper bound $\mathrm{lub}(d)$ and represents the sets $\{v \mid \mathrm{glb}(d) \subseteq v \subseteq \mathrm{lub}(d)\}$.

In our terminology, we define the domain approximation A_{Set} as the set of all convex sets. A_{Set} is indeed a domain approximation, as $\emptyset$, *Val*, and all singletons $\{s\}$ for $s \in Val$ are convex, and the intersection of two convex sets is again convex. We write $\mathrm{glb}(S.x)$ and $\mathrm{lub}(S.x)$ to denote the greatest lower resp. least upper bound of x in the store S.

Table 3. Evaluating range expressions in a store

$r_{\text{glb}}(x,S)$	$= \text{glb}(S.x)$	$r_{\text{glb}}(\overline{R},S)$	$= \overline{r_{\text{lub}}(R,S)}$
$r_{\text{glb}}(R_1 \cup R_2,S)$	$= r_{\text{glb}}(R_1,S) \cup r_{\text{glb}}(R_2,S)$	$r_{\text{glb}}(\emptyset,S) = r_{\text{lub}}(\emptyset,S)$	$= \emptyset$
$r_{\text{glb}}(R_1 \cap R_2,S)$	$= r_{\text{glb}}(R_1,S) \cap r_{\text{glb}}(R_2,S)$		
$r_{\text{lub}}(R,S)$	*analogous*		

4.2 Range Expressions for Finite Set Projectors

Given A_{Set}, a projector for a finite set variable x can be written as a function $p_x \in$ *Store* $\rightarrow$ *Val* $\times$ *Val*, returning the pruned greatest lower and least upper bound for x.

For finite domain constraints, indexicals have proven a useful projector programming language. The main idea goes back to cc(FD) [12,13] and was later elaborated in the context of clp(FD), AKL, and SICStus Prolog [9,6,8]. Indexicals build on *range expressions* as a syntax for set-valued expressions that can be used to define the projection of a constraint. These ideas easily carry over to finite set projectors over A_{Set}. We define range expressions by the following grammar:

$$R \ ::= x \mid R \cup R \mid R \cap R \mid \overline{R} \mid \emptyset$$

A finite set projector for the variable x can now be defined by two range expressions, one pruning the greatest lower bound of x, and one pruning the upper bound. We write such a projector $p_x = (R_1 \subseteq x \subseteq R_2)$.

Range expressions are evaluated in a store using the functions r_{glb} and r_{lub} from Table 3. A projector $p_x = (R_1 \subseteq x \subseteq R_2)$ is defined to compute the function $p_x(S) = (r_{\text{glb}}(R_1,S) \cup \text{glb}(S.x), r_{\text{lub}}(R_2,S) \cap \text{lub}(S.x))$.

Proposition 1. *A function $p_x = (R_1 \subseteq x \subseteq R_2)$ is contracting and monotone and therefore a propagator.*

Proof. The function p_x is contracting by construction. It is monotone iff $\forall S' \subseteq S : p_x(S') \subseteq p_x(S)$, or equivalently $\forall S' \subseteq S : \text{glb}(p_x(S)) \subseteq \text{glb}(p_x(S'))$ and $\text{lub}(p_x(S')) \subseteq \text{lub}(p_x(S))$. For a projector defined by range expressions $(R_1 \subseteq x \subseteq R_2)$, we must have $\forall S' \subseteq S : r_{\text{glb}}(R_1,S) \subseteq r_{\text{glb}}(R_1,S')$ and $r_{\text{lub}}(R_2,S') \subseteq r_{\text{lub}}(R_2,S)$. This can be shown by induction over the structure of range expressions.

5 ∃MSO Specifications for Projectors

We have seen two specification languages so far, one high-level declarative language for specifying set constraints, and one programming language for set projectors. This section shows how to connect the two languages: on the one hand, we want to find a formula describing the constraint a projector implements, on the other hand, we want to find projectors implementing the constraint represented by a formula.

We derive a ∃MSO formula φ for a given projector p_x such that p_x is sound for $[\varphi]$. The formula thus declaratively describes the constraint that p_x implements.

For the other direction, we generate a set of projectors P for a given formula φ such that P is sound and complete for $[\varphi]$. This allows us to use ∃MSO as a specification language for sound and complete finite set projectors.

Table 4. Translating a range expression to a formula

$e_v(x) = v \in x$	$e_v(\overline{R}) = \neg e_v(R)$
$e_v(R_1 \cup R_2) = e_v(R_1) \vee e_v(R_2)$	$e_v(\emptyset) = \bot$
$e_v(R_1 \cap R_2) = e_v(R_1) \wedge e_v(R_2)$	

5.1 From Range Expressions to ∃MSO Specifications

Given a projector p_x, we now derive a formula φ_{p_x} such that p_x is sound for $[\varphi_{p_x}]$.

The correspondence between relational algebra and logic gives rise to the definition of the function e (Table 4). For a range expression R, $e_v(R)$ is a formula that is true iff $v \in R$. Furthermore, a subset relation corresponds to a logical implication. Hence, for a projector $p_x = (R_1 \subseteq x \subseteq R_2)$, we define $\varphi_{p_x} = \forall v. (e_v(R_1) \rightarrow v \in x) \wedge (v \in x \rightarrow e_v(R_2))$. We can now show that p_x is sound for $[\varphi_{p_x}]$.

Lemma 1. *Range expressions have the same extension as the corresponding formulas.*

$$\alpha \in [\forall v.e_v(R) \rightarrow v \in x] \Leftrightarrow r_{\text{glb}}(R, \{\alpha\}) \subseteq \alpha(x)$$
$$\alpha \in [\forall v.v \in x \rightarrow e_v(R)] \Leftrightarrow \alpha(x) \subseteq r_{\text{lub}}(R, \{\alpha\})$$

Proof. By induction over the structure of range expressions.

Proposition 2. *Every projector $p_x = (R_1 \subseteq x \subseteq R_2)$ is sound for the constraint $[\varphi_{p_x}]$.*

Proof. We have to show $[\varphi_{p_x}] \cap \{\alpha\} = p_x(\{\alpha\})$ for all α. This is equivalent to showing $\alpha \in [\varphi_{p_x}] \Leftrightarrow p_x(\{\alpha\}) = \{\alpha\}$. From the definition of projectors, we get $p_x(\{\alpha\}) = \{\alpha\} \Leftrightarrow r_{\text{glb}}(R_1, \{\alpha\}) \subseteq \alpha(x) \wedge \alpha(x) \subseteq r_{\text{lub}}(R_2, \{\alpha\})$. Lemma 1 says that this is equivalent to $\alpha \in [\forall v.e_v(R_1) \rightarrow v \in x]$ and $\alpha \in [\forall v.v \in x \rightarrow e_v(R_2)]$. This can be combined into $\alpha \in [\forall v. (e_v(R_1) \rightarrow v \in x) \wedge (v \in x \rightarrow e_v(R_2))] = [\varphi_{p_x}]$.

Equivalence of projectors. We say that two projectors p_x and p'_x are equivalent iff they are sound for the same constraint. Using the translation to formulas as introduced above, p_x and p'_x are equivalent iff $\varphi_{p_x} \equiv \varphi_{p'_x}$. Note that two equivalent projectors may still differ in propagation strength, for instance if only one of them is complete for φ_{p_x}.

5.2 From Specifications to Projectors

The previous section shows that for every set projector p_x one can find a formula φ_{p_x} such that p_x is sound for $[\varphi_{p_x}]$. We now want to find a set of projectors P for a given formula φ such that P is sound and complete for $[\varphi]$. Remember that we need a set of projectors, as each individual projector only affects one variable. For completeness, all variable domains have to be pruned.

A first step extracts all implications to a single variable x from a given formula φ, yielding a normal form for φ. A second step then transforms this normal form into a projector for x. Using the transformation to normal form, we can construct a set of projectors, one for each variable in φ. Finally, we show that the set of projectors obtained this way is complete for $[\varphi]$.

Step 1: Extraction of Implications. As we use the convex-set approximation, a projector for a variable x has to answer two questions: which values for x occur in all assignments satisfying the constraint (the greatest lower bound), and which values can occur at all in a satisfying assignment (the least upper bound). The idea of our transformation is that the first question can be answered by a formula of the form $\forall v. \psi_1 \rightarrow v \in x$, while the second question corresponds to a formula $\forall v. v \in x \rightarrow \psi_2$. The remaining problem is hence to transform any formula φ into an equivalent $\varphi' = \bigwedge_x \forall v : (\psi_{1_x} \rightarrow v \in x) \wedge (v \in x \rightarrow \psi_{2_x})$.

The transformation proceeds in four steps: (1) Skolemization, (2) merging of first-order quantifiers, (3) transformation to conjunctive normal form, and (4) extraction of implications for each variable.

Skolemization removes all second-order existential quantifiers and replaces them with fresh variables. Intuitively, the fresh variables play the role of intermediate variables. Though this is not strictly an equivalence transformation, the extension of the resulting formula is the same for the variables of the original formula. We can *merge universal quantifiers* $(\forall v. \psi) \wedge (\forall v. \psi')$ into $\forall v. \psi \wedge \psi'$. After these transformations, we arrive at a formula of the form $\forall v. \psi$ where ψ is quantifier-free. We can then transform ψ into conjunctive normal form (CNF), into a set of *clauses* $\{C_1, \ldots, C_n\}$, where each C_i is a set of *literals* L (either $v \in x$ or $v \notin x$).

From the CNF, one can extract all implications for a variable x as follows:

$$\begin{aligned}
\forall v. \psi &\equiv \forall v. \textstyle\bigwedge_i \bigvee_{L' \in C_i} L' \\
&\equiv \forall v. \textstyle\bigwedge_i \left(\bigwedge_{L' \neq (v \in x) \in C_i} \overline{L'} \rightarrow v \in x \right) \wedge \left(\bigwedge_{L' \neq (v \notin x) \in C_i} \overline{L'} \rightarrow v \notin x \right) \\
&\equiv \forall v. \left(\textstyle\bigvee_i \bigwedge_{L' \neq (v \in x) \in C_i} \overline{L'} \rightarrow v \in x \right) \wedge \left(v \in x \rightarrow \textstyle\bigwedge_i \bigvee_{L' \neq (v \notin x) \in C_i} L' \right)
\end{aligned}$$

If ψ does not contain $v \in x$ (or $v \notin x$), the corresponding implication is trivially true. Thus, in practice, this transformation only has to consider the free variables of ψ (after Skolemization) instead of all the variables.

We call this form L-implying normal form of φ, written $INF_L(\varphi)$. We refer to $\bigvee_i (\bigwedge_{L' \neq (v \in x) \in C_i} \overline{L'})$ as ψ_{1_x} and to $\bigwedge_i \bigvee_{L' \neq (v \notin x) \in C_i} L'$ as ψ_{2_x}.

Step 2: Compilation to Projectors. The two subformulas of an L-implying normal form, ψ_{1_x} and ψ_{2_x}, are quantifier-free and contain a single first-order variable v. We observe that the function e_v from Table 4 has an inverse e^{-1} for quantifier-free formulas with a single free first-order variable v.

With Proposition 2 we can thus argue that $(e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x}))$ is sound for $[INF_x(\varphi)]$. Furthermore, for any formula $\varphi' = \bigwedge_x INF_x(\varphi)$, it is easy to show that the set $P = \{(e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x})) \mid x \in Var\}$ is sound for $[\varphi]$.

Example. Consider the ternary intersection constraint $x = y \cap z$. It can be expressed in $\exists$MSO as $\forall v. v \in x \leftrightarrow v \in y \wedge v \in z$. The implied normal forms for x, y, and z are

$\forall v.\ (v \in y \wedge v \in z \rightarrow v \in x) \ \wedge\ (v \in x \rightarrow v \in y \wedge v \in z)$
$\forall v.\ (v \in x \rightarrow v \in y) \ \wedge\ (v \in y \rightarrow v \in x \vee v \notin z)$
$\forall v.\ (v \in x \rightarrow v \in z) \ \wedge\ (v \in z \rightarrow v \in x \vee v \notin y)$

Deriving projectors from these formulas, we get $p_x = (y \cap z \subseteq x \subseteq y \cap z)$, $p_y = (x \subseteq y \subseteq x \cup \overline{z})$, and $p_z = (x \subseteq z \subseteq x \cup \overline{y})$.

Proposition 3. *We can now prove completeness of the generated projector set: Given a formula $\varphi' = \bigwedge_x INF_x(\varphi)$, the projector set $P = \{(e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x})) \mid x \in Var\}$ is complete for $[\varphi]$.*

Proof. We only sketch the proof due to space limitations. For completeness, we have to show $\bigsqcap_P S \subseteq ([\varphi] \cap S)_{Store}$. If $([\varphi] \cap S)_{Store} = S$, this is trivial. Otherwise, we only have to show that at least one projector can make a contribution.

First, we can show that if $[\varphi]$ removes values from the greatest lower or least upper bounds of variables in S, then there exists at least one x such that already $[INF_x(\varphi)]$ prunes S. More formally, $([\varphi] \cap S)_{Store} \neq S$ implies $\mathrm{glb}(([\forall v.\psi_{1_x} \to v \in x] \cap S).x) \neq \mathrm{glb}(S.x)$ or $\mathrm{lub}(([\forall v.v \in x \to \psi_{2_x}] \cap S).x) \neq \mathrm{lub}(S.x)$ for some x. This is true because of the way implications are extracted from φ.

The second step in the proof is to show that if $\mathrm{glb}(([\forall v.\psi_{1_x} \to v \in x] \cap S).x) \neq \mathrm{glb}(S.x)$, then $\mathrm{glb}(([\forall v.\psi_{1_x} \to v \in x] \cap S).x) \subseteq r_{\mathrm{glb}}(e^{-1}(\psi_{1_x}), S) \cup \mathrm{glb}(S.x)$, and dually for lub. This means that the projector for x makes a contribution and narrows the domain.

Finally, if none of the projectors can contribute for the store S, we know on the one hand that $([\varphi] \cap S)_{Store} = S$ (we have just proved that), and on the other hand that $\bigsqcap_P S = S$ (S must be a fixpoint). This concludes the proof.

With Propositions 2 and 3, it follows that the set of projectors $\{p_x = (e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x})) \mid x \in Var\}$ is sound and complete for $[\bigwedge_x INF_x(\varphi)] = [\varphi]$.

Finding small formulas. The projectors derived from two equivalent formulas $\varphi \equiv \varphi'$ are equivalent in the sense that they are sound and complete for both formulas. However, the size of their range expressions, and therefore the time complexity of propagating them (as discussed in Section 9), can differ dramatically.

The antecedents in an L-implied normal form are in disjunctive normal form (DNF). We can apply well-known techniques from circuit minimization to find equivalent minimal formulas (e.g. the Quine-McCluskey and Petrick's methods). Note that for the "standard constraints" like those from Table 2, the generated *INF* are minimal.

6 Negated and Reified Constraints

In this section, we show how to specify and execute negated and reified constraints, by transferring the ideas of entailment checking indexicals [7] to finite set projectors. Negation adds first-order existential quantifiers to the specification language.

Checking entailment of a projector. A propagator p is called *entailed* by a store S iff for all stores $S' \subseteq S$ we have $p(S') = S'$.

In the indexical scheme, entailment can be checked using anti-monotone indexicals [7]. Following this approach, we use the anti-monotone interpretation of a projector to check its entailment. For example, a sufficient condition for $p = (R_1 \subseteq x \subseteq R_2)$ being entailed is $r_{\mathrm{lub}}(R_1, S) \subseteq \mathrm{glb}(S.x)$ and $\mathrm{lub}(S.x) \subseteq r_{\mathrm{glb}}(R_2, S)$.

Negated projectors – existential quantifiers. A negated projector $\overline{p}$ can be propagated by checking entailment of p. If p is entailed, $\overline{p}$ is failed, and vice versa. Such a $\overline{p}$ is sound for $[\neg\varphi_p]$, but not necessarily complete.

We observe that this gives us a sound implementation for formulas $\neg\varphi$, where $\varphi = \forall v.\,(\psi_1 \rightarrow v \in x) \wedge (v \in x \rightarrow \psi_2)$. This is equivalent to $\neg\varphi = \exists v.\neg(\psi_1 \rightarrow v \in x \wedge v \in x \rightarrow \psi_2)$. We can thus extend our formulas with existential first-order quantification:

$$F \;::=\; \forall v.B \mid \boxed{\exists v.B} \mid F \wedge F$$

One important constraint we could not express in Section 3 was disequality of sets, $x \neq y$. Using existential quantification, this is easy: $\exists v.\neg(v \in x \leftrightarrow v \in y)$.

Reified constraints. A reified constraint is a constraint that can be expressed as a formula $\varphi \leftrightarrow b$, for a $0/1$ finite domain variable b. Exactly as for reified finite domain constraints implemented as indexicals [8], we can detect entailment and dis-entailment of φ, and we can propagate φ and $\neg\varphi$. Thus, we can reify any constraint expressible in our $\exists$MSO fragment.

7 Generating Projectors for BDD-Based Solvers

Solvers based on binary decision diagrams (BDDs) have been proposed as a way to implement full domain consistency for finite set constraints [11]. This section briefly recapitulates how BDD-based solvers work. We can then show that $\exists$MSO can also be used as a specification language for BDD-based propagators.

Domains and constraints as Boolean functions. The solvers as described by Hawkins et al. [11] represent both the variable domains and the propagators as Boolean functions.

A finite integer set s can be represented by its characteristic function: $\chi_s(i) = 1 \Leftrightarrow i \in s$. A domain, i.e. a set of sets, is a disjunction of characteristic functions.

Constraints can also be seen as Boolean functions. In fact, formulas in our $\exists$MSO fragment are a compact representation of Boolean functions. The universal quantification $\forall v$ corresponds to a finite conjunction over all v, because the set of values *Val* is finite. For instance, the formula $\forall v.v \in x \rightarrow v \in y$, modeling the constraint $x \subseteq y$, can be transformed into the Boolean function $\bigwedge_v x_v \rightarrow y_v$.

Reduced Ordered Binary Decision Diagrams. ROBDDs are a well-known data structure for storing and manipulating Boolean functions. Hawkins et al. propose to store complete variable domains and propagators as ROBDDs. Although this representation may still be exponential in size, it works well for many practical examples.

Propagation using ROBDDs also performs a projection of a constraint on a single variable, with respect to a store. The fundamental difference to our setup is the choice of domain approximation, as ROBDDs allow to use the full $A = \mathcal{P}(Dom)$.

Hawkins et al. also discuss approximations including cardinality information and lexicographic bounds [17]. These approximations can yield stronger propagation than simple convex bounds but, in contrast to the full domain representation, have guaranteed polynomial size.

From specification to ROBDD. As we have sketched above, ∃MSO formulas closely correspond to Boolean functions, and can thus be used as a uniform specification language for projectors based on both range expressions and BDDs.

Although BDDs can be used to implement the approximation based on convex sets and cardinality, our approach still has some advantages: (1) It can be used for existing systems that do not have a BDD-based finite set solver. (2) A direct implementation of the convex-set approximation may be more memory efficient. (3) Projectors can be compiled statically, and independent of the size of $\mathcal{U}$. Still, projectors based on range expressions offer the same compositionality as BDD-based projectors.

8 Implementing Projectors

The implementation techniques developed for finite domain indexicals directly carry over to set projectors. We thus sketch only briefly how to implement projectors. Furthermore, we present three ideas for efficient projector execution: grouping projectors, common subexpression elimination, and computing without intermediate results.

Evaluating projectors. The operational model of set projectors is very similar to that of finite domain indexicals. We just have to compute two sets (lower and upper bound) instead of one (the new domain).

The main functionality a projector $(R_1 \subseteq x \subseteq R_2)$ has to implement is the evaluation of $r_{\text{glb}}(R_1,S)$ and $r_{\text{lub}}(R_2,S)$. Just like indexicals, we can implement r_{glb} and r_{lub} using a stack-based interpreter performing a bottom-up evaluation of a range expression [6], or generate code that directly evaluates ranges [9,15].

Grouping projectors. Traditionally, one projector (or indexical) is treated as one propagator. Systems like SICStus Prolog [14] schedule indexicals with a higher priority than propagators for global constraints.

However, from research on virtual machines it is well known that grouping several instructions into one *super-instruction* reduces the dispatch overhead and possibly enables an optimized implementation of the super-instructions.

In constraint programming systems, propagators play the role of instructions in a virtual machine. Müller [15] proposes a scheme where the set of projectors that implements one constraint is compiled statically into one propagator.

Common Subexpression Elimination. Range expressions form a tree. A well-known optimization for evaluating tree-shaped expressions is *common subexpression elimination* (or CSE): if a sub-expression appears more than once, only evaluate it once and reuse the result afterwards.

Using CSE for range expressions has been suggested already in earlier work on indexicals (e.g. [6]). Common subexpressions can be shared both within a single range expression, between the two range expressions of a set projector, and between range expressions of different projectors.

If subexpressions are shared between projectors, special care must be taken of the order in which the projectors are evaluated, as otherwise evaluating one projector may invalidate already computed subexpressions. This makes grouping projectors a prerequisite for inter-projector CSE, because grouping fixes the order of evaluation.

Using iterators to evaluate range expressions. Bottom-up evaluation of range expressions usually places intermediate results on a stack. This imposes a significant performance penalty for finite set constraints, which can be avoided using *range iterators* [19]. Range iterators allow to evaluate range expressions without intermediate results.

A range iterator represents a set of integers implicitly by providing an interface for iterating over the maximal disjoint ranges of integers that define the set. Set variables can provide iterators for their greatest lower and least upper bound. The union or intersection of two iterators can again be implemented as an iterator. Domain operations on variables, such as updating the bounds, can take iterators as their arguments. Iterators can serve as the basis for both compiling the evaluation of range expressions to e.g. C++, and for evaluating range expressions using an interpreter.

Implementation in Gecode. We have implemented an interpreter and a compiler for finite set projectors for the Gecode library ([20], version 1.3 or higher). Both interpreter and compiler provide support for negated and reified propagators.

The interpreter is a generic Gecode propagator that can be instantiated with a set of projectors, defined by range expressions. Propagation uses range iterators to evaluate the range expressions. This approach allows to define new set propagators at run-time.

The compiler generates C++ code that implements a Gecode propagator for a set of projectors. Again, range iterators are used for the actual evaluation of range expressions.

Compiling to C++ code has three main advantages. First, the generated code has no interpretation overhead. Second, more expensive static analysis can be done to determine a good order of propagation for the grouped projectors (see [15]). Third, the generated code uses template-based polymorphism instead of virtual function calls (as discussed in [19]). This allows the C++ compiler to perform aggressive optimizations.

9 Run-Time Analysis

In this section, we analyze the time complexity of evaluating set projectors. This analysis shows that the naive decomposition of a constraint over n variables into n projectors leads to quadratic run-time $O(n^2)$. We develop an extended range expression language that allows to evaluate some important n-ary projectors in linear time $O(n)$.

The technique presented here is independent of the constraint domain. We show that the same technique can be used for indexicals over finite integer domain variables.

Run-time complexity of projectors. The time needed for evaluating a projector depends on the size of its defining range expressions. We define the *size* of a range expression R as the number of set operations (union, intersection, complement) R contains, and write it $|R|$. To evaluate a projector $p_x = (R_1 \subseteq x \subseteq R_2)$, one has to perform $|R_1| + |R_2|$ set operations. Abstracting from the cost of individual operations (as it depends on how sets are implemented), the run-time of p_x is in $O(|R_1| + |R_2|)$.

Example for an n-ary propagator. The finite set constraint $y = \bigcup_{1 \le i \le n} x_i$ can be stated as $n+1$ finite set projectors:

$p_y = (\text{glb}(x_1) \cup \cdots \cup \text{glb}(x_n) \subseteq y \subseteq \text{lub}(x_1) \cup \cdots \cup \text{lub}(x_n))$

$p_{x_i} = (\text{glb}(y) \setminus \bigcup_{j \ne i} \text{lub}(x_j) \subseteq x_i \subseteq \text{lub}(y))$ for all x_i

The range expressions in each projector have size n. Propagating all projectors once therefore requires time $O(n^2)$. If however these $n+1$ projectors are grouped, and thus their order of propagation is fixed, we can apply a generalized form of CSE in order to propagate in linear time.

Let us assume we propagate in the order $p_{x_1} \dots p_{x_n}$. Then at step p_{x_i}, we know that for all $j > i$, we have not yet changed the domain of x_j. Thus, we can use a precomputed table $\mathit{right}[i] = \bigcup_{j>i} \text{lub}(x_j)$. The other half of the union, $\mathit{left}_i = \bigcup_{j<i} \text{lub}(x_j)$, can be maintained incrementally while moving from step $i-1$ to step i. The projectors can thus be written as

$p_{x_i} = (\text{glb}(y) \setminus (\mathit{right}[i] \cup \mathit{left}_i) \subseteq x_i \subseteq \text{lub}(y))$ for all i

Computing the $\mathit{right}[i]$ requires time $O(n)$. Maintaining left_i is constant time, and each resulting projector p_{x_i} can be executed in time $O(1)$, too. This yields $O(n)$ for running all projectors once.

Indexed range expressions. We now extend range expressions so that they can be evaluated efficiently in the n-ary case, using the method sketched in the example above.

We assume that a subset of the variables $\mathit{Var}_{\text{idx}} \subseteq \mathit{Var}$ is *indexed*, such that $x_i \in \mathit{Var}_{\text{idx}}$ for all $1 \le i \le k$. We extend range expressions to *indexed range expressions*:

$$R \ ::= x \mid R \cup R \mid R \cap R \mid \overline{R} \mid \bigcup_{1 \le j \le k, j \ne i} x_j \mid \bigcap_{1 \le j \le k, j \ne i} x_j \mid \emptyset$$

To simplify presentation, we do not consider nested indexed range expressions here. A family of projectors can now be stated together as $p_{x_i} = (R_1 \subseteq x_i \subseteq R_2)$. We extend the functions r_{lub} and r_{glb} evaluating range expressions to take the index i of the projector as an argument. The functions r_{lub} and r_{glb} implement the optimization sketched above:

$r_{\text{lub}}(\bigcup_{1 \le j \le n, j \ne i} x_j, i, S) = \mathit{right}[i] \cup \mathit{left}_i$

where *right* and left_i are the “two halves” of the union as described in the example. The evaluation of the lower bound and the intersection is analogous.

From specification to indexed range expressions. In order to generate indexed range expressions from formulas as specifications, we have two options. We can either add syntactic sugar to the formulas that allows to express indexed conjunctions and disjunctions, or we search for sub-formulas of the form $\bigwedge_{i \ne j} x_i$ (and similar for disjunction).

Application to n-ary finite domain projectors. The same scheme applies to finite domain projectors. An n-ary linear equation $\sum_i x_i = c$ can be stated as

$$p_{x_i} = x_i \text{ in } c - \textstyle\sum_{j \ne i} \max(x_i) \ \dots \ c - \sum_{j \ne i} \min(x_i)$$

Again, if the p_{x_i} are evaluated in fixed order, the sums can be precomputed. As for the set projectors, this scheme allows propagation in time $O(n)$ instead of $O(n^2)$.

10 Conclusions and Future Work

We have presented two specification languages: $\exists$MSO, a high-level, purely declarative language for specifying finite set constraints, and range expressions, a programming language for implementing finite set projectors. Set projectors transfer the ideas of indexicals to the domain of finite sets.

We have captured both languages within one formal framework. On the one hand, this allows us to prove soundness and completeness for projectors with respect to constraints specified as formulas. On the other hand, we can derive sound and complete sets of projectors from constraints specified as formulas. Furthermore, we have shown that we can derive sound propagators for negated and reified constraints, and that $\exists$MSO is a suitable specification language for BDD-based finite set solvers.

With $\exists$MSO we thus have an expressive, declarative, high-level specification language for a large class of sound and complete finite set projectors, both for domain approximations using convex sets, and for complete domain representations using BDDs.

The run-time analysis we have presented shows that using plain projectors for n-ary constraints results in quadratic run-time. We have solved this problem with the help of indexed range expressions and evaluating projectors in a group, leading to linear run-time for important n-ary constraints. This result carries over to finite domain indexicals.

An implementation of finite set projectors is available in the Gecode library.

Future Work. We are currently integrating a BDD-based solver into the Gecode library. This will allow us to use the same constraint specifications with different solvers. In addition, it will ease the comparison of propagation strength as well as efficiency of different finite set solvers.

The translation from $\exists$MSO formulas to range expressions still has to be implemented. We currently conduct our experiments by translating formulas by hand.

Besides the implementation, our focus is on extensions of the logic as well as the projector language. We will add cardinality reasoning, which has proven very effective for several areas of application. An interesting further question is whether and how propagators for a domain approximation based on lexicographic bounds [17] can be derived automatically.

Acknowledgements. Christian Schulte is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953. Guido Tack is partially funded by DAAD travel grant D/05/26003. The authors thank Mikael Lagerkvist and the anonymous reviewers for comments on earlier versions of this paper.

References

1. K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. M. Ågren, P. Flener, and J. Pearson. Incremental algorithms for local search from existential second-order logic. In P. van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *LNCS*, pages 47–61. Springer, 2005.
3. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In Wallace [21], pages 107–122.

4. F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76. Springer, 1996.
5. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence constraints. In *IJCAI*, pages 60–65, Aug. 2005.
6. B. Carlson. *Compiling and Executing Finite Domain Constraints*. PhD thesis, Uppsala University, Sweden, 1995.
7. B. Carlson, M. Carlsson, and D. Diaz. Entailment of finite domain constraints. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 339–353. MIT Press, 1994.
8. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *PLILP*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
9. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
10. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
11. P. Hawkins, V. Lagoon, and P. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *J. Artif. Intell. Res. (JAIR)*, 24:109–156, 2005.
12. P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(FD). Technical report, Brown University, 1991.
13. P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):293–316, 1998.
14. Intelligent Systems Laboratory. SICStus Prolog user's manual, 3.12.1. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, 2006.
15. T. Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.
16. J.-F. Puget. PECOS: A high level constraint programming language. In *Proceedings of the first Singapore international conference on Intelligent Systems (SPICIS)*, pages 137–142, 1992.
17. A. Sadler and C. Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In Wallace [21], pages 604–618.
18. C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In Wallace [21], pages 619–633.
19. C. Schulte and G. Tack. Views and iterators for generic constraint implementations. In M. Carlsson, F. Fages, B. Hnich, and F. Rossi, editors, *Recent Advances in Constraints, 2005*, volume 3978 of *LNCS*, pages 118–132. Springer, 2006.
20. The Gecode team. Generic constraint development environment. `www.gecode.org`, 2006.
21. M. Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *LNCS*. Springer, 2004.

Compiling Finite Linear CSP into SAT

Naoyuki Tamura[1], Akiko Taga[2], Satoshi Kitagawa[2], and Mutsunori Banbara[1]

[1] Information Science and Technology Center, Kobe University, Japan
tamura@kobe-u.ac.jp
[2] Graduate School of Science and Technology, Kobe University, Japan

Abstract. In this paper, we propose a method to encode Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP) with integer linear constraints into Boolean Satisfiability Testing Problems (SAT). The encoding method is basically the same with the one used to encode Job-Shop Scheduling Problems by Crawford and Baker. Comparison $x \leq a$ is encoded by a different Boolean variable for each integer variable x and integer value a. To evaluate the effectiveness of this approach, we applied the method to Open-Shop Scheduling Problems (OSS). All 192 instances in three OSS benchmark sets are examined, and our program found and proved the optimal results for all instances including three previously undecided problems.

1 Introduction

Recent advances in SAT solver technologies [1,2,3,4,5] have enabled solving a problem by encoding it to a SAT problem, and then to use the efficient SAT solver to find a solution, such as for model checking, planning, and scheduling [6,7,8,9,10,11,12].

In this paper, we propose a method to encode Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP) with integer linear constraints into Boolean Satisfiability Testing Problems (SAT) of CNF (product-of-sums) formulas.

As Hoos discussed in [8], basically two encoding methods are known: "sparse encoding" and "compact encoding". Sparse encoding [13] encodes each assignment of a value to an integer variable by a different Boolean variable, that is, Boolean variable representing $x = a$ is used for each integer variable x and integer value a. Compact encoding [14,7] assigns a Boolean variable for each bit of each integer variable.

Encoding method used in this paper is different from these. The method is basically the same with the one used to encode Job-Shop Scheduling Problems by Crawford and Baker in [9] and studied by Soh, Inoue, and Nabeshima in [10,11,12]. It encodes a comparison $x \leq a$ by a different Boolean variable for each integer variable x and integer value a.

The benefit of this encoding is the natural representation of the order relation on integers. Axiom clauses with two literals, such as $\{\neg(x \leq a), x \leq a + 1\}$ for each integer a, represent the order relation for an integer variable x. Clauses, for example $\{x \leq a, \neg(y \leq a)\}$ for each integer a, can be used to represent the constraint among integer variables, i.e. $x \leq y$.

The original encoding method in [9,10,11,12] is only for Job-Shop Scheduling Problems. In this paper, we extend the method so that it can be applied for any finite linear CSPs and COPs.

To evaluate the effectiveness of this approach, we applied the method to Open-Shop Scheduling Problems (OSS). All 192 instances in three OSS benchmark sets [15,16,17] are examined, and our program found and proved the optimal results for all instances including three previously undecided problems [18,19,20].

2 Finite Linear CSP and SAT

In this section, we define finite linear *Constraint Satisfaction Problems* (CSP) and *Boolean Satisfiability Testing Problems* (SAT) of CNF formulas.

$\mathbf{Z}$ is used to denote a set of integers and $\mathbf{B}$ is used to denote a set of Boolean constants ($\top$ and $\bot$ are the only elements of $\mathbf{B}$ representing "true" and "false" respectively).

We also prepare two countably infinite sets of *integer variables* $\mathcal{V}$ and *Boolean variables* $\mathcal{B}$. Although only a finite number of variables are used in a specific CSP or SAT, countably infinite variables are prepared to introduce new variables during the translation. Symbols x, y, z, x_1, y_1, z_1, $\dots$, are used to denote integer variables, and symbols p, q, r, p_1, q_1, r_1, $\dots$, are used to denote Boolean variables.

Linear expressions over $V \subset \mathcal{V}$, denoted by $E(V)$, are algebraic expressions in the form of $\sum a_i\, x_i$ where a_i's are non-zero integers and x_i's are integer variables (elements of V). We also add the restriction that x_i's are mutually distinct.

Literals over $V \subset \mathcal{V}$ and $B \subset \mathcal{B}$, denoted by $L(V, B)$, consist of Boolean variables $\{p \mid p \in B\}$, negations of Boolean variables $\{\neg p \mid p \in B\}$, and *comparisons* $\{e \leq c \mid e \in E(V), c \in \mathbf{Z}\}$. Please note that we restrict comparison literals to only appear positively and in the form of $\sum a_i\, x_i \leq c$ without loss of generality. For example, $\neg(a_1x_1 + a_2x_2 \leq c)$ can be represented with $-a_1x_1 - a_2x_2 \leq -c - 1$, and $x \neq y$ (that is, $(x < y) \vee (x > y)$) can be represented with $(x - y \leq -1) \vee (-x + y \leq -1)$.

Clauses over $V \subset \mathcal{V}$ and $B \subset \mathcal{B}$, denoted by $C(V, B)$, are defined as usual where literals are chosen from $L(V, B)$, that is, a clause represents a disjunction of element literals. Integer variables occurring in a clause are treated as free variables, that is, a clause $\{x \leq 0\}$ does not mean $\forall x.(x \leq 0)$.

Definition 1 (Finite linear CSP). A (finite linear) CSP (Constraint Satisfaction Problem) is defined as a tuple (V, ℓ, u, B, S) where

(1) V is a finite subset of *integer variables* $\mathcal{V}$,
(2) ℓ is a mapping from V to $\mathbf{Z}$ representing the *lower bound* of the integer variable,
(3) u is a mapping from V to $\mathbf{Z}$ representing the *upper bound* of the integer variable,
(4) B is a finite subset of *Boolean variables* $\mathcal{B}$, and

(5) S is a finite set of clauses (that is, a finite subset of $C(V, B)$) representing the constraint to be satisfied.

In the rest of this paper, we simply call finite linear CSP as CSP.

We extend the functions ℓ and u for any linear expressions $e \in E(V)$, e.g. $\ell(2x-3y) = -9$ and $u(2x-3y) = 6$ when $\ell(x) = \ell(y) = 0$ and $u(x) = u(y) = 3$.

An *assignment* of a CSP (V, ℓ, u, B, S) is a pair (α, β) where α is a mapping from V to $\mathbf{Z}$ and β is a mapping from B to $\{\top, \bot\}$.

Definition 2 (Satisfiability). Let (V,ℓ,u, B, S) be a CSP. A clause $C \in C(V, B)$ is *satisfiable* by an assignment (α, β) if the assignment makes the clause C be true and $\ell(x) \le \alpha(x) \le u(x)$ for all $x \in V$. We denote this satisfiability relation as follows.

$$(\alpha, \beta) \models C$$

A clause C is satisfiable if C is satisfiable by some assignment.

A set of clauses is satisfiable when all clauses in the set are satisfiable by the same assignment. A logical formula is satisfiable when its clausal form is satisfiable. The CSP is satisfiable if S is satisfiable.

Finally, we define SAT as a special form of CSP.

Definition 3 (SAT). A SAT (Boolean Satisfiability Testing Problem) is a CSP without integer variables, that is, $(\emptyset, \emptyset, \emptyset, B, S)$.

3 Encoding Finite Linear CSP to SAT

3.1 Converting Comparisons to Primitive Comparisons

In this section, we will explain a method to transform a comparison into primitive comparisons.

A *primitive comparison* is a comparison in the form of $x \le c$ where x is an integer variable and c is an integer satisfying $\ell(x) - 1 \le c \le u(x)$. In fact, it is possible to restrict the range of c to $\ell(x) \le c \le u(x) - 1$ since $x \le \ell(x) - 1$ is always false and $x \le u(x)$ is always true. However, we use the wider range to simplify the discussion.

Let us consider a comparison of $x + y \le 7$ when $\ell(x) = \ell(y) = 0$ and $u(x) = u(y) = 6$. As shown in Figure 1, the comparison can be equivalently expressed as $(x \le 1 \vee y \le 5) \wedge (x \le 2 \vee y \le 4) \wedge (x \le 3 \vee y \le 3) \wedge (x \le 4 \vee y \le 2) \wedge (x \le 5 \vee y \le 1)$ in which 10 black dotted points are contained as satisfiable assignments since $0 \le x, y \le 6$. Please note that conditions $(x \le 1 \vee y \le 5)$ and $(x \le 5 \vee y \le 1)$, which are equivalent to $y \le 5$ and $x \le 5$ respectively, are necessary to exclude cases of $x = 2$, $y = 6$ and $x = 6$, $y = 2$.

Now, we will show the following lemma before describing the conversion to primitive comparisons in general.

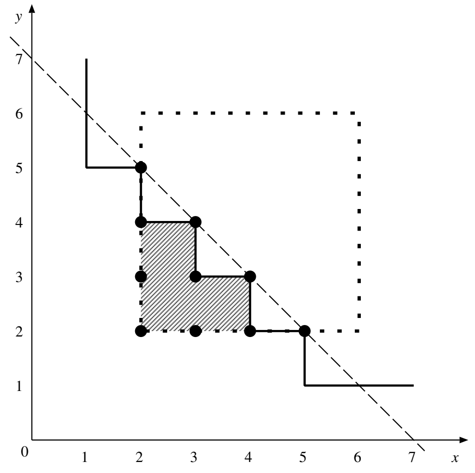


Fig. 1. Converting $x + y \le 7$ to primitive comparisons

Lemma 1. Let (V, ℓ, u, B, S) be a CSP, then for any assignment (α, β) of the CSP, for any linear expressions $e, f \in E(V)$, and for any integer $c \ge \ell(e) + \ell(f)$, the following holds.

$$\begin{aligned} & (\alpha, \beta) \models e + f \le c \\ \Longleftrightarrow \quad & (\alpha, \beta) \models \bigwedge_{a+b=c-1} (e \le a \ \vee \ f \le b) \end{aligned}$$

Parameters a and b range over $\mathbf{Z}$ satisfying $a + b = c - 1$, $\ell(e) - 1 \le a \le u(e)$, and $\ell(f) - 1 \le b \le u(f)$. The conjunction represents $\top$ if there are no such a and b.

Proof. ($\Longrightarrow$) From the hypotheses and the definition of satisfiability, we get $\alpha(e) + \alpha(f) \le c$, $\ell(e) \le \alpha(e) \le u(e)$, and $\ell(f) \le \alpha(f) \le u(f)$. Let a and b be any integers satisfying $a + b = c - 1$, $\ell(e) - 1 \le a \le u(e)$, and $\ell(f) - 1 \le b \le u(f)$. If there are no such a and b, the conclusion holds.

If $\alpha(e) \le a$, $e \le a$ in the conclusion is satisfied. Otherwise, $f \le b$ in the conclusion is satisfied since $\alpha(f) \le c - \alpha(e) \le c - a - 1 = (a + b + 1) - a - 1 = b$. Therefore, $e \le a \vee f \le b$ is satisfied for any a and b.

($\Longleftarrow$) From the hypotheses, $\alpha(e) \le a \vee \alpha(f) \le b$ is true for any a and b satisfying $a + b = c - 1$, $\ell(e) - 1 \le a \le u(e)$, and $\ell(f) - 1 \le b \le u(f)$. From the definition of satisfiability, we also have $\ell(e) \le \alpha(e) \le u(e)$ and $\ell(f) \le \alpha(f) \le u(f)$. Now, we show the conclusion through a proof by contradiction. Assume that $\alpha(e) + \alpha(f) > c$ which is the negation of the conclusion.

When $\alpha(e) \ge c - \ell(f) + 1$, we choose $a = c - \ell(f)$ and $b = \ell(f) - 1$. It is easy to check the conditions $\ell(e) - 1 \le a \le u(e)$ and $\ell(f) - 1 \le b \le u(f)$ are satisfied, and $\alpha(e) \le a \vee \alpha(f) \le b$ becomes false for such a and b, which contradicts the hypotheses.

When $\alpha(e) < c - \ell(f) + 1$, we choose $a = \alpha(e) - 1$ and $b = c - \alpha(e)$. It is easy to check the conditions $\ell(e) - 1 \leq a \leq u(e)$ and $\ell(f) - 1 \leq b \leq u(f)$ are satisfied, and $\alpha(e) \leq a \vee \alpha(f) \leq b$ becomes false for such a and b, which contradicts the hypotheses. □

The following proposition shows a general method to convert a (linear) comparison into primitive comparisons.

Proposition 1. Let (V, ℓ, u, B, S) be a CSP, then for any assignment (α, β) of the CSP, for any linear expression $\sum_{i=1}^{n} a_i\, x_i \in E(V)$, and for any integer $c \geq \ell(\sum_{i=1}^{n} a_i\, x_i)$ the following holds.

$$\begin{aligned} & (\alpha, \beta) \models \sum_{i=1}^{n} a_i\, x_i \leq c \\ \Longleftrightarrow \quad & (\alpha, \beta) \models \bigwedge_{\sum_{i=1}^{n} b_i = c - n + 1} \bigvee_{i} (a_i\, x_i \leq b_i)^{\#} \end{aligned}$$

Parameters b_i's range over $\mathbf{Z}$ satisfying $\sum_{i=1}^{n} b_i = c - n + 1$ and $\ell(a_i x_i) - 1 \leq b_i \leq u(a_i x_i)$ for all i. The translation $()^{\#}$ is defined as follows.

$$(a\, x \leq b)^{\#} \;\equiv\; \begin{cases} x \leq \left\lfloor \dfrac{b}{a} \right\rfloor & (a > 0) \\ \neg \left(x \leq \left\lceil \dfrac{b}{a} \right\rceil - 1 \right) & (a < 0) \end{cases}$$

Proof. The satisfiability of $\sum a_i\, x_i \leq c$ is equivalent to the satisfiability of $\bigwedge \bigvee (a_i\, x_i \leq b_i)$ from Lemma 1, and the satisfiability of each $a_i\, x_i \leq b_i$ is equivalent to the satisfiability of $(a_i\, x_i \leq b_i)^{\#}$. □

Therefore, any comparison literal $\sum a_i\, x_i \leq c$ in a CSP can be converted to a CNF (product-of-sums) formula of primitive comparisons (or Boolean constants) without changing its satisfiability. Please note that the comparison literal should occur positively in the CSP to perform this conversion.

Example 1. When $\ell(x) = \ell(y) = \ell(z) = 0$ and $u(x) = u(y) = u(z) = 3$, comparison $x + y < z - 1$ is converted into $(x \leq -1 \vee y \leq -1 \vee \neg(z \leq 1)) \wedge (x \leq -1 \vee y \leq 0 \vee \neg(z \leq 2)) \wedge (x \leq -1 \vee y \leq 1 \vee \neg(z \leq 3)) \wedge (x \leq 0 \vee y \leq -1 \vee \neg(z \leq 2)) \wedge (x \leq 0 \vee y \leq 0 \vee \neg(z \leq 3)) \wedge (x \leq 1 \vee y \leq -1 \vee \neg(z \leq 3))$.

3.2 Encoding to SAT

As shown in the previous subsection, any (finite linear) CSP can be converted into a CSP with only primitive comparisons.

Now, we eliminate each primitive comparison $x \leq c$ ($x \in V$, $\ell(x) - 1 \leq c \leq u(x)$) by replacing it with a newly introduced Boolean variable $p(x, c)$ which is chosen from $\mathcal{B}$. We denote a set of these new Boolean variables as follows.

$$B' = \{p(x, c) \mid x \in V,\ \ell(x) - 1 \leq c \leq u(x)\}$$

We also need to introduce the following axiom clauses $A(x)$ for each integer variable x in order to represent the bound and the order relation.

$$\begin{aligned} A(x) = \{ & \{\neg p(x, \ell(x)-1)\}, \{p(x, u(x))\}\} \\ & \cup \{\{\neg p(x, c-1), p(x, c)\} \mid \ell(x) \le c \le u(x)\} \end{aligned}$$

As previously described, clauses of $\{\neg p(x, \ell(x) - 1)\}$ and $\{p(x, u(x))\}$ are redundant. However, these will be removed in the early stage of SAT solving and will not much affect the performance of the solver.

Proposition 2. Let (V, ℓ, u, B, S) be a CSP with only primitive comparisons, let S^* be a clausal form formula obtained from S by replacing each primitive comparison $x \le c$ with $p(x, c)$, and let $A = \bigcup_{x \in V} A(x)$. Then, the following holds.

$$\begin{aligned} & (V, \ell, u, B, S) \text{ is satisfiable} \\ \Longleftrightarrow \quad & (\emptyset, \emptyset, \emptyset, B \cup B', S^* \cup A) \text{ is satisfiable} \end{aligned}$$

Proof. ($\Longrightarrow$) Since (V, ℓ, u, B, S) is satisfiable, there is an assignment (α, β) which makes S be true and $\ell(x) \le \alpha(x) \le u(x)$ for all $x \in V$. We extend the mapping β to β^* as follows.

$$\beta^*(p) = \begin{cases} \beta(p) & (p \in B) \\ \alpha(x) \le c & (p = p(x, c) \in B') \end{cases}$$

Then an assignment (α, β^*) satisfies $S^* \cup A$.

($\Longleftarrow$) From the hypotheses, there is an assignment $(\emptyset, \beta)$ which makes $S^* \cup A$ be true. We define a mapping α as follows.

$$\alpha(x) = \min \{c \mid \ell(x) \le c \le u(x),\ p(x, c)\}$$

It is straightforward to check the assignment (α, β) satisfies S. □

3.3 Keeping Clausal Form

When encoding a clause of CSP to SAT, the encoded formula is no more a clausal form in general.

Consider a case of encoding a clause $\{x - y \le -1, -x + y \le -1\}$ which means $x \ne y$. Each of $x - y \le -1$ and $-x + y \le -1$ is encoded into a CNF formula of primitive comparisons. Therefore, when we expand the conjunctions to get a clausal form, the number of obtained clauses is the multiplication of two numbers of primitive comparisons.

As it is well known, introduction of new Boolean variables is useful to reduce the size. Suppose $\{c_1, c_2, \dots, c_n\}$ is a clause of original CSP where c_i's are comparison literals, and $\{C_{i1}, C_{i2}, \dots, C_{in_i}\}$ is an encoded CNF formula (in clausal form) of c_i for each i.

We introduce new Boolean variables p_1, p_2, …, p_n chosen from $\mathcal{B}$, and replace the original clause with $\{p_1, p_2, \ldots, p_n\}$. We also introduce new clauses $\{\neg p_i\} \cup C_{ij}$ for each $1 \le i \le n$ and $1 \le j \le n_i$.

This conversion does not affect the satisfiability which can be shown from the following Lemma.

Lemma 2. Let (V, ℓ, u, B, S) be a CSP, $\{L_1, L_2, \ldots, L_n\}$ be a clause of the CSP, and p_1, p_2, …, p_n be new Boolean variables. Then, the following holds.

$$\begin{aligned} & \{L_1, L_2, \ldots, L_n\} \text{ is satisfiable} \\ \Longleftrightarrow\quad & \{\{p_1, p_2, \ldots, p_n\}\, \{\neg p_1, L_1\}, \{\neg p_2, L_2\}, \ldots, \{\neg p_n, L_n\}\} \text{ is satisfiable} \end{aligned}$$

Proof. ($\Longrightarrow$) From the hypotheses, there is an assignment (α, β) which satisfies some L_i. We extend the mapping β so that $\beta(p_i) = \top$ and $\beta(p_j) = \bot$ $(j \neq i)$. Then, the assignment satisfies converted clauses.

($\Longleftarrow$) From the hypotheses, there is an assignment (α, β) which satisfies some p_i. The assignment also satisfies $\{\neg p_i, L_i\}$, and therefore L_i. Hence the conclusion holds. □

Example 2. Consider an example of encoding a clause $\{x - y \le -1, -x + y \le -1\}$ when $\ell(x) = \ell(y) = 0$ and $u(x) = u(y) = 2$. $x - y \le -1$ and $-x + y \le -1$ are converted into $S_1 = (p(x,-1) \vee \neg p(y,0)) \wedge (p(x,0) \vee \neg p(y,1)) \wedge (p(x,1) \vee \neg p(y,2))$ and $S_2 = (\neg p(x,2) \vee p(y,1)) \wedge (\neg p(x,1) \vee p(y,0)) \wedge (\neg p(x,0) \vee p(y,-1))$ respectively. Expanding $S_1 \vee S_2$ generates 9 clauses. However, by introducing new Boolean variables p and q, we obtain the following seven clauses.

$$\begin{array}{ccc} & \{p, q\} & \\ \{\neg p, p(x,-1), \neg p(y,0)\} & \{\neg p, p(x,0), \neg p(y,1)\} & \{\neg p, p(x,1), \neg p(y,2)\} \\ \{\neg q, \neg p(x,2), p(y,1)\} & \{\neg q, \neg p(x,1), p(y,0)\} & \{\neg q, \neg p(x,0), p(y,-1)\} \end{array}$$

3.4 Size of the Encoded SAT Problem

Usually the size of the encoded SAT problem becomes large.

Suppose the number of integer variables is n, and the size of integer variable domains is d, that is, $d = u(x) - \ell(x) + 1$ for all $x \in V$. Then the size of newly introduced Boolean variables B' is $O(n\,d)$, the size of axiom clauses A is also $O(n\,d)$, and the number of literals in each axiom clause is at most two.

Each comparison $\sum_{i=1}^{m} a_i\,x_i \le c$ will be encoded into $O(d^{m-1})$ clauses in general by Proposition 1.

However, it is possible to reduce the number of integer variables in each comparison at most three. For example, $x_1 + x_2 + x_3 + x_4 \le c$ can be replaced with $x + x_3 + x_4 \le c$ by introducing a new integer variable x and new constraints $x - x_1 - x_2 \le 0$ and $-x + x_1 + x_2 \le 0$, that is, $x = x_1 + x_2$.

Therefore, each comparison $\sum_{i=1}^{m} a_i\,x_i \le c$ can be encoded by at most $O(d^2) + O(md)$ clauses even when $m \ge 4$, and the number of literals in each clause is at most four (three for integer variables and one for the case handling described in the previous subsection).

$$(p_{ij}) = \begin{pmatrix} 661 & 6 & 333 \\ 168 & 489 & 343 \\ 171 & 505 & 324 \end{pmatrix}$$

Fig. 2. OSS benchmark instance gp03-01

4 Encoding Finite Linear COP to SAT

Definition 4 (Finite linear COP). A (finite linear) COP (Constraint Optimization Problem) is defined as a tuple (V, ℓ, u, B, S, v) where

(1) (V, ℓ, u, B, S) is a finite linear CSP, and
(2) $v \in V$ is an integer variable representing the objective variable to be minimized (without loss of generality we assume COPs as minimization problems).

The optimal value of COP (V, ℓ, u, B, S, v) can be obtained by repeatedly solving CSPs.

$$\min \{c \mid \ell(v) \leq c \leq u(v),\ \text{CSP}\ (V, \ell, u, B, S \cup \{\{v \leq c\}\}) \text{ is satisfiable}\}$$

Of course, instead of linear search, binary search method is useful to find the optimal value efficiently as used in previous works [10,11,12].

It is also possible to encode COP to SAT once at first, and repeatedly modify only the clause $\{v \leq c\}$ for a given c. This procedure substantially reduces the time spent for encoding.

5 Solving OSS

In order to show the applicability of our method, we applied it to OSS (Open-Shop Scheduling) problems. There are three well-known sets of OSS benchmark problems by Guéret and Prins [15] (80 instances denoted by gp*), Taillard [16] (60 instances denoted by tai_*), and Brucker et al. [17] (52 instances denoted by j*), which are also used in [18,19,20].

Some problems in these benchmark sets are very hard to solve. Actually, three instances (j7-per0-0, j8-per0-1, and j8-per10-2) are still open, and 37 instances are closed recently in 2005 by complete MCS-based search solver of ILOG [20].

Representing OSS problem as CSP is straightforward. Figure 2 defines a benchmark instance gp03-01 of 3 jobs and 3 machines. Each element p_{ij} represents the process time of the operation O_{ij} $(0 \leq i, j \leq 2)$. The instance gp03-01 can be represented as a CSP of 27 clauses as shown in Figure 3.

In the figure, integer variables m represents the makespan and each s_{ij} represents the start time of each operation O_{ij}. Clauses $\{s_{ij} + p_{ij} \leq m\}$ represent deadline constraint such that operations should be completed before m. Clauses

$$\begin{array}{ccc}
\{s_{00}+661 \le m\} & \{s_{01}+6 \le m\} & \{s_{02}+333 \le m\} \\
\{s_{10}+168 \le m\} & \{s_{11}+489 \le m\} & \{s_{12}+343 \le m\} \\
\{s_{20}+171 \le m\} & \{s_{21}+505 \le m\} & \{s_{22}+324 \le m\}
\end{array}$$

$$\begin{array}{rr}
\{s_{00}+661 \le s_{01},\ s_{01}+6 \le s_{00}\} & \{s_{00}+661 \le s_{02},\ s_{02}+333 \le s_{00}\} \\
\{s_{01}+6 \le s_{02},\ s_{02}+333 \le s_{01}\} & \{s_{10}+168 \le s_{11},\ s_{11}+489 \le s_{10}\} \\
\{s_{10}+168 \le s_{12},\ s_{12}+343 \le s_{10}\} & \{s_{11}+489 \le s_{12},\ s_{12}+343 \le s_{11}\} \\
\{s_{20}+171 \le s_{21},\ s_{21}+505 \le s_{20}\} & \{s_{20}+171 \le s_{22},\ s_{22}+324 \le s_{20}\} \\
\{s_{21}+505 \le s_{22},\ s_{22}+324 \le s_{21}\} & \{s_{00}+661 \le s_{10},\ s_{10}+168 \le s_{00}\} \\
\{s_{00}+661 \le s_{20},\ s_{20}+171 \le s_{00}\} & \{s_{10}+168 \le s_{20},\ s_{20}+171 \le s_{10}\} \\
\{s_{01}+6 \le s_{11},\ s_{11}+489 \le s_{01}\} & \{s_{01}+6 \le s_{21},\ s_{21}+505 \le s_{01}\} \\
\{s_{11}+489 \le s_{21},\ s_{21}+505 \le s_{11}\} & \{s_{02}+333 \le s_{12},\ s_{12}+343 \le s_{02}\} \\
\{s_{02}+333 \le s_{22},\ s_{22}+324 \le s_{02}\} & \{s_{12}+343 \le s_{22},\ s_{22}+324 \le s_{12}\}
\end{array}$$

Fig. 3. CSP representation of `gp03-01`

$\{s_{ij}+p_{ij} \le s_{kl},\ s_{kl}+p_{kl} \le s_{ij}\}$ represent resource capacity constraint such that the operation O_{ij} and O_{kl} should not be overlapped each other.

Before encoding the CSP to SAT, we also need to determine the lower and upper bound of integer variables. We used the following values ℓ and u (where n is the number of jobs and machines).

$$\ell = \max\left(\max_{0 \le i < n} \sum_{0 \le j < n} p_{ij},\ \max_{0 \le j < n} \sum_{0 \le i < n} p_{ij}\right)$$

$$u = \sum_{0 \le k < n} \max_{(i-j) \bmod n = k} p_{ij}$$

The value u is used for the upper bound of s_{ij}'s and m, and the value ℓ is used for the lower bound of m (the lower bound 0 is used for s_{ij}'s). For example, $\ell = 1000$ and $u = 1509$ for the instance `gp03-01`.

We developed a program called `CSP2SAT` which encodes a CSP representation (of a given OSS problem) into SAT and repeatedly invokes a complete SAT solver to find the optimal solution by binary search[1]. We used MiniSat [5] as the backend complete SAT solver because it is known to be very efficient (MiniSat is a winner of all industrial categories of the SAT 2005 competition).

We run `CSP2SAT` for all 192 instances of the three benchmark sets on Intel Xeon 2.8GHz 4GB memory machine with the time limit of 3 hours (10800 seconds).

Figures 7, 8, and 9 provides the results. The column named "Optim." describes the optimal value found by the program, and "CPU" describes the total CPU time in seconds including encoding process. The column named "SAT" describes the numbers of Boolean variables and clauses in the encoded SAT problem. Although time spent for encoding is not shown separately in the figures, it ranges from 1 second to 1163 seconds and fits linearly with the number of clauses in the encoded SAT program.

[1] The program will be available at `http://bach.istc.kobe-u.ac.jp/csp2sat/`

$$(s_{ij}) = \begin{pmatrix} 247 & 296 & 110 & 618 & 537 & 31 & 500 & 127 \\ 815 & 50 & 328 & 274 & 311 & 672 & 550 & 6 \\ 1 & 583 & 120 & 339 & 876 & 842 & 675 & 58 \\ 293 & 669 & 5 & 72 & 250 & 502 & 403 & 994 \\ 286 & 517 & 870 & 594 & 612 & 347 & 0 & 297 \\ 404 & 252 & 73 & 28 & 83 & 25 & 300 & 734 \\ 707 & 997 & 560 & 12 & 48 & 87 & 842 & 340 \\ 53 & 6 & 703 & 285 & 342 & 872 & 526 & 547 \end{pmatrix}$$

Fig. 4. Optimal Scheduling of `j8-per10-2` found by `CSP2SAT`

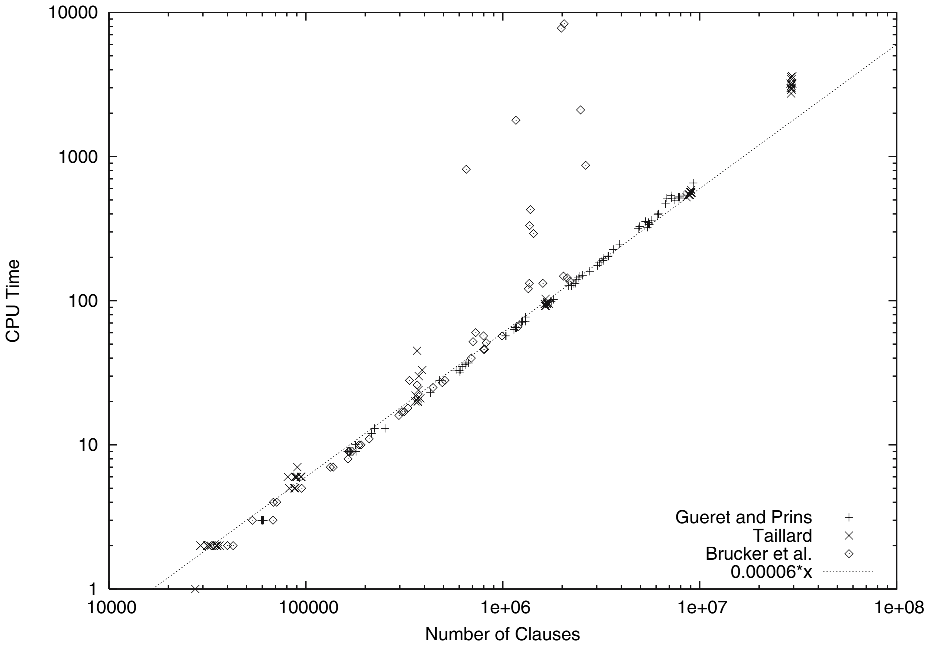


Fig. 5. Log scale plot of the number of clauses and the CPU time

Instance	Makespan	Previously known bounds	
		Lower bound	Upper bound
`j7-per0-0`	1048	1039	1048
`j8-per0-1`	1039	1018	1039
`j8-per10-2`	**1002**	1000	1009

Fig. 6. New results found and proved to be optimal

`CSP2SAT` found the optimal solutions for 189 known problems and one unknown problem (`j8-per10-2`) within 3 hours.

The known upper bound of `j8-per10-2` was 1009. `CSP2SAT` improved the result to 1002 and proved there are no solutions for 1001. Figure 4 shows the start times s_{ij} of the optimal scheduling found by the program.

Instance	Optim.	CPU	SAT	
			Variables	Clauses
gp03-01	1168	3	14155	61133
gp03-02	1170	3	13945	59978
gp03-03	1168	3	13945	59978
gp03-04	1166	3	13995	60253
gp03-05	1170	3	13855	59483
gp03-06	1169	3	13915	59813
gp03-07	1165	3	13925	59868
gp03-08	1167	3	13955	60033
gp03-09	1162	3	14075	60693
gp03-10	1165	3	13945	59978
gp04-01	1281	10	28097	179010
gp04-02	1270	13	33928	223257
gp04-03	1288	9	28182	179655
gp04-04	1261	12	32925	215646
gp04-05	1289	10	27927	177720
gp04-06	1269	9	27383	173592
gp04-07	1267	9	25955	162756
gp04-08	1259	9	26516	167013
gp04-09	1280	9	26737	168690
gp04-10	1263	13	37736	252153
gp05-01	1245	36	72727	643703
gp05-02	1247	33	65993	578694
gp05-03	1265	37	75457	670058
gp05-04	1258	23	50497	429098
gp05-05	1280	33	68151	599527
gp05-06	1269	37	74131	657257
gp05-07	1269	32	68801	605802
gp05-08	1287	28	55489	477290
gp05-09	1262	35	70387	621113
gp05-10	1254	33	69009	607810
gp06-01	1264	57	96410	1038543
gp06-02	1285	65	106659	1158484
gp06-03	1255	72	115317	1259806
gp06-04	1275	63	104957	1138566
gp06-05	1299	65	107806	1171907
gp06-06	1284	65	106400	1155453
gp06-07	1290	77	119091	1303972
gp06-08	1265	71	113726	1241187
gp06-09	1243	72	118943	1302240
gp06-10	1254	57	95559	1028584

Instance	Optim.	CPU	SAT	
			Variables	Clauses
gp07-01	1159	99	137537	1761090
gp07-02	1185	148	188537	2461830
gp07-03	1237	132	179037	2331300
gp07-04	1167	131	176437	2295576
gp07-05	1157	141	182137	2373894
gp07-06	1193	127	166587	2160237
gp07-07	1185	102	141187	1811241
gp07-08	1180	144	184787	2410305
gp07-09	1220	150	194437	2542896
gp07-10	1270	127	171837	2232372
gp08-01	1130	160	186315	2762188
gp08-02	1135	190	216215	3233688
gp08-03	1110	197	215955	3229588
gp08-04	1153	227	242020	3640613
gp08-05	1218	247	259830	3921463
gp08-06	1115	175	203085	3026638
gp08-07	1126	204	229215	3438688
gp08-08	1148	183	207245	3092238
gp08-09	1114	189	213225	3186538
gp08-10	1161	203	227980	3419213
gp09-01	1129	323	317881	5423978
gp09-02	1110	327	291477	4954180
gp09-03	1115	395	357077	6121380
gp09-04	1130	340	322063	5498387
gp09-05	1180	362	333871	5708483
gp09-06	1093	401	359455	6163691
gp09-07	1090	339	325507	5559665
gp09-08	1105	349	321325	5485256
gp09-09	1123	316	286803	4871017
gp09-10	1110	355	310993	5301422
gp10-01	1093	470	353491	6705492
gp10-02	1097	526	412677	7878078
gp10-03	1081	535	376317	7157718
gp10-04	1077	515	378438	7199739
gp10-05	1071	515	358743	6809544
gp10-06	1071	508	410960	7844061
gp10-07	1079	523	408839	7802040
gp10-08	1093	498	392578	7479879
gp10-09	1112	541	434897	8318298
gp10-10	1092	656	483276	9276777

Fig. 7. Results for benchmark instances provided by Guéret and Prins

Figure 5 provides the log scale plot of the number of clauses in the encoded SAT problem (x-axis) and the total CPU time (y-axis) for 190 problems. The mark $+$ is used for `gp*` benchmarks, $\times$ is used for `tai*` benchmarks, and $\diamond$ is used for `j*` benchmarks. Dotted line is a plot of $y = 0.00006x$.

Except some instances of `j*` benchmarks, it seems the total CPU time linearly fits with the number of clauses. This shows that the encoding used in this paper is natural and does not uselessly increase the complexity for SAT solver.

For the remaining two open problems `j7-per0-0` and `j8-per0-1`, we solved and proved their optimal values by using 10 Mac mini machines (PowerPC G4 1.42GHz 1GB memory) running in parallel on Xgrid system [21] and by dividing the problem into 120 subproblems where each subproblem is obtained by specifying the order of six operations. Optimal solutions were found and proved for both of the two remaining instances within 13 hours.

Figure 6 summarizes the newly obtained results. All three remaining open problems in [18,19,20] are now closed.

Instance	Optim.	CPU	SAT	
			Variables	Clauses
tai_4x4_1	193	2	5043	31706
tai_4x4_2	236	1	4643	27426
tai_4x4_3	271	2	5460	32925
tai_4x4_4	250	2	5358	32341
tai_4x4_5	295	2	6081	36418
tai_4x4_6	189	2	4721	29194
tai_4x4_7	201	2	4743	29188
tai_4x4_8	217	2	5629	35110
tai_4x4_9	261	2	5328	31517
tai_4x4_10	217	2	5611	35444
tai_5x5_1	300	6	11526	94098
tai_5x5_2	262	5	10110	82314
tai_5x5_3	323	6	11318	90297
tai_5x5_4	310	5	11047	88190
tai_5x5_5	326	6	10356	80906
tai_5x5_6	312	5	10942	87344
tai_5x5_7	303	6	10951	87906
tai_5x5_8	300	6	11009	88852
tai_5x5_9	353	6	11940	94884
tai_5x5_10	326	7	11344	90508
tai_7x7_1	435	21	30952	370295
tai_7x7_2	443	24	31244	372853
tai_7x7_3	468	30	31669	374258
tai_7x7_4	463	20	31224	370305
tai_7x7_5	416	22	30171	360661
tai_7x7_6	451	45	30986	367026
tai_7x7_7	422	33	32415	389596
tai_7x7_8	424	20	30863	370287
tai_7x7_9	458	21	31929	380761
tai_7x7_10	398	20	29939	359194

Instance	Optim.	CPU	SAT	
			Variables	Clauses
tai_10x10_1	637	98	94183	1678890
tai_10x10_2	588	95	95343	1716326
tai_10x10_3	598	92	92303	1651992
tai_10x10_4	577	92	91314	1639647
tai_10x10_5	640	96	93978	1677177
tai_10x10_6	538	95	91151	1642608
tai_10x10_7	616	103	92285	1648788
tai_10x10_8	595	95	91094	1631685
tai_10x10_9	595	97	94528	1697235
tai_10x10_10	596	95	93315	1674220
tai_15x15_1	937	523	309784	8563684
tai_15x15_2	918	567	325397	9026993
tai_15x15_3	871	543	315726	8767426
tai_15x15_4	934	560	326511	9067128
tai_15x15_5	946	541	323109	8940331
tai_15x15_6	933	560	326512	9067214
tai_15x15_7	891	566	322034	8943618
tai_15x15_8	893	546	319320	8866998
tai_15x15_9	899	568	324060	8998985
tai_15x15_10	902	586	325865	9053491
tai_20x20_1	1155	3105	775142	29178719
tai_20x20_2	1241	3559	777061	29153596
tai_20x20_3	1257	2990	770228	28898989
tai_20x20_4	1248	3442	779059	29238508
tai_20x20_5	1256	3603	785066	29485803
tai_20x20_6	1204	2741	773489	29073596
tai_20x20_7	1294	2912	779414	29225385
tai_20x20_8	1169	2990	778336	29262619
tai_20x20_9	1289	3204	785835	29493666
tai_20x20_10	1241	3208	770645	28917758

Fig. 8. Results for benchmark instances provided by Taillard

Instance	Optim.	CPU	SAT	
			Variables	Clauses
j3-per0-1	1127	2	10805	42708
j3-per0-2	1084	5	20335	95123
j3-per10-0	1131	3	12675	53453
j3-per10-1	1069	3	15335	68062
j3-per10-2	1053	4	15355	68341
j3-per20-0	1026	2	10015	39923
j3-per20-1	1000	2	9245	35496
j3-per20-2	1000	4	15755	71137
j4-per0-0	1055	7	22062	133215
j4-per0-1	1180	11	32160	209841
j4-per0-2	1071	8	26057	163530
j4-per10-0	1041	10	29457	190740
j4-per10-1	1019	7	22538	137589
j4-per10-2	1000	9	26057	164892
j4-per20-0	1000	10	28726	186429
j4-per20-1	1004	9	26074	165849
j4-per20-2	1009	9	26822	171525
j5-per0-0	1042	28	40825	335726
j5-per0-1	1054	28	58687	508163
j5-per0-2	1063	26	44127	367603
j5-per10-0	1004	18	39967	329523
j5-per10-1	1002	17	37653	307928
j5-per10-2	1006	16	36509	296700
j5-per20-0	1000	17	38329	315830
j5-per20-1	1000	27	56607	492707
j5-per20-2	1012	25	51485	442196

Instance	Optim.	CPU	SAT	
			Variables	Clauses
j6-per0-0	1056	817	63443	652740
j6-per0-1	1045	57	92340	990913
j6-per0-2	1063	57	75801	797362
j6-per10-0	1005	52	67661	705462
j6-per10-1	1021	46	76467	808206
j6-per10-2	1012	51	77799	823964
j6-per20-0	1000	60	69400	727773
j6-per20-1	1000	46	75431	798740
j6-per20-2	1000	40	66181	692002
j7-per0-0	–	–	85887	1051419
j7-per0-1	1055	428	109837	1380492
j7-per0-2	1056	292	113537	1431330
j7-per10-0	1013	332	108687	1368170
j7-per10-1	1000	121	107087	1347411
j7-per10-2	1011	1786	93887	1165467
j7-per20-0	1000	66	95487	1193523
j7-per20-1	1005	132	125087	1595847
j7-per20-2	1003	132	107987	1361349
j8-per0-1	–	–	145495	2118473
j8-per0-2	1052	870	177995	2630988
j8-per10-0	1017	2107	168310	2481679
j8-per10-1	1000	8346	140620	2047787
j8-per10-2	1002	7789	136655	1984646
j8-per20-0	1000	148	139255	2030756
j8-per20-1	1000	136	149265	2191364
j8-per20-2	1000	144	145300	2125157

Fig. 9. Results for benchmark instances provided by Brucker et al.

6 Conclusion

In this paper, we proposed a method to encode Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP) with integer linear constraints into Boolean Satisfiability Testing Problems (SAT).

To evaluate the effectiveness of the encoding, we applied the method to Open-Shop Scheduling Problems (OSS). All 192 instances in three OSS benchmark sets are examined, and our program found and proved the optimal results for all instances including three previously undecided problems.

Acknowledgments

We would like to give thanks to Katsumi Inoue, Hidetomo Nabeshima, Takehide Soh, and Shuji Ohnishi for their helpful suggestions.

References

1. Selman, B., Kautz, H.A., Cohen, B.: Local search strategies for satisfiability testing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science **26** (1996) 521–532
2. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 97). (1997) 366–371
3. Marques-Silva, J.P., Sakallah, K.A.: GRAPS: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48** (1999) 506–521
4. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001). (2001) 530–535
5. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003). (2003) 502–518
6. Kautz, H.A., McAllester, D.A., Selman, B.: Encoding plans in propositional logic. In: Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96). (1996) 374–384
7. Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic SAT-compilation of planning problems. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 97). (1997) 1169–1177
8. Hoos, H.H.: SAT-encodings, search space structure, and local search performance. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 99). (1999) 296–303
9. Crawford, J.M., Baker, A.B.: Experimental results on the application of satisfiability algorithms to scheduling problems. In: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94). (1994) 1092–1097
10. Soh, T., Inoue, K., Banbara, M., Tamura, N.: Experimental results for solving job-shop scheduling problems with multiple SAT solvers. In: Proceedings of the 1st International Workshop on Distributed and Speculative Constraint Processing (DSCP'05). (2005)

11. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics (2006) (to appear).
12. Nabeshima, H., Soh, T., Inoue, K., Iwanuma, K.: Lemma reusing for SAT based planning and scheduling. In: Proceedings of the International Conference on Automated Planning and Scheduling 2006 (ICAPS'06). (2006) 103–112
13. de Kleer, J.: A comparison of ATMS and CSP techniques. In: Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI 89). (1989) 290–296
14. Iwama, K., Miyazaki, S.: SAT-variable complexity of hard combinatorial problems. In: Proceedings of the IFIP 13th World Computer Congress. (1994) 253–258
15. Guéret, C., Prins, C.: A new lower bound for the open-shop problem. Annals of Operations Research **92** (1999) 165–183
16. Taillard, E.D.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64** (1993) 278–285
17. Brucker, P., Hurink, J., Jurisch, B., Wöstmann, B.: A branch & bound algorithm for the open-shop problem. Discrete Applied Mathematics **76** (1997) 43–59
18. Jussien, N., Lhomme, O.: Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence **139** (2002) 21–45
19. Blum, C.: Beam-ACO — hybridizing ant colony optimization with beam search: an application to open shop scheduling. Computers & OR **32** (2005) 1565–1591
20. Laborie, P.: Complete MCS-based search: Application to resource constrained project scheduling. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) 181–186
21. Apple Computer Inc.: Xgrid Guide. (2004)

Differentiable Invariants

Pascal Van Hentenryck[1] and Laurent Michel[2]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-2155

Abstract. Invariants that incrementally maintain the value of expressions under assignments to their variables are a natural abstraction to build high-level local search algorithms. But their functionalities are not sufficient to allow arbitrary expressions as constraints or objective functions as in constraint programming. Differentiable invariants bridge this expressiveness gap. A differentiable invariant maintains the value of an expression and its variable gradients, it supports differentiation to evaluate the effect of local moves. The benefits of differentiable invariants are illustrated on a number of applications which feature complex, possibly reified, expressions and whose models are essentially similar to their CP counterparts. Experimental results demonstrate their practicability.

1 Introduction

Local search algorithms approach the solving of combinatorial optimization problems by moving from solutions to solutions until a feasible solution or a high-quality solution is found. These algorithms typically maintain sophisticated data structures to evaluate or to propagate the effect of local moves quickly. Since, in general, the neighborhood does not vary dramatically when moving from one solution to one of its neighbors, these incremental data structures may significantly speed up local search algorithms.

Invariants were introduced in Localizer [3] to automate the tedious and error-prone implementation of incremental data structures. An invariant declaratively specifies a (numerical, set, or graph) expression whose value must be maintained incrementally under local moves. Invariants were shown to be instrumental in simplifying the implementation of many local search algorithms. However, the resulting algorithms were still not expressed at a similar level of abstraction as constraint programming (CP) approaches for the same problems. This recognition led to the concept of differentiable objects [5,8] which have emerged as the cornerstone of constraint-based local search (CBLS). In CBLS, objective functions and constraints, which are differentiable objects, not only maintain the value of an expression: they also maintain variable gradient (e.g., to determine how the expression value increases/decreases by changing a variable) and support differentiation (e.g., to assess the effect of a local move on the expression value). Although differentiable objects are often implemented using invariants (see [8] for some examples), it is still cumbersome, difficult, and repetitive to derive correct invariants for a given differentiable object.

This paper aims at bridging the expressiveness gap between invariants and differentiable objects by providing systematic ways of deriving differential objectives and constraints. *It proposes the concept of differentiable invariant that automatically lifts an arbitrarily complex expression into a differentiable objective function.* Like invariants, the resulting objective function incrementally maintains the value of the expression. Unlike invariants, it also maintains variable gradients (to determine how much a variable may increase/decrease the value of the expression) and supports the differentiability (to determine the effect of local moves on the expression value). Moreover, since a differentiable constraint can be seen as differentiable objective function maintaining its violations, *differentiable invariants automatically lift arbitrarily complex relations into differentiable constraints.* The resulting differentiable constraints maintain the violations of the relations, their variable violations (to determine how much a variable may increase/decrease the violations), and support differentiability.

As a consequence, differentiable invariants bring two main benefits for CBLS. First, expressions can now be used to state complex idiosyncratic constraints and objectives declaratively, a functionality that have accounted for much of the industrial success of constraint programming and that relieves programmers from deriving specific invariants and algorithms for each possible expressions. In other words, in the same way as CP languages perform domain reduction on arbitrary expressions, Comet now allows arbitrary expressions and relations as differentiable invariants, maintaining their values, their violations, and their variable gradients, as well as supporting differentiability. Second, differentiable invariants allow CBLS and CP models to be remarkably close since both now feature a similar, rich language for stating constraints and objectives.

The rest of this paper illustrates the concept of differentiable invariants, describes their implementation, and reports experimental results. Sections 2–4 show how differentiable invariants are a natural vehicle for modeling the spatially balanced latin square, scene allocation, and progressive party problems. Sections 5–7 show how to implement differentiable invariants in stepwise refinements, starting with their evaluations and gradients before presenting constraints and their reification. Section 8 presents the experimental results.

2 Totally Spatially Balanced Latin Squares

The first application consists of generating spatially balanced scientific experiment designs and, in particular, totally spatially balanced Latin squares [2].

The Problem: A latin square of size n is an $n \times n$ matrix in which each number in $1..n$ appears exactly once in each row and column. The distance $d_r(v,w)$ of a pair (v,w) in row r is the absolute difference of the column indices in which v and w appear in row r. The total distance of a pair (v,w) is given by

$$d(v,w) = \sum_{r=1}^{n} d_r(v,w).$$

A Latin square is totally spatially balanced if

$$d(v,w) = \frac{n(n+1)}{3} \quad (1 \leq k < l \leq n).$$

Gomes et al [2] introduced this problem to the community and proposed both local search and constraint programming solutions. Subsequently, Gomes [6] proposed a streamlined local search which solves large instances rapidly by permuting columns. This section only considers the local search in [2] since it raises more interesting modeling issues (for our purposes) and motivated our initial research on differentiable invariants.

The Model: The model uses a variable $col[r,v]$ to denote the column of value v in row. The (latin square) constraint that a value v appears in exactly one column is expressed by an *alldifferent* constraint $\mathit{alldifferent}(col[1,v],...,col[n,v])$. The (latin square) constraint that all the values in a row are different is an (implicit) hard constraint. It holds initially by assigning all rows to a permutation of $1..n$ and is maintained during the search by the local moves. The constraint that the Latin square be totally spatially balanced is soft and is transformed into an objective function. Hence the goal is to find a latin square minimizing

$$O = \sum_{1 \leq v < w \leq n} \big(d(v,w) - \frac{n(n+1)}{3}\big)^2.$$

Since the column constraint is a soft constraint as well, the overall problem can then viewed as minimizing the objection function $n \times viol(S) + O$ where *viol(S)* denotes the violations of the soft constraints and is weighted by n.

The Search: The local search is a tabu procedure swapping the position of two values on the same row. The best non-tabu move is selected at each step. The local search also uses an intensification component and a restart strategy.

The Comet Program: Figure 1 depicts the Comet statement. The declaration of data and decision variables are in lines 1–8. The soft constraints are specified in lines 9–11. The objective for spatial balance is in lines 12–14. The global objective is in line 15. The search procedure is in line 17–27. The Comet program is almost a one-to-one mapping of the informal description presented earlier and we review some of its components now. The matrix of decision variables is declared in line 4. All variables have a domain $1..n$ and each row is initialized by a random permutation (lines 5–8). A constraint system `S` is declared in line 9 and it contains all the soft constraints expressing that a value `v` appears atmost once in each column. The objective function for balancing the latin square spatially is specified in line 12. Variable `OS` (declared in line 12) is a sum of objectives, one for each pair of values `(v,w)` to express their relative balance, i.e.,

```
((sum(r in R) (abs(col[r,v] - col[r,w])))-balance)^2
```

Such an objective is a differentiable invariant involving absolute values, a subtraction, a square function, and an aggregate operator. The value of the expression is maintained incrementally under changes to its variables (i.e., all the decision variables `col[r,w]` and `col[r,w]` associated with values `v` and `w`). Moreover, the objective is differentiable and can be queried to evaluate the effect of local moves. Finally, it also maintains gradient information to estimate how much each variable may increase or decrease its value. Observe also that the expression involves absolutes values and a square function. The soft constraints are transformed into an objective function in line 15, specifying the overall objective `O` that combines the weighted violations of the soft constraints with the sum of the balance objectives.

(Part of) the tabu search is depicted in lines 17–27. As long as `O` evaluates to strictly positive value, the search selects the positions of the values v and w on row r that are not tabu and whose swap produces the best value of the objective function. The call in line 22, i.e., `O.getSwapDelta(col[r,v],col[r,w])`, is particularly interesting, as it queries the soft constraints and the objectives to evaluate the candidate swap. This ability to estimate the effect of loval moves on arbitrary expressions is one of the novel contributions of differentiable invariants.

It is useful to emphasize that COMET enables a direct and natural formulation of the model. The constraints and objective functions are expressed declaratively. Their violations and evaluations are maintained incrementally and can be queried to assess the impact of local moves, providing the clean separation between model and search typically associated with CP and CBLS. The novelty for CBLS is the ability to use complex expressions as objectives.

3 Scene Allocation

The second application is the scene allocation problem [7].

The Problem: A scene allocation consists of deciding when to shoot scenes for a movie. Each scene involves a number of actors and each actor may appear in a number of different scenes. All actors of a scene must be present on the day the scene is shot and at most 5 scenes a day can be filmed. The actors have fees representing the amount to be paid per day they spent in the studio. The goal of the application is to minimize the production costs while satisfying the capacity constraints on the number of scenes per day.

The Model: The local search model is essentially the same as the CP model. It associates a variable $scene[s]$ with every scene s to represent the day s is filmed. The objective function uses reification to decide whether to pay an actor on a given day. The capacity on the scenes is an (implicit) hard constraint. It is satisfied by the initial assignment and maintained through local moves.

The Local Search: The local search is again a tabu procedure whose local moves swaps the days allocated to two scenes. Once again, the best non-tabu swap is selected at each step. A restarting strategy is also used.

```
 int n = 8;
 range R = 1..n;
 int balance = n*(n+1)/3;
 var{int} col[R,R](mgr,R);
 forall(r in R) {
    RandomPermutation p(R);
    forall(v in R) col[r,v] := p.get();
 }
 ConstraintSystem S(mgr);
 forall(v in R)
    S.post(alldifferent(all(r in R) col[r,v]));
 ObjectiveSum OS(mgr);
 forall(v in R, w in R: v < w)
    OS.post(((sum(r in R) (abs(col[r,v] - col[r,w])))-balance)^2);
 Objective O = n * S + OS;
 mgr.close();
 int tabu[R,R,R] = -1;
 int tabuLength = 10;
 int it = 0;
 while (O.evaluation() > 0) {
    selectMin(r in R,v in R, w in R: v < w && tabu[r,v,w] <= it)
             (O.getSwapDelta(col[r,v],col[r,w])) {
      col[r,v] :=: col[r,w];
      tabu[r,v,w] = it + tabuLength;
   }
   it++;
}
```

Fig. 1. A Simple Tabu-Search Algorithm for the Balanced Latin Square Problem

The Comet Program: The COMET program is (partially) depicted in Figure 2. For space reasons, the search procedure is omitted but is essentially the same as in the latin square application. Lines 1–8 declare and initialize the data. In particular, they declare the scenes, the days, the actors (line 4), the actors' fees (line 5), and the actors appearing in the scenes (line 6). Line 8 also specifies the scenes in which an actor appears, which is convenient to state the constraints.

The data and decision variables are declared in lines 9–17. A decision variable `scene[s]` specifies the day scene `s` is scheduled and the scene are allocated randomly initially (see lines 11–12). The most interesting part of the model is the objective function `O` (declared in line 14) which sums the fees of all actors on all days (lines 15–16). Each sub-objective thus represents the fee to be paid by an actor `a` on a day `d`. It is expressed by the differentiable expression

```
pay[a] * (or(s in which[a]) scene[s] == d)
```

which uses reification to determine whether actor `a` plays on day `d`. More precisely, if a scene `s` in `which[a]` is scheduled on day `d`, the disjunction holds and is reified to 1, in which case the amount is `pay[a]`. Otherwise, the disjunction does not hold and is reified to 0.

```
include "localSolver";
int maxScene = 19; range Scenes = 0..maxScene-1;
int maxDay = 5; range Days = 0..maxDay-1;
enum Actor = ...;
int pay[Actor] = ...;
set{Actor} appears[Scenes];
...
set{int} which[a in Actor] = setof(s in Scenes) member(a,appears[s]);
LocalSolver mgr();
var{int} scene[Scenes](mgr,Days);
RandomPermutation perm(Scenes);
forall(i in Scenes) scene[perm.get()] := i/maxDay;

ObjectiveSum O(mgr);
forall(a in Actor, d in Days)
   O.post(pay[a] * (or(s in which[a]) scene[s] == d));
```

Fig. 2. A Simple Tabu-Search Algorithm for the Scene Allocation Problem

What we find remarkable here is that the COMET model is almost identical to the OPL model in [7]: the only difference (besides syntactical details) is the fact that the cardinality constraint on the days is omitted since it holds initially and is maintained by local moves. This similarity is possible because differentiable invariants allows objective function to be complex reified expressions.

4 The Progressive Party Problem

Our last application is the progressive party problem which has been used several times to illustrate constraint-based local search [5,9]. The motivation here is to show that differentiable invariants can also be used to state constraints, providing the equivalent for CBLS of high-order or meta-constraint in CP. The key insight is to recognize that a constraint is nothing else but a differentiable invariant maintaining its violations.

The Comet Program: The COMET program is (partially) depicted in Figure 3. The search procedure can be found in earlier publications (e.g., [8]). The data is described in lines 1–7 and the decision variables are declared and initialized in lines 10–11. A decision variable `boat[g,p]` specifies the boat that group `g` visits in period `p`. The core of the model are the constraints in line 12–18.

The novelty here is in how the model expresses that no two groups meet more than once (in line 18). In [8], this constraint was expressed using a cardinality operator `atmost`. The model above uses a meta-constraint

```
sum(p in Periods) (boat[i,p] == boat[j,p]) <= 1
```

which is the way it would probably be expressed using a traditional constraint-programming tool such as OPL or ILOG SOLVER. The COMET implementation automatically derives the constraint violations of the constraints, i.e.,

```
include "LocalSolver";
int up = 6;
range Hosts = 1..13;
range Guests = 1..29;
range Periods = 1..up;
int cap[Hosts] = ...;
int crew[Guests] = ...;

LocalSolver m();
UniformDistribution distr(Hosts);
var{int} boat[Guests,Periods](m,Hosts) := distr.get();

ConstraintSystem S(m);
forall(g in Guests)
   S.post(2 * alldifferent(all(p in Periods) boat[g,p]));
forall(p in Periods)
   S.post(2 * knapsack(all(g in Guests) boat[g,p],crew,cap));
forall(i in Guests, j in Guests : j > i)
   S.post(sum(p in Periods) (boat[i,p] == boat[j,p]) <= 1);
```

Fig. 3. A Comet Model for the Progressive Party Problem

```
max(0,sum(p in Periods) (boat[i,p] == boat[j,p]) - 1)
```

which is a differentiable invariant involving, once again, reification. Note also that the Comet implementation must automatically derive the variable violations for these constraints, since the search procedure first selects variable with the most violations before choosing the value decreasing the violations the most. This highlights the benefits of differentiable invariants: they let programmers state constraints declaratively while systematically deriving their violations, their variable violations, and differentiation algorithms. How this is achieved is the topic of the next sections.

5 Expressions as Differentiable Objective Functions

As mentioned earlier, a differentiable invariant transforms an expression into a differentiable objective function. The syntax of the expressions used in this paper is given in Figure 4. Differentiable invariants must thus implement, for any such expression, the interface of objective functions depicted in 5. Method `evaluation` specifies the value of the objective function, which is maintained by invariants. Methods `increase` and `decrease` return gradient information for a decision variable `x`, i.e., they estimate by how much the objective may increase or decrease by re-assigning `x`. These gradients are also maintained incrementally. Note that the ability of determining both increasing and decreasing gradients is critical even if one is interested in minimization only. The next three methods specify how the objective value evolve under local moves, i.e., the assignment of a value to a variable, the swap of two variables, and the assignments of values

$v \in \mathcal{N}$; $x, y \in$ *Variable*; $e \in$ *Expression*;
$e ::= v \mid x \mid e + e \mid e - e \mid e \times e \mid \min(e, e) \mid \max(e, e) \mid -e \mid abs(e) \mid e^2 \mid (e) \mid c$

Fig. 4. The Syntax of Expressions (Partial Description)

```
interface Objective {
  var{int} evaluation();
  var{int} increase(var{int} x);
  var{int} decrease(var{int} x);
  int getAssignDelta(var{int} x,int v);
  int getSwapDelta(var{int} x,var{int} y);
  int getAssignDelta(var{int}[] x,var[] v);
  var{int}[] getVariables();
}
```

Fig. 5. The Objective Interface in Comet (Partial Description)

to a set of variables. The rest of this section shows how to implement these functionalities. Aggregate operations (e.g., for summation) can be viewed as shorthands for multiple applications of the same operators and are not discussed here for space reasons.

Evaluations Figure 6 shows how to evaluate an expression and how to maintain it through invariants. In the figure, $\mathbb{E}_\alpha[e]$ denotes the value of expression e under variable assignment α and i_e is the invariant maintaining $\mathbb{E}_\alpha[e]$. Both $\mathbb{E}_\alpha[e]$ and i_e are defined by induction on the structure of expression e. In particular, there is one invariant associated with every sub-expression in e, which is important to implement gradients and differentiations efficiently. In this paper, a variable assignment is a function from variables to integers. Moreover, $\alpha[x/v]$ denotes the assignment behaving like α except that x is now assigned to v. This notation may be generalized to multiple variables. These evaluations do not raise any difficulty and the algorithms to maintain these invariants efficiently are presented in [4]. Note that method `evaluation` in Figure 5 returns i_e for the objective function associated with e.

Gradients. Many search procedures choose local moves by a two-step approach, first selecting the variable to re-assign and then the new value. Typically, the variable selection uses gradients, i.e., information on how much the objective function or the violations may increase/decrease by changing the value of a variable. Since such a variable selection takes place at every iteration of the local search, such gradients are typically maintained incrementally in systems such as Comet. The section shows how to evaluate and maintain gradients for the expressions depicted earlier. The gradients must satisfy the following inequalities:

$$\underset{\alpha}{\overset{x}{\uparrow}} e \geq \max_{v \in D_x} \underset{\alpha[x/v]}{\mathbb{E}}[e] - \underset{\alpha}{\mathbb{E}}[e] \qquad \text{and} \qquad \underset{\alpha}{\overset{x}{\downarrow}} e \geq \underset{\alpha}{\mathbb{E}}[e] - \min_{v \in D_x} \underset{\alpha[x/v]}{\mathbb{E}}[e]$$

where D_x denotes the domain of variable x. Variable gradients thus provide optimistic evaluations to the maximum increase/decrease of expression e by

$$\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{E}}[v] & = v & i_v & \leftarrow v \\
\underset{\alpha}{\mathbb{E}}[x] & = \alpha(x) & i_x & \leftarrow x \\
\underset{\alpha}{\mathbb{E}}[e_1 + e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] + \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1+e_2} & \leftarrow i_{e_1} + i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[e_1 - e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] - \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1-e_2} & \leftarrow i_{e_1} - i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[e_1 \times e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \times \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1\times e_2} & \leftarrow i_{e_1} \times i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[-e] & = -\underset{\alpha}{\mathbb{E}}[e] & i_{-e} & \leftarrow -i_e \\
\underset{\alpha}{\mathbb{E}}[abs(e)] & = abs(\underset{\alpha}{\mathbb{E}}[e]) & i_{abs(e)} & \leftarrow abs(i_e) \\
\underset{\alpha}{\mathbb{E}}[e^2] & = (\underset{\alpha}{\mathbb{E}}[e])^2 & i_{e^2} & \leftarrow (i_e)^2 \\
\underset{\alpha}{\mathbb{E}}[\min(e_1, e_2)] & = \min(\underset{\alpha}{\mathbb{E}}[e_1], \underset{\alpha}{\mathbb{E}}[e_2]) & i_{\min(e_1,e_2)} & \leftarrow \min(i_{e_1}, i_{e_2}) \\
\underset{\alpha}{\mathbb{E}}[\max(e_1, e_2)] & = \max(\underset{\alpha}{\mathbb{E}}[e_1], \underset{\alpha}{\mathbb{E}}[e_2]) & i_{\max(e_1,e_2)} & \leftarrow \max(i_{e_1}, i_{e_2})
\end{array}$$

Fig. 6. The Evaluation of Expressions and their Underlying Invariants

re-assigning variable x only. It is critical to use optimistic evaluations since pessimistic evaluations may artificially reduce the connectivity of the neighborhood. Many of the gradients satisfy these relations at equality. However, for efficiency reasons, it may be beneficial to approximate the right-hand sides for complex nonlinear expressions with multiple occurrences of the same variables. In the following, the assignment α is implicit (unless specified otherwise). Similarly, the gradients are always taken with respect to variable x, and y denotes a variable different from x. The minimum and maximum values in the domain D_x of variable x are denoted by m_x and M_x.

Figure 7 depicts the evaluations of the gradients whose definitions are mutually recursive. It is useful to review some of the rules to convey the intuition. The increasing gradient for subtraction, i.e., $\uparrow[e_1 - e_2] = \uparrow e_1 + \downarrow e_2$, uses the increasing gradient on e_1 and the decreasing gradient on e_2. The rule for absolute value can be written as

$$\uparrow[abs(e)] = \max(abs(\mathbb{E}[e] + \uparrow e), abs(\mathbb{E}[e] - \downarrow e)) - \mathbb{E}[abs(e)].$$

It indicates that there are two ways to increase the absolute value of e: increase or decrease e. The definition captures the increase and subtracts the current value of e. The rule for square and for min/max are similar in spirit, while the multiplication has a more complex case analysis due to the possible signs of the underlying expressions. Observe also the base case for variable x which returns the difference between M_x and $\alpha(x)$. The decreasing gradient for absolute value

$$\downarrow[abs(e)] = \text{if } \mathbb{E}[e] \geq 0 \text{ then } \min(\mathbb{E}[e], \downarrow e) \text{ else } \min(-\mathbb{E}[e], \uparrow e)$$

is interesting. If $\mathbb{E}[e] \geq 0$, the gradient is obtained by decreasing e but the decrease must be bounded by $\mathbb{E}[e]$ since zero is the smallest possible value. Observe that the maximum decrease of $abs(e)$ is not necessarily obtained by the maximum decrease of e and hence the gradient is optimistic. The case $\mathbb{E}[e] < 0$ is symmetric and obtained by increasing e up to $-\mathbb{E}[e]$.

$$
\begin{array}{ll}
\mathbb{E}[e]^{+} & = \mathbb{E}[e] + {\uparrow}e \\
\mathbb{E}[e]^{-} & = \mathbb{E}[e] - {\downarrow}e \\
{\uparrow}[v] & = 0 \\
{\uparrow}[y] & = 0 \\
{\uparrow}[x] & = M_x - \alpha(x) \\
{\uparrow}[e_1 + e_2] & = {\uparrow}e_1 + {\uparrow}e_2 \\
{\uparrow}[e_1 - e_2] & = {\uparrow}e_1 + {\downarrow}e_2 \\
{\uparrow}[-e] & = {\downarrow}e \\
{\uparrow}[abs(e)] & = \max(abs(\mathbb{E}[e]^{+}), abs(\mathbb{E}[e]^{-})) - \mathbb{E}[abs(e)] \\
{\uparrow}[e^2] & = \max((\mathbb{E}[e] + {\uparrow}e)^2, (\mathbb{E}[e] - {\downarrow}e)^2) - \mathbb{E}[e^2] \\
{\uparrow}[\max(e_1, e_2)] & = \max(\mathbb{E}[e_1]^{+}, \mathbb{E}[e_2]^{+}) - \mathbb{E}[\max(e_1, e_2)] \\
{\uparrow}[\min(e_1, e_2)] & = \min(\mathbb{E}[e_1]^{+}, \mathbb{E}[e_2]^{+}) - \mathbb{E}[\min(e_1, e_2)] \\
{\uparrow}[e_1 * e_2] & = \max(\mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{-}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{-}) \\
 & \quad -\mathbb{E}[e_1 * e_2]
\end{array}
$$

$$
\begin{array}{ll}
{\downarrow}[v] & = 0 \\
{\downarrow}[y] & = 0 \\
{\downarrow}[x] & = \alpha(x) - m_x \\
{\downarrow}[e_1 + e_2] & = {\downarrow}e_1 + {\downarrow}e_2 \\
{\downarrow}[e_1 - e_2] & = {\downarrow}e_1 + {\uparrow}e_2 \\
{\downarrow}[-e] & = {\uparrow}e \\
{\downarrow}[abs(e)] & = \text{if } \mathbb{E}[e] \geq 0 \text{ then } \min(\mathbb{E}[e], {\downarrow}e) \text{ else } \min(-\mathbb{E}[e], {\uparrow}e) \\
{\downarrow}[e^2] & = \mathbb{E}[e]^2 - \text{if } \mathbb{E}[e] \geq 0 \text{ then } (\mathbb{E}[e] - \min(\mathbb{E}[e], {\downarrow}e))^2 \text{ else } (\mathbb{E}[e] + \min(-\mathbb{E}[e], {\uparrow}e))^2 \\
{\downarrow}[\max(e_1, e_2)] & = \mathbb{E}[\max(e_1, e_2)] - \max(\mathbb{E}[e_1]^{-}, \mathbb{E}[e_2]^{-}) \\
{\downarrow}[\min(e_1, e_2)] & = \mathbb{E}[\min(e_1, e_2)] - \min(\mathbb{E}[e_1]^{-}, \mathbb{E}[e_2]^{-}) \\
{\downarrow}[e_1 * e_2] & = \mathbb{E}[e_1 * e_2] - \\
 & \quad \min(\mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{-}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{-})
\end{array}
$$

Fig. 7. The Evaluation for the Variable Gradients

Figure 8 depicts the invariants maintaining the gradients. There is an invariant $i_e^{\uparrow}$ and an invariant $i_e^{\downarrow}$) associated with each expression e and each variable x (which is implicit in the figure). These gradient invariants use both gradient invariants on the sub-expressions and evaluation invariants. For instance, the (increasing) gradient invariant for subtraction, i.e.,

$$i^{\uparrow}_{e_1 - e_2} \leftarrow i^{\uparrow}_{e_1} - i^{\downarrow}_{e_2}$$

uses increasing and decreasing gradient invariants on the sub-expressions. The gradient invariant for absolute value can be written as

$$i^{\uparrow}_{abs(e)} \leftarrow \max(abs(i_e + i^{\uparrow}_e), abs(i_e - i^{\downarrow}_e) - i_{abs(e)}$$

and illustrates the use of invariants i_e and $i_{abs(e)}$ for accessing the current value of e and $abs(e)$. Note that methods `increase` and `decrease` in Figure 5 returns $i_e^{\uparrow}$ and $i_e^{\downarrow}$ for the objective function associated with e.

Differentiation Differentiable methods can be evaluated directly. Indeed, given an expression e, a variable x, and a value v, method `e.getAssignDelta(x,v)` returns $\mathbb{E}_{\alpha[x/v]}[e] - \mathbb{E}_{\alpha}[e]$ where α is the current assignment. It is too costly to

$$\begin{array}{ll}
i_e^+ & \leftarrow i_e + i_e^\uparrow \\
i_e^- & \leftarrow i_e - i_e^\downarrow \\
\\
i_v^\uparrow & \leftarrow 0 \\
i_y^\uparrow & \leftarrow 0 \\
i_x^\uparrow & \leftarrow M_x - x \\
i_{e_1+e_2}^\uparrow & \leftarrow i_{e_1}^\uparrow + i_{e_2}^\uparrow \\
i_{e_1-e_2}^\uparrow & \leftarrow i_{e_1}^\uparrow + i_{e_2}^\downarrow \\
i_{-e}^\uparrow & \leftarrow i_e^\downarrow \\
i_{abs(e)}^\uparrow & \leftarrow \max(abs(i_e^+), abs(i_e^-)) - i_{abs(e)} \\
i_{e^2}^\uparrow & \leftarrow \max((i_e^+)^2, (i_e^-)^2) - i_{e^2} \\
i_{\max(e_1,e_2)}^\uparrow & \leftarrow \max(i_{e_1}^+, i_{e_2}^+) - i_{\max(e_1,e_2)} \\
i_{\min(e_1,e_2)}^\uparrow & \leftarrow \min(i_{e_1}^+, i_{e_2}^+) - i_{\min(e_1,e_2)} \\
i_{e_1*e_2}^\uparrow & \leftarrow \max(i_{e_1}^+ * i_{e_2}^+, i_{e_1}^+ * i_{e_2}^-, i_{e_1}^- * i_{e_2}^+, i_{e_1}^- * i_{e_2}^-) - i_{e_1*e_2} \\
\\
i_v^\downarrow & \leftarrow 0 \\
i_y^\downarrow & \leftarrow 0 \\
i_x^\downarrow & \leftarrow x - m_x \\
i_{e_1+e_2}^\downarrow & \leftarrow i_{e_1}^\downarrow + i_{e_2}^\downarrow \\
i_{e_1-e_2}^\downarrow & \leftarrow i_{e_1}^\downarrow + i_{e_2}^\uparrow \\
i_{-e}^\downarrow & \leftarrow i_e^\uparrow \\
i_{abs(e)}^\downarrow & \leftarrow \text{if } i_e \geq 0 \text{ then } \min(i_e, i_e^\downarrow) \text{ else } \min(-i_e, i_e^\uparrow) \\
i_{e^2}^\downarrow & \leftarrow i_{e^2} - \text{if } i_e \geq 0 \text{ then } (i_e - \min(i_e, i_e^\downarrow))^2 \text{ else } (i_e + \min(-i_e, i_e^\uparrow))^2 \\
i_{\max(e_1,e_2)}^\downarrow & \leftarrow i_{\max(e_1,e_2)} - \max(i_{e_1}^-, i_{e_2}^-) \\
i_{\min(e_1,e_2)}^\downarrow & \leftarrow i_{\min(e_1,e_2)} - \min(i_{e_1}^-, i_{e_2}^-) \\
i_{e_1*e_2}^\downarrow & \leftarrow i_{e_1*e_2} - \min(i_{e_1}^+ * i_{e_2}^+, i_{e_1}^+ * i_{e_2}^-, i_{e_1}^- * i_{e_2}^+, i_{e_1}^- * i_{e_2}^-)
\end{array}$$

Fig. 8. The Invariants of the Variable Gradients

maintain these evaluations incrementally for each pair (x, v) in general. However, differentiable methods may exploit the fact that variables typically occur only in some sub-expressions and reuse the evaluations that are maintained incrementally. This is particularly important for aggregate operators. Consider an expression $e_1 + \ldots + e_n$ and assume that x appears only in e_1. Then

$$\mathop{\mathbb{E}}_{\alpha[x/v]}[e_1 + \ldots + e_n] = \mathop{\mathbb{E}}_{\alpha[x/v]}[e_1] - \mathop{\mathbb{E}}_{\alpha}[e_1]$$

and the differentiable method only needs to evaluate $\mathbb{E}_{\alpha[x/v]}[e_1]$. By induction, differentiable methods then only evaluates the leaves containing the variables to be assigned and the branches from the root to these leaves.

6 Relational Expressions as Differentiable Constraints

This section shows how relational expressions can be translated into constraints. Recall that, in CBLS, a constraint is a differentiable object maintaining its violations and its variable violations, and supporting differentiation. In order to

$$\begin{aligned}
&r \in \quad \textit{Relation}.\\
&r ::= e = e \mid e \leq e \mid e \neq e \mid r \vee r \mid r \wedge r \mid \neg r
\end{aligned}$$

$$\begin{aligned}
\mathbb{V}[e_1 = e_2] &= abs(e_1 - e_2)\\
\mathbb{V}[e_1 \leq e_2] &= \max(e_1 - e_2, 0)\\
\mathbb{V}[e_1 \neq e_2] &= 1 - \min(1, abs(e_1 - e_2))\\
\\
\mathbb{V}[r_1 \wedge r_2] &= \mathbb{V}[r_1] + \mathbb{V}[r_2]\\
\mathbb{V}[r_1 \vee r_2] &= \min(\mathbb{V}[r_1], \mathbb{V}[r_2])\\
\mathbb{V}[\neg r] &= 1 - \min(1, \mathbb{V}[r])
\end{aligned}$$

Fig. 9. Constraints as Objective Functions

```
interface constraint {
  var{int} violations();
  var{int} violations(var{int} x);
  int getAssignDelta(var{int} x,int v);
  int getSwapDelta(var{int} x,var{int} y);
  int getAssignDelta(var{int}[] x,var[] v);
  var{int}[] getVariables();
}
```

Fig. 10. The Constraint Interface in Comet (Partial Description)

translate a relation into a constraint, the key idea is to map the relation r into an expression $\mathbb{V}[r]$ denoting its violations. Once such a mapping $\mathbb{V} : \textit{Relation} \rightarrow \textit{Expression}$ is available, the constraint interface depicted in Figure 10 can be naturally implemented. In particular,

- $\underset{\alpha}{\mathbb{E}}[\mathbb{V}[r]]$ denotes the violations of r for α, incrementally maintained by $i_{\mathbb{V}[r]}$ which is returned by method `violations()` of the constraint interface.
- $\overset{x}{\underset{\alpha}{\downarrow}}\mathbb{V}[r]$ is the variable violations of x for α, incrementally maintained by $i^{\downarrow}_{\mathbb{V}[r]}$ which is returned by method `violations(var{int} x)`.

Figure 9 depicts the syntax of relations and a mapping $\mathbb{V}$ for a variety of relations and logical connectives. For instance, the violations of a relation $e_1 = e_2$ are specified by the expression $\mathbb{V}[e_1 = e_2] = abs(e_1 - e_2)$. The resulting expression can then be transformed into a differentiable objective which incrementally maintains the constraint violations using the invariant $i_{abs(e_1-e_2)}$ and the variable violations using the gradient invariant $i^{\downarrow}_{abs(e_1-e_2)}$. Observe how differentiable invariants preclude the need to derive specific variable violations (as in [1]), since variable violations are directly inherited from violation expressions.

7 Reification

Since expressions and relations can be both transformed into objectives, it becomes natural to support reification in expressions. Reification, a fundamental

$$
\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{E}}[r] & = \delta(\underset{\alpha}{\mathbb{B}}[r]) & i_r & \leftarrow \delta(b_r) \\
\underset{\alpha}{\mathbb{B}}[e_1 = e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] = \underset{\alpha}{\mathbb{E}}[e_2] & b_{e_1 = e_2} & \leftarrow i_{e_1} = i_{e_2} \\
\underset{\alpha}{\mathbb{B}}[e_1 \leq e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \leq \underset{\alpha}{\mathbb{E}}[e_2] & b_{e_1 \leq e_2} & \leftarrow i_{e_1} \leq i_{e_2} \\
\underset{\alpha}{\mathbb{B}}[e_1 \neq e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \neq \underset{\alpha}{\mathbb{E}}[e_2] & b_{e_1 \neq e_2} & \leftarrow i_{e_1} \neq i_{e_2} \\
\underset{\alpha}{\mathbb{B}}[r_1 \vee r_2] & = \underset{\alpha}{\mathbb{B}}[r_1] \vee \underset{\alpha}{\mathbb{B}}[r_2] & b_{r_1 \vee r_2} & \leftarrow b_{r_1} \vee b_{r_2} \\
\underset{\alpha}{\mathbb{B}}[r_1 \wedge r_2] & = \underset{\alpha}{\mathbb{B}}[r_1] \wedge \underset{\alpha}{\mathbb{B}}[r_2] & b_{r_1 \wedge r_2} & \leftarrow b_{r_1} \wedge b_{r_2} \\
\underset{\alpha}{\mathbb{B}}[\neg r] & = \neg \underset{\alpha}{\mathbb{B}}[r] & b_{\neg r} & \leftarrow \neg b_r
\end{array}
$$

Fig. 11. The Evaluation of Reified Constraints and their Corresponding Invariants

$$
\begin{array}{l|l}
\overset{x}{\underset{\alpha}{\downarrow}} r = \text{let } e = \mathbb{V}[r] \text{ in } \delta(\underset{\alpha}{\mathbb{B}}[r] \ \wedge \ \overset{x}{\underset{\alpha}{\uparrow}} e > 0) & i_r^{\downarrow} \leftarrow \delta(b_r \ \wedge \ i^{\uparrow}_{\mathbb{V}[r]} > 0) \\
\overset{x}{\underset{\alpha}{\uparrow}} r = \text{let } e = \mathbb{V}[r] \text{ in } \delta(\neg \underset{\alpha}{\mathbb{B}}[r] \ \wedge \ \overset{x}{\underset{\alpha}{\downarrow}} e \geq \underset{\alpha}{\mathbb{E}}[e]) & i_r^{\uparrow} \leftarrow \delta(\neg b_r \ \wedge \ i^{\downarrow}_{\mathbb{V}[r]} \geq i_{\mathbb{V}[r]})
\end{array}
$$

Fig. 12. The Gradients of Reified Constraints and their Corresponding Invariants

technique in CP, was illustrated in the scene allocation and progressive party problems, in which expressions includes arithmetic operations over relations. It is different from, and more challenging than, the reification from differentiable constraints to differentiable objectives which already presented in [9]. To support reification in CBLS, it is necessary to specify how to evaluate reified expressions and their gradients. Figure 11 depicts the extensions of Figure 6 to handle reification in evaluations and their corresponding invariants. In the figure, $\mathbb{B}_\alpha[e]$ denotes the truth value of expression e under assignment α and b_e denotes the corresponding Boolean invariant. The figure also uses the Kronecker symbol δ to convert Boolean values into 0/1 values:

$$\delta(b) = \begin{cases} 1 & \text{if } b = \mathit{true}; \\ 0 & \text{otherwise.} \end{cases}$$

It remains to define how to evaluate the gradients of reified constraints, which is depicted in Figure 12. The definitions are specified generically using $\mathbb{B}$ and $\mathbb{V}$. The intuition is as follows: given an assignment α, changing x may decrease the evaluation of r if r holds for α (i.e., $\mathbb{B}_\alpha\, r$) and changing x may violate r (i.e., $i^{\uparrow}_{\mathbb{V}[r]} > 0$). Similarly, changing x may increase the evaluation of c if c does not hold for α and changing x may remove all violations of c, i.e.,

$$\overset{x}{\underset{\alpha}{\downarrow}} \mathbb{V}[r] \geq \underset{\alpha}{\mathbb{E}}[\mathbb{V}[r]].$$

Again, the invariants for maintaining gradients can be derived systematically from the evaluations.

Table 1. The Overhead of Differential Invariants

n	6	7	8	9
PP(Atmost)	0.85	1.01	7.46	145.89
PP(DI)	2.18	2.46	12.46	213.29
%Overhead	256.47	243.56	67.20	46.20

It is also interesting to illustrate the expressions obtained for the reified constraints in the progressive party problem. These constraints are of the form

$$(x_1 = y_1) + \ldots + (x_p = y_p) \leq 1$$

where $x_1, \ldots, x_p, y_1, \ldots, y_p$ are all distinct variables. The invariants maintaining the violations are of the form

$$i_c \leftarrow \max(i_m, 0) \qquad i_m \leftarrow i_{\delta(x_1=y_1)} + \ldots + i_{\delta(x_p=y_p)} - 1$$

The gradient for variable x_1 is maintained through invariants of the form

$$i_c^{\downarrow} \leftarrow \max(i_m + i_m^{\downarrow}, 0) - i_c \qquad i_m^{\downarrow} \leftarrow \delta(b_{x_1=y_1} \wedge i^{\uparrow}_{abs(x_1-y_1)} > 0)$$

Observe how differentiable invariants abstract away the complexity behind these constraints, elegantly encompass reification, and allow constraint-based local search to support a constraint language as rich as in traditional CP languages.

8 Experimental Evaluation

This section provides preliminary evidence of the practicability of differentiable invariants. It studies the cost of differentiable invariants and the benefits of gradients invariants, and gives results on the applications.

The Cost of Differentiable Invariants. Table 1 reports the cost of differentiable invariants. It measures the time in seconds for finding solutions to the progressive party problem of increasing sizes (from 6 to 9 periods) when the hosts are boats 1–13. The table compares the COMET program with differentiable invariants (PP(DI) shown in Figure 3) with the same program is replaced by the cardinality operator proposed in [9]. Since there are a quadratic number of these constraints, this is where most of the computation time is spent. Both programs are compared using the deterministic mode of COMET so that they execute exactly the same local moves. The results indicate that the overhead of using differential invariants decreases as the problem size grows and goes down to 46% for the largest instance. Differentiable invariants thus introduces a reasonable overhead compared to a tailored cardinality operator. This overhead should be largely compensated by the simplicity of expressing complex idiosyncratic constraints, which frees programming from implementing special-purpose constraints, objective functions, or combinators.

Table 2. The Benefits of Gradient Invariants

n	6	7	8	9
PP(DI-G)	6.05	6.98	122.76	2777.74
PP(DI)	2.18	2.46	12.46	213.29
Speedup	2.77	2.83	9.85	13.02

Table 3. Experimental Results on Scene Allocation and Balanced Latin Squares

Bench	min(S)	$\mu(S)$	max(S)	$\mu(TS)$	$\sigma(S)$	$\sigma(TS)$
scene	334144.00	335457.38	343256.00	1.10%	0.72	0.06
balance(8)	0	0	0	0.0	13.85	14.74
balance(9)	0	0	0	0.0	61.26	51.78

The Benefits of Gradient Invariants. Table 2 reports experimental results on the benefits on maintaining variable gradients incrementally. It compares the results of the model in Figure 3 when the implementation incrementally updates (PP(DI)) or evaluates (PP(DI-G)) variable gradients. The results highlight the importance of gradient invariants as the speed-ups increase with the problem size to reach a 13-fold improvement on the largest instance.

Other Experimental Results. For completeness, Table 3 reports the experimental results on scene allocation and spatially balanced latin squares. The first three columns report the min, average, and maximal values of the objective function, the fourth column reports the average CPU time (in seconds), and the last two columns show the standard deviation. The scene allocation program, despite its simplicity, performs extremely well and does not need the symmetry-breaking required in the CP solution for good performance (see [7]). The COMET program for latin square is very competitive with the local search algorithms presented in [2] which use a similar neighborhood (but a different search strategy which is not specified precisely enough for reproduction).

Overall, these results show that differentiable invariants are an effective high-level abstraction to bridge the gap between invariants and differentiable objects. They allow programmers to express complex, idiosyncratic constraints declaratively, while leaving the system deriving invariants and incremental algorithms so important in constraint-based local search.

References

1. M. Agren, P. Flener, and J. Pearson. Inferring Variable Conflicts for Local Search. In *CP'06*, September 2006.
2. C. Gomes, M. Sellmann, C. van Es1, and H. van Es. The Challenge of Generating Spatially Balanced Scientific Experiment Designs. In *CP-AI-OR'04*, Nice, 2004.
3. L. Michel and P. Van Hentenryck. Localizer: A Modeling Language for Local Search. In *CP'97)*, October 1997.

4. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.
5. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *OOPSLA-02*, November 2002.
6. C. Smith, C. Gomes, and C. Fernàndez. Streamlining Local Search for Spatially Balanced Latin Squares. In *IJCAI-05*, Edinburgh, Scotland, July 2005.
7. P. Van Hentenryck. Constraint and Integer Programming in OPL. *Informs Journal on Computing*, 14(4):345–372, 2002.
8. P. Van Hentenryck. *Constraint-Based Local Search.* The MIT Press, 2005.
9. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-Based Combinators for Local Search. In *CP'04*, October 2004.

Revisiting the Sequence Constraint

Willem-Jan van Hoeve[1], Gilles Pesant[2,3], Louis-Martin Rousseau[2,3,4], and Ashish Sabharwal[1]

[1] Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853, USA
{vanhoeve, sabhar}@cs.cornell.edu
[2] École Polytechnique de Montréal, Montreal, Canada
[3] Centre for Research on Transportation (CRT), Université de Montréal, C.P. 6128, succ. Centre-ville, Montreal, H3C 3J7, Canada
[4] Oméga Optimisation Inc.
{pesant, louism}@crt.umontreal.ca

Abstract. Many combinatorial problems, such as car sequencing and rostering, feature `sequence` constraints, restricting the number of occurrences of certain values in every subsequence of a given width. To date, none of the filtering algorithms proposed guaranteed domain consistency. In this paper, we present three filtering algorithms for the `sequence` constraint, with complementary strengths. One borrows ideas from dynamic programming; another reformulates it as a `regular` constraint; the last is customized. The last two algorithms establish domain consistency. Our customized algorithm does so in polynomial time, and can even be applied to a generalized `sequence` constraint for subsequences of variable widths. Experimental results show the practical usefulness of each.

1 Introduction

The `sequence` constraint was introduced by Beldiceanu and Contejean [4] as a set of overlapping `among` constraints. The constraint is also referred to as `among_seq` in [3]. An `among` constraint restricts the number of variables to be assigned to a value from a specific set. For example, consider a nurse-rostering problem in which each nurse can work at most 2 night shifts during every 7 consecutive days. The `among` constraint specifies the 2-out-of-7 relation, while the `sequence` constraint imposes such `among` for every subsequence of 7 days.

Beldiceanu and Carlsson [2] have proposed a filtering algorithm for the `sequence` constraint, while Régin and Puget [10] have presented a filtering algorithm for the `sequence` constraint in combination with a global cardinality constraint [8] for a car sequencing application. Neither approach establishes domain consistency, however. As the constraint is inherent to many real-life problems, improved filtering could have a substantial industrial impact.

In this work we present three novel filtering algorithms for the `sequence` constraint. The first is based on dynamic programming concepts and runs in polynomial time, but it does not establish domain consistency. The second algorithm is based on the `regular` constraint [7]. It establishes domain consistency, but needs exponential time in the worst case. In most practical cases it is very efficient however. Our third algorithm establishes domain consistency in polynomial time. It can be applied to a generalized version of the `sequence` constraint, for which the subsequences are of variable length. Moreover the number of occurrences may also vary per subsequence. Each algorithm has advantages over the others, either in terms of (asymptotic) running time or in terms of filtering.

The rest of the paper is structured as follows. Section 2 presents some background and notation on constraint programming. Section 3 recalls and discusses the `among` and `sequence` constraints. Sections 4 to 6 describe filtering algorithms for `sequence`. Section 7 compares the algorithms experimentally. Finally, Section 8 summarizes the contributions of the paper and discusses possible extensions.

2 Constraint Programming Preliminaries

We first introduce basic constraint programming concepts. For more information on constraint programming we refer to [1].

Let x be a variable. The *domain* of x is a set of values that can be assigned to x and is denoted by $D(x)$. In this paper we only consider variables with *finite* domains. Let $X = x_1, x_2, \ldots, x_k$ be a sequence of variables. We denote $D(X) = \bigcup_{1\leq i\leq k} D(x_i)$. A *constraint* C on X is defined as a subset of the Cartesian product of the domains of the variables in X, i.e. $C \subseteq D(x_1)\times D(x_2)\times\cdots\times D(x_k)$. A tuple $(d_1, \ldots, d_k) \in C$ is called a *solution* to C. We also say that the tuple *satisfies* C. A value $d \in D(x_i)$ for some $i = 1, \ldots, k$ is *inconsistent* with respect to C if it does not belong to a tuple of C, otherwise it is *consistent*. C is *inconsistent* if it does not contain a solution. Otherwise, C is called *consistent*.

A *constraint satisfaction problem*, or a *CSP*, is defined by a finite sequence of variables $\mathcal{X} = x_1, x_2, \ldots, x_n$, together with a finite set of constraints $\mathcal{C}$, each on a subsequence of $\mathcal{X}$. The goal is to find an assignment $x_i = d_i$ with $d_i \in D(x_i)$ for $i = 1, \ldots, n$, such that all constraints are satisfied. This assignment is called a *solution to the CSP*.

The solution process of constraint programming interleaves *constraint propagation*, or *propagation* in short, and *search*. The search process essentially consists of enumerating all possible variable-value combinations, until we find a solution or prove that none exists. We say that this process constructs a *search tree*. To reduce the exponential number of combinations, *constraint propagation* is applied to each node of the search tree: Given the current domains and a constraint C, remove domain values that do not belong to a solution to C. This is repeated for all constraints until no more domain value can be removed. The removal of inconsistent domain values is called *filtering*.

In order to be effective, filtering algorithms should be efficient, because they are applied many times during the solution process. Further, they should remove as many inconsistent values as possible. If a filtering algorithm for a constraint C removes *all* inconsistent values from the domains with respect to C, we say that it makes C *domain consistent*. Formally:

Definition 1 (Domain consistency, [6]). *A constraint C on the variables $x_1, \ldots, x_k$ is called* domain consistent *if for each variable x_i and each value $d_i \in D(x_i)$ $(i = 1, \ldots, k)$, there exist a value $d_j \in D(x_j)$ for all $j \neq i$ such that $(d_1, \ldots, d_k) \in C$.*

In the literature, domain consistency is also referred to as *hyper-arc consistency* or *generalized-arc consistency.*

Establishing domain consistency for *binary constraints* (constraints defined on two variables) is inexpensive. For higher arity constraints this is not necessarily the case since the naive approach requires time that is exponential in the number of variables. Nevertheless the underlying structure of a constraint can sometimes be exploited to establish domain consistency much more efficiently.

3 The Among and Sequence Constraints

The `among` constraint restricts the number of variables to be assigned to a value from a specific set:

Definition 2 (Among constraint, [4]). *Let $X = x_1, x_2, \ldots, x_q$ be a sequence of variables and let S be a set of domain values. Let $0 \leq \min \leq \max \leq q$ be constants. Then*

$$\begin{aligned}\texttt{among}(X, S, \min, \max) = \{(d_1, \ldots, d_q) \mid \forall i \in \{1, \ldots, q\}\ d_i \in D(x_i),\\ \min \leq |\{i \in \{1, \ldots, q\}\ :\ d_i \in S\}| \leq \max\}.\end{aligned}$$

Establishing domain consistency for the `among` constraint is not difficult. Subtracting from min, max, and q the number of variables that must take their value in S, and subtracting further from q the number of variables that cannot take their value in S, we are in one of four cases:

1. $\max < 0$ or $\min > q$: the constraint is inconsistent;
2. $\max = 0$: remove values in S from the domain of all remaining variables, making the constraint domain consistent;
3. $\min = q$: remove values not in S from the domain of all remaining variables, making the constraint domain consistent;
4. $\max > 0$ and $\min < q$: the constraint is already domain consistent.

The `sequence` constraint applies the same `among` constraint on every q consecutive variables:

Definition 3 (Sequence constraint, [4]). *Let $X = x_1, x_2, \ldots, x_n$ be an ordered sequence of variables (according to their respective indices) and let S be a set of domain values. Let $1 \leq q \leq n$ and $0 \leq \min \leq \max \leq q$ be constants. Then*

$$\texttt{sequence}(X, S, q, \min, \max) \ = \ \bigwedge_{i=1}^{n-q+1} \texttt{among}(s_i, S, \min, \max),$$

where s_i represents the sequence $x_i, \ldots, x_{i+q-1}$.

In other words, the `sequence` constraint states that every sequence of q consecutive variables is assigned to at least min and at most max values in S. Note that working on each `among` constraint separately, and hence locally, is not as powerful as reasoning globally. In particular, establishing domain consistency on each `among` of the conjunction does not ensure domain consistency for `sequence`.

Example 1. Let $X = x_1, x_2, x_3, x_4, x_5, x_6, x_7$ be an ordered sequence of variables variables with domains $D(x_i) = \{0, 1\}$ for $i \in \{3, 4, 5, 7\}$, $D(x_1) = D(x_2) = \{1\}$, and $D(x_6) = \{0\}$. Consider the constraint $\texttt{sequence}(X, \{1\}, 5, 2, 3)$, i.e., every sequence of five consecutive variables must account for two or three 1's. Each individual `among` is domain consistent but it is not the case for `sequence`: value 0 is unsupported for variable x_7. ($x_7 = 0$ forces at least two 1's among $\{x_3, x_4, x_5\}$, which brings the number of 1's for the leftmost `among` to at least four.)

Establishing domain consistency for the `sequence` constraint is not nearly as easy as for `among`. The algorithms proposed so far in the literature may miss such global reasoning. The filtering algorithm proposed in [10] and implemented in Ilog Solver does not filter out 0 from $D(x_7)$ in the previous example. However in some special cases domain consistency can be efficiently computed: When min equals max, it can be established in linear time. Namely, if there is a solution, then x_i must equal x_{i+q} because of the constraints $a_i + a_{i+1} + \cdots + a_{i+q-1} = \min$ and $a_{i+1} + \cdots + a_{i+q} = \min$. Hence, if one divides the sequence up into n/q consecutive subsequences of size q each, they must all look exactly the same. Thus, establishing domain consistency now amounts to propagating the "settled" variables (i.e. $D(x_i) \subseteq S$ or $D(x_i) \cap S = \varnothing$) to the first subsequence and then applying the previously described algorithm for `among`. Two of the filtering algorithms we describe below establish domain consistency in the general case.

Without loss of generality, we shall consider instances of `sequence` in which $S = \{1\}$ and the domain of each variable is a subset of $\{0, 1\}$. Using an `element` constraint, we can map every value in S to 1 and every other value (i.e., $D(X) \backslash S$) to 0, yielding an equivalent instance on new variables.

4 A Graph-Based Filtering Algorithm

We propose a first filtering algorithm that considers the individual `among` constraints of which the `sequence` constraint is composed. First, it filters the `among` constraints for each sequence of q consecutive variables s_i. Then it filters the conjunction of every pair of consecutive sequences s_i and s_{i+1}. This is presented as SuccessiveLocalGraph (SLG) in Algorithm 1, and discussed below.

4.1 Filtering the `among` Constraints

The individual `among` constraints are filtered with the algorithm FilterLocalGraph. For each sequence $s_i = x_i, \ldots, x_{i+q-1}$ of q consecutive variables in

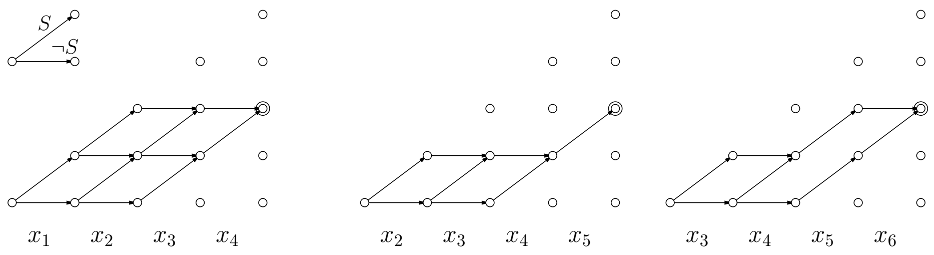


Fig. 1. Filtered Local Graphs of Example 2

$X = x_1, \ldots, x_n$, we build a digraph $G_{s_i} = (V_i, A_i)$ as follows. The vertex set and the arc set are defined as

$$V_i = \{v_{j,k} \mid j \in \{i-1, \ldots, i+q-1\}, k \in \{0, \ldots, j\}\},$$

$$\begin{aligned} A_i = & \{(v_{j,k}, v_{j+1,k}) \mid j \in \{i-1, \ldots, i+q-2\}, k \in \{0, \ldots, j\}, D(x_{j+1}) \setminus S \neq \emptyset\} \cup \\ & \{(v_{j,k}, v_{j+1,k+1}) \mid j \in \{i-1, \ldots, i+q-2\}, k \in \{0, \ldots, j\}, D(x_{j+1}) \cap S \neq \emptyset\}. \end{aligned}$$

In other words, the arc $(v_{j,k}, v_{j+1,k+1})$ represents variable x_{j+1} taking its value in S, while the arc $(v_{j,k}, v_{j+1,k})$ represents variable x_{j+1} not taking its value in S. The index k in $v_{j,k}$ represents the number of variables in $x_i, \ldots, x_{j-1}$ that take their value in S. This is similar to the dynamic programming approach taken in [11] to filter knapsack constraints.

Next, the individual `among` constraint on sequence s_i is filtered by removing all arcs that are not on a path from vertex $v_{i-1,0}$ to a *goal vertex* $v_{i+q-1,k}$ with $\min \leq k \leq \max$. This can be done in linear time (in the size of the graph, $\Theta(q^2)$) by breadth-first search starting from the goal vertices. Naturally, if the filtered graph contains no arc $(v_{j,k}, v_{j+1,k})$ for all k, we remove S from $D(x_{j+1})$. Similarly, we remove $D(X) \setminus S$ from $D(x_{j+1})$ if it contains no arc $(v_{j,k}, v_{j+1,k+1})$ for all k.

Example 2. Let $X = x_1, x_2, x_3, x_4, x_5, x_6$ be an ordered sequence of variables with domains $D(x_i) = \{0, 1\}$ for $i \in \{1, 2, 3, 4, 6\}$ and $D(x_5) = \{1\}$. Let $S = \{1\}$. Consider the constraint `sequence`$(X, S, 4, 2, 2)$. The filtered local graphs of this constraint are depicted in Figure 1.

4.2 Filtering for a Sequence of `among`

We filter the conjunction of two "consecutive" `among` constraints. This algorithm has a "forward" phase and a "backward" phase. In the forward phase, we compare the `among` on s_i with the `among` on s_{i+1} for increasing i, using the algorithm COMPARE. This is done by *projecting* $G_{s_{i+1}}$ onto G_{s_i} such that corresponding variables overlap. Doing so, the projection keeps only arcs that appear in both original local graphs. We can either project vertex $v_{i+1,0}$ of $G_{s_{i+1}}$ onto vertex $v_{i+1,0}$ of G_{s_i}, or onto vertex $v_{i+1,1}$ of G_{s_i}. We consider both projections separately, and label all arcs "valid" if they belong to a path from vertex $v_{i,0}$ to

Algorithm 1. Filtering algorithm for the `sequence` constraint

```
SuccessiveLocalGraph($X, S, q, \min, \max$) begin
    build a local graph $G_{s_i}$ for each sequence $s_i$ $(1 \leq i \leq n - q)$
    for $i = 1, \dots, n - q$ do
        FilterLocalGraph($G_{s_i}$)
    for $i = 1, \dots, n - q - 1$ do
        Compare($G_{s_i}, G_{s_i+1}$)
    for $i = n - q - 1, \dots, 1$ do
        Compare($G_{s_i}, G_{s_i+1}$)
end
FilterLocalGraph($G_{s_i}$) begin
    mark all arcs of $G_{s_i}$ as invalid.
    by breadth-first search, mark as valid every arc on a path from $v_{i-1,0}$ to a goal vertex
    remove all invalid arcs
end
Compare($G_{s_i}, G_{s_{i+1}}$) begin
    mark all arcs in $G_{s_i}$ and $G_{s_{i+1}}$ "invalid"
    for $k = 0, 1$ do
        project $G_{s_{i+1}}$ onto vertex $v_{i,k}$ of $G_{s_i}$
        by breadth-first search, mark all arcs on a path from $v_{i-1,0}$ to a goal vertex in $G_{s_{i+1}}$
        "valid"
    remove all invalid arcs
end
```

goal vertex in $G_{s_{i+1}}$ in one of the composite graphs. All other arcs are labeled "invalid", and are removed from the original graphs G_{s_i} and $G_{s_{i+1}}$. In the backward phase, we compare the `among` on s_i with the `among` on s_{i+1} for decreasing i, similarly to the forward phase.

4.3 Analysis

SuccessiveLocalGraph does not establish domain consistency for the sequence constraint. We illustrate this in the following example.

Example 3. Let $X = x_1, x_2, \dots, x_{10}$ be an ordered sequence of variables with domains $D(x_i) = \{0, 1\}$ for $i \in \{3, 4, 5, 6, 7, 8\}$ and $D(x_i) = \{0\}$ for $i \in \{1, 2, 9, 10\}$. Let $S = \{1\}$. Consider the constraint `sequence`$(X, S, 5, 2, 3)$, i.e., every sequence of 5 consecutive variables must take between 2 and 3 values in S. The first `among` constraint imposes that at least two variables out of $\{x_3, x_4, x_5\}$ must be 1. Hence, at most one variable out of $\{x_6, x_7\}$ can be 1, by the third `among`. This implies that x_8 must be 1 (from the last `among`). Similarly, we can deduce that x_3 must be 1. This is however not deduced by our algorithm.

The problem occurs in the Compare method, when we merge the valid arcs coming from different projection. Up until that point there is a direct equivalence between a path in a local graph and a support for the constraint. However the union of the two projection breaks this equivalence and thus prevents this algorithm from being domain consistent.

The complexity of the algorithm is polynomial since the local graphs are all of size $O(q \cdot \max)$. Hence FilterLocalGraph runs in $O(q \cdot \max)$ time, which is called $n - q$ times. The algorithm Compare similarly runs for $O(q \cdot \max)$ steps and is called $2(n-q)$ times. Thus, the filtering algorithm runs in $O((n-q)\cdot q\cdot \max)$ time. As $\max \leq q$, it follows that the algorithm runs in $O(nq^2)$ time.

5 Reaching Domain Consistency Through `regular`

The `regular` constraint [7], defining the set of allowed tuples for a sequence of variables as the language recognized by a given automaton, admits an incremental filtering algorithm establishing domain consistency. In this section, we give an automaton recognizing the tuples of the `sequence` constraint whose number of states is potentially exponential in q. Through that automaton, we can express `sequence` as a `regular` constraint, thereby obtaining domain consistency.

The idea is to record in a state the last q values encountered, keeping only the states representing valid numbers of 1's for a sequence of q consecutive variables and adding the appropriate transitions between those states. Let Q^q_k denote the set of strings of length q featuring exactly k 1's and $q-k$ 0's — there are $\binom{q}{k}$ such strings. Given the constraint `sequence`$(X, \{1\}, q, \ell, u)$, we create states for each of the strings in $\bigcup_{k=\ell}^{u} Q^q_k$. By a slight abuse of notation, we will refer to a state using the string it represents. Consider a state $d_1 d_2 \ldots d_q$ in $Q^q_k, \ell \leq k \leq u$. We add a transition on 0 to state $d_2 d_3 \ldots d_q 0$ if and only if $d_1 = 0 \vee (d_1 = 1 \wedge k > \ell)$. We add a transition on 1 to state $d_2 d_3 \ldots d_q 1$ if and only if $d_1 = 1 \vee (d_1 = 0 \wedge k < u)$.

We must add some other states to encode the first $q-1$ values of the sequence: one for the initial state, two to account for the possible first value, four for the first two values, and so forth. There are at most 2^q-1 of those states, considering that some should be excluded because the number of 1's does not fall within $[\ell, u]$. More precisely, we will have states

$$\bigcup_{i=0}^{q-1} \bigcup_{k=\max(0,\ell-(q-i))}^{\min(i,u)} Q^i_k.$$

Transitions from a state $d_1 \ldots d_i$ in Q^i_k to state $d_1 \ldots d_i 0$ in Q^{i+1}_k on value 0 and to state $d_1 \ldots d_i 1$ in Q^{i+1}_{k+1} on value 1, provided such states are part of the automaton. Every state in the automaton is considered a final (accepting) state. Figure 2 illustrates the automaton that would be built for the constraint `sequence`$(X, \{1\}, 4, 1, 2)$.

The filtering algorithm for `regular` guarantees domain consistency provided that the automaton recognizes precisely the solutions of the constraint. By construction, the states $Q^q_\star$ of the automaton represent all the valid configurations of q consecutive values and the transitions between them imitate a shift to the right over the sequence of values. In addition, the states $Q^i_\star$, $0 \leq i < q$ are linked so that the first q values reach a state that encodes them. All states are accepting states so the sequence of n values is accepted if and only if the automaton completes the processing. Such a completion corresponds to a successful scan of every subsequence of length q, precisely our solutions.

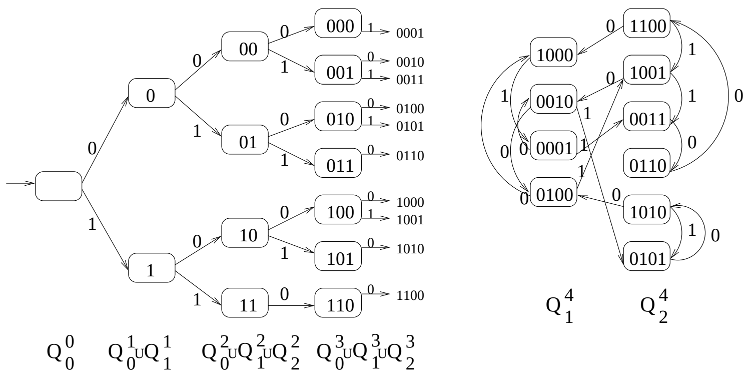


Fig. 2. Automaton for $\texttt{sequence}(X, \{1\}, 4, 1, 2)$

The resulting algorithm runs in time linear in the size of the underlying graph, which has $O(n2^q)$ vertices and arcs in the worst case. Nevertheless, in most practical problems q is much smaller than n. Note also that subsequent calls of the algorithm run in time proportional to the number of updates in the graph and not to the size of the whole graph.

6 Reaching Domain Consistency in Polynomial Time

The filtering algorithms we considered thus far apply to `sequence` constraints with fixed `among` constraints for the same q, min, and max. In this section we present a polynomial-time algorithm that achieves domain consistency in a more generalized setting, where we have m *arbitrary* among constraints over sequences of consecutive variables in X. These m constraints may have different min and max values, be of different length, and overlap in an arbitrary fashion. A conjunction of k `sequence` constraints over the same ordered set of variables, for instance, can be expressed as a *single* generalized sequence constraint. We define the generalized sequence constraint, `gen-sequence`, formally as follows:

Definition 4 (Generalized sequence constraint). *Let* $X = x_1, \ldots, x_n$ *be an ordered sequence of variables (according to their respective indices) and* S *be a set of domain values. For* $1 \le j \le m$*, let* s_j *be a sequence of consecutive variables in* X*,* $|s_j|$ *denote the length of* s_j*, and integers* $\min_j$ *and* $\max_j$ *be such that* $0 \le \min_j \le \max_j \le |s_j|$*. Let* $\Sigma = \{s_1, \ldots, s_m\}$, $\text{Min} = \{\min_1, \ldots, \min_m\}$, *and* $\text{Max} = \{\max_1, \ldots, \max_m\}$*. Then*

$$\texttt{gen-sequence}(X, S, \Sigma, \text{Min}, \text{Max}) \;=\; \bigwedge_{j=1}^{m} \texttt{among}(s_j, S, \min_j, \max_j).$$

For simplicity, we will identify each $s_j \in \Sigma$ with the corresponding `among` constraint on s_j. The basic structure of the filtering algorithm for the `gen-sequence`

Algorithm 2. Complete filtering algorithm for the gen-sequence constraint

```
CompleteFiltering($X$, $S = \{1\}$, $\Sigma$, Min, Max) begin
    for $x_i \in X$ do
        for $d \in D(x_i)$ do
            if CheckConsistency($x_i, d$) = $false$ then
                $D(x_i) \leftarrow D(x_i) \setminus \{d\}$
end

CheckConsistency($x_i, d$) begin
    fix $x_i = d$, i.e., temporarily set $D(x_i) = \{d\}$
    $y[0] \leftarrow 0$
    for $\ell \leftarrow 1, \ldots, n$ do
        $y[\ell] \leftarrow$ number of forced 1's among $x_1, \ldots, x_\ell$
    while a constraint $s_j \in \Sigma$ is violated, i.e., $value(s_j) < \min_j$ or $value(s_j) > \max_j$ do
        if $value(s_j) < \min_j$ then
            $idx \leftarrow$ right end-point of $s_j$
            PushUp($idx, \min_j - value(s_j)$)
        else
            $idx \leftarrow$ left end-point of $s_j$
            PushUp($idx, value(s_j) - \max_j$)
        if $s_j$ still violated then
            return false
    return true
end

PushUp($idx, v$) begin
    $y[idx] \leftarrow y[idx] + v$
    if $y[idx] > idx$ then return false
    // repair $y$ on the left
    while $(idx > 0) \wedge ((y[idx] - y[idx-1] > 1) \vee ((y[idx] - y[idx-1] = 1) \wedge (1 \notin D(x_{idx-1}))))$
    do
        if $1 \notin D(x_{idx-1})$ then
            $y[idx-1] \leftarrow y[idx]$
        else
            $y[idx-1] \leftarrow y[idx] - 1$
        if $y[idx-1] > idx - 1$ then
            return false
        $idx \leftarrow idx - 1$
    // repair $y$ on the right
    while $(idx < n) \wedge ((y[idx] - y[idx+1] > 0) \vee ((y[idx] - y[idx+1] = 0) \wedge (0 \notin D(x_{idx}))))$
    do
        if $0 \notin D(x_{idx})$ then
            $y[idx+1] \leftarrow y[idx] + 1$
        else
            $y[idx+1] \leftarrow y[idx]$
        $idx \leftarrow idx + 1$
end
```

constraint is presented as Algorithm 2. The main loop, CompleteFiltering, simply considers all possible domain values of all variables. If a domain value is yet unsupported, we check its consistency via procedure CheckConsistency. If it has no support, we remove it from the domain of the corresponding variable.

Procedure CheckConsistency is the heart of the algorithm. It finds a single solution to the gen-sequence constraint, or proves that none exists. It uses a single array y of length $n + 1$, such that $y[0] = 0$ and $y[i]$ represents the number of 1's among $x_1, \ldots, x_i$. The invariant for y maintained throughout is that $y[i + 1] - y[i]$ is either 0 or 1. Initially, we start with the lowest possible array, in which y is filled according to the lower bounds of the variables in X.

For clarity, let L_j and R_j denote the left and right end-points, respectively, of the among constraint $s_j \in \Sigma$; $R_j = L_j + |s_j| - 1$. As an example, for the usual sequence constraint with among constraints of size q, L_j would be i and R_j would be $i+q-1$. The *value* of s_j is computed using the array y: value$(s_j) = y[R_j] - y[L_j - 1]$. In other words, value$(s_j)$ counts exactly the number of 1's in the sequence s_j. Hence, a constraint s_j is satisfied if and only if $\min_j \leq$ value$(s_j) \leq \max_j$. In order to find a solution, we consider all among constraints $s_j \in \Sigma$. Whenever a constraint s_j is violated, we make it consistent by "pushing up" either $y[R_j]$ or $y[L_j - 1]$:

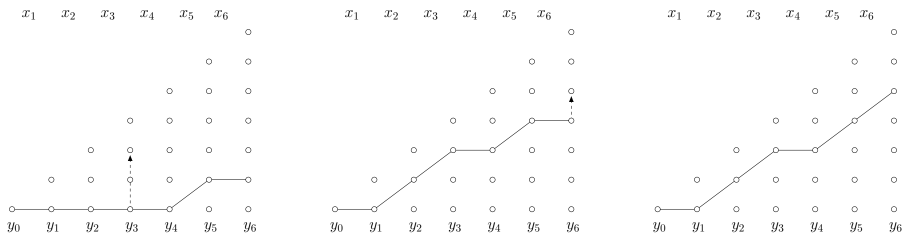
if value$(s_j) < \min_j$, then push up $y[R_j]$ with value $\min_j -$ value(s_j),
if value$(s_j) > \max_j$, then push up $y[L_j - 1]$ with value value$(s_j) - \max_j$.

Such a "push up" may result in the invariant for y being violated. We therefore *repair* y in a minimal fashion to restore its invariant as follows. Let $y[idx]$ be the entry that has been pushed up. We first push up its neighbors on the left side (from idx downward). In case x_{idx-1} is fixed to 0, we push up $y[idx - 1]$ to the same level $y[idx]$. Otherwise, we push it up to $y[idx] - 1$. This continues until the difference between all neighbors is at most 1. Whenever $y[i] > i$ for some i, we need more 1's than there are variables up to i, and we report an immediate failure. Repairing the array on the right side is done in a similar way.

Example 4. Consider again the sequence constraint from Example 2, i.e., the constraint sequence$(X, S, 4, 2, 2)$ with $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $D(x_i) = \{0, 1\}$ for $i \in \{1, 2, 3, 4, 6\}$, $D(x_5) = \{1\}$, and $S = \{1\}$. The four among constraints are over $s_1 = \{x_1, x_2, x_3\}$, $s_2 = \{x_2, x_3, x_4\}$, $s_3 = \{x_3, x_4, x_5\}$, and $s_4 = \{x_4, x_5, x_6\}$. We apply CheckConsistency to find a minimum solution. The different steps are depicted in Figure 3. We start with $y = [0, 0, 0, 0, 0, 1, 1]$, and consider the different among constraints. First we consider s_1, which is violated. Namely, value$(s_1) = y[3] - y[0] = 0 - 0 = 0$, while it should be at least 2. Hence, we push up $y[3]$ with 2 units, and obtain $y = [0, 0, 1, 2, 2, 3, 3]$. Note that we push up $y[5]$ to 3 because x_5 is fixed to 1.

Next we consider s_2 with value $y[4] - y[1] = 2$, which is not violated. We continue with s_3 with value $y[5] - y[2] = 2$, which is not violated. Then we consider s_4 with value $y[6] - y[3] = 1$, which is violated as it should be at least 2. Hence, we push up $y[6]$ by 1, and obtain $y = [0, 0, 1, 2, 2, 3, 4]$. One more loop over the among constraint concludes consistency, with minimum solution $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 1$.

We have optimized the basic procedure in Algorithm 2 in several ways. The main loop of CompleteFiltering is improved by maintaining a support for all domain values. Namely, one call to CheckConsistency (with positive response) yields a support for n domain values. This immediately reduces the number of calls to CheckConsistency by half, while in practice the reduction is even more. A second improvement is achieved by starting out CompleteFiltering with the computation of the "minimum" and the "maximum" solutions to gen-sequence, in a manner very similar to the computation in CheckConsistency but without restricting the value of any variable. This defines bounds



x_1 x_2 x_3 x_4 x_5 x_6
y_0 y_1 y_2 y_3 y_4 y_5 y_6

Fig. 3. Finding a minimum solution to Example 4

y_{min} and y_{max} within which y must lie for all subsequent consistency checks (details in the following section).

6.1 Analysis

A solution to a gen-sequence constraint can be thought of as the corresponding binary sequence or, equivalently, as the y array for it. This y array representation has a useful property. Let y and y' be two solutions. Define array $y \oplus y'$ to be the smaller of y and y' at each point, i.e., $(y \oplus y')[i] = \min(y[i], y'[i])$.

Lemma 1. *If y, y' are solutions to a* gen-sequence *constraint, then so is $y \oplus y'$.*

Proof. Suppose for the sake of contradiction that $y^* = y \oplus y'$ violates an among constraint s of the gen-sequence constraint. Let L and R denote the left and right end-points of s, respectively. Suppose y^* violates the min constraint, i.e., $y^*[R] - y^*[L-1] < \min(s)$. Since y and y' satisfy s, it must be that y^* agrees with y on one end-point of s and with y' on the other. W.l.o.g., assume $y^*[L-1] = y'[L-1]$ and $y^*[R] = y[R]$. By the definition of y^*, it must be that $y[L-1] \geq y'[L-1]$, so that $y[R]-y[L-1] \leq y[R]-y'[L-1] = y^*[R]-y^*[L-1] < \min(s)$. In other words, y itself violates s, a contradiction. A similar reasoning works when y^* violates the max constraint of s. □

As a consequence of this property, we can unambiguously define an absolute *minimum* solution for gen-sequence as the one whose y value is the lowest over all solutions. Denote this solution by y_{min}; we have that for all solutions y and for all i, $y_{min}[i] \leq y[i]$. Similarly, define the absolute *maximum* solution, y_{max}.

Lemma 2. *The procedure* CheckConsistency *constructs the minimum solution to the* gen-sequence *constraint or proves that none exists, in $O(n^2)$ time.*

Proof. CheckConsistency reports success only when no among constraint in gen-sequence is violated by the current y values maintained by it, i.e., y is a solution. Hence, if there is no solution, this fact is detected. We will argue that when CheckConsistency does report success, its y array exactly equals y_{min}.

We first show by induction that y never goes above y_{min} at any point, i.e., $y[i] \leq y_{min}[i], 0 \leq i \leq n$ throughout the procedure. For the base case, $y[i]$ is clearly initialized to a value not exceeding $y_{min}[i]$, and the claim holds trivially. Assume inductively that the claim holds after processing $t \geq 0$ `among` constraint violations. Let s be the $t+1^{st}$ violated constraint processed. We will show that the claim still holds after processing s.

Let L and R denote the left and right end-points of s, respectively. First consider the case that the min constraint was violated, i.e., $y[R] - y[L-1] < \min(s)$, and index $L-1$ was pushed up so that the new value of $y[L-1]$, denoted $\hat{y}[L-1]$, became $y[R] - \min(s)$. Since this was the first time a y value exceeded y_{min}, we have $y[R] \leq y_{min}[R]$, so that $\hat{y}[L-1] \leq y_{min}[R] - \min(s) \leq y_{min}[L-1]$. It follows that $\hat{y}[L-1]$ itself does not exceed $y_{min}[L-1]$. It may still be that the resulting repair on the left or the right causes a y_{min} violation. However, the repair operations only lift up y values barely enough to be consistent with the possible domain values of the relevant variables. In particular, repair on the right "flattens out" y values to equal $\hat{y}[L-1]$ (forced 1's being exceptions) as far as necessary to "hit" the solution again. It follows that since $\hat{y}[L-1] \leq y_{min}[L-1]$, all repaired y values must also not go above y_{min}. A similar argument works when instead the max constraint is violated. This finishes the inductive step.

This shows that by performing repeated PushUp operations, one can never accidentally "go past" the solution y_{min}. Further, since each PushUp increases y in at least one place, repeated calls to it will eventually "hit" y_{min} as a solution.

For the time complexity of CheckConsistency, note that $y[i] \leq i$. Since we monotonically increase y values, we can do so at most $\sum_{i=1}^{n} i = O(n^2)$ times. The cost of each PushUp operation can be charged to the y values it changes because the `while` loops in it terminate as soon as they find a y value that need not be changed. Finally, simple book-keeping can be used to locate a violated constraint in constant time. This proves the desired bound of $O(n^2)$ overall. □

The simple loop structure of CompleteFiltering immediately implies:

Theorem 1. *Algorithm* CompleteFiltering *establishes domain consistency on the* `gen-sequence` *constraint or proves that it is inconsistent, in* $O(n^3)$ *time.*

Remark 1. Régin proved that finding a solution to an arbitrary combination of `among` constraints is NP-complete [9]. Our algorithm finds a solution in polynomial time to a more restricted problem, namely, when each `among` constraint is defined on a sequence of consecutive variables with respect to a fixed ordering.

7 Experimental Results

To evaluate the different filtering algorithms presented, we used two sets of benchmark problems. The first is a very simple model, constructed with only one sequence constraint, allowing us to isolate and evaluate the performance of each method. Then we conduct a limited series of experiments on the well-known car sequencing problem. Successive Local Graph (SLG), Generalized Sequence (GS), and `regular`-based implementation (REG) are compared with the `sequence` constraint provided in the Ilog Solver library in both basic (IB) and

Table 1. Comparison on instances with $n = 100, d = 10$

		IB		IE		SLG		GS		REG	
q	Δ	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
5	1	–	–	33976.9	18.210	0.2	0.069	0	0.014	0	0.009
6	2	361770	54.004	19058.3	6.390	0	0.078	0	0.013	0	0.018
7	1	380775	54.702	113166	48.052	0	0.101	0	0.012	0	0.020
7	2	264905	54.423	7031	4.097	0	0.129	0	0.016	0	0.039
7	3	286602	48.012	0	0.543	0	0.129	0	0.015	0	0.033
9	1	–	–	60780.5	42.128	0.1	0.163	0	0.010	0	0.059
9	3	195391	43.024	0	0.652	0	0.225	0	0.016	0	0.187

Table 2. Comparison on instances with $\Delta = 1, d = 10$

		IB		IE		SLG		GS		REG	
q	n	BT	CPU	BT	CPU	BT	CPU	BT	CPU	BT	CPU
5	50	459154	18.002	22812	18.019	0.4	0.007	0	0.001	0	0.001
5	100	192437	12.008	11823	12.189	1	0.041	0	0.005	0	0.005
5	500	48480	12.249	793	41.578	0.7	1.105	0	0.466	0	0.023
5	1000	942	1.111	2.3	160.000	1.1	5.736	0	4.374	0	0.062
7	50	210107	12.021	67723	12.309	0.2	0.015	0	0.001	0	0.006
7	100	221378	18.030	44963	19.093	0.4	0.059	0	0.005	0	0.010
7	500	80179	21.134	624	48.643	2.8	2.115	0	0.499	0	0.082
7	1000	30428	28.270	46	138.662	588.5	14.336	0	3.323	0	0.167
9	50	18113	1.145	18113	8.214	0.9	0.032	0	0.001	0	0.035
9	100	3167	0.306	2040	10.952	1.6	0.174	0	0.007	0	0.087
9	500	48943	18.447	863	65.769	2.2	4.311	0	0.485	0	0.500
9	1000	16579	19.819	19	168.624	21.9	16.425	0	3.344	0	0.843

extended (IE) propagation modes. Experiments were run with Ilog Solver 6.2 on a bi-processor Intel Xeon HT 2.8Ghz, 3G RAM.

7.1 Single Sequence

To evaluate the filtering both in terms of domain reduction and efficiency, we build a very simple model consisting of only one sequence constraint.

The first series of instances is generated in the following manner. All instances contain n variables of domain size d and the S set is composed of the first $d/2$ elements. We generate a family of instances by varying the size of q and of the difference between min and max, $\Delta = \max - \min$. For each family we try to generate 10 challenging instances by randomly filling the domain of each variable and by enumerating all possible values of min. These instances are then solved using a random choice for both variable and value selection, keeping only the ones that are solved with more than 10 backtracks by method IB. All runs were stopped after one minute of computation.

Table 1 reports on instances with a fixed number of variables (100) and varying q and Δ. Table 2 reports on instances with a fixed Δ (1) and growing number of variables. The results confirm that the new algorithms are very efficient. The average number of backtracks for SLG is generally very low. As predicted by its time complexity, GS is very stable for fixed n in the first table but becomes more time consuming as n grows in the second table. The performance of SLG and REG decreases as q grows but REG remains competitive throughout these

Table 3. Comparison on small car sequencing instances

Version	Average		Median	
	BT	CPU	BT	CPU
A	1067	26.5	0	4.6
B	1067	10.3	0	3.8
C	802	8.4	0	4.1
D	798	34.3	0	7.0

tests. We expect that the latter would suffer with still larger values of q and Δ but it proved difficult to generate challenging instances in that range — they tended to be loose enough to be easy for every algorithm.

7.2 Car Sequencing

In order to evaluate this constraint in a more realistic setting, we turned to the car sequencing problem. We ran experiments using the first set of instances on the CSPLib web site and out of the 78 instances we kept the 31 that could be solved within 5 minutes using a program found in the Ilog distribution. Recall that the Ilog version of the `sequence` constraint also allows to specify individual cardinalities for values in S so it is richer than our version of `sequence` . Table 3 compares the following versions of the sequencing constraint: **(A)** original Ilog program; **(B)** A + REG (added as a redundant constraint); **(C)** A + REG with cost [5], using the cost variable to restrict the total number of cars with a particular option; **(D)** A + REG with cost, using the cost variable to restrict the total number of cars of a particular configuration for each option. For A,B and C we thus introduce one constraint per option and for D we add one constraint per configuration and option.

It is interesting to see that adding REG as a redundant constraint significantly improves performance as it probably often detects a dead end before IloSequence does, thus avoiding expensive work. The simple cost version (C) does quite well since it also incorporates a weak form of cardinality constraint within the `se-quence` constraint. For a fairer comparison, we chose not to compare our two other algorithms as we do not currently have incremental implementations.

8 Discussion

We have proposed, analyzed, and evaluated experimentally three filtering algorithms for the `sequence` constraint. They have different strengths that complement each other well. The local graph approach of Section 4 does not guarantee domain consistency but causes quite a bit of filtering, as witnessed in the experiments. Its asymptotic time complexity is $O(nq^2)$. The reformulation as a `regular` constraint, described in Section 5, establishes domain consistency but its asymptotic time and space complexity are exponential in q, namely $O(n2^q)$. Nevertheless for small q, not uncommon in applications, it performs very well partly due to its incremental algorithm. The generalized sequence approach of

Section 6 also establishes domain consistency on the `sequence` constraint, as well as on a more general variant defined on arbitrary `among` constraints. It has an asymptotic time complexity that is polynomial in both n and q, namely $O(n^3)$. Also in practice this algorithm performed very well, being often even faster than the local graph approach. It should be noted that previously known algorithms did not establish domain consistency.

Since q plays an important role in the efficiency of some of the approaches proposed, it is worth estimating it in some typical applications. For example, in car sequencing values between 2 and 5 are frequent, whereas the shift construction problem may feature widths of about 12.

As a possible extension of this work, our first two algorithms lend themselves to a generalization of `sequence` in which the number of occurrences is represented by a set (as opposed to an interval of values).

Acknowledgments

We thank Marc Brisson and Sylvain Mouret for their help with some of the implementations and experiments, as well as the referees for their comments.

References

1. K.R. Apt. *Principles of Constraint Programming.* Cambridge Univ. Press, 2003.
2. N. Beldiceanu and M. Carlsson. Revisiting the Cardinality Operator and Introducing the Cardinality-Path Constraint Family. In *ICLP 2001*, volume 2237 of *LNCS*, pages 59–73. Springer, 2001.
3. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog. Technical Report T2005-08, SICS, 2005.
4. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.
5. S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*. under final review.
6. R. Mohr and G. Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI)*, pages 651–656, 1988.
7. G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP 2004*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.
8. J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *AAAI/IAAI*, pages 209–215. AAAI Press/The MIT Press, 1996.
9. J.-C. Régin. Combination of Among and Cardinality Constraints. In *CPAIOR 2005*, volume 3524 of *LNCS*, pages 288–303. Springer, 2005.
10. J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In *CP'97*, volume 1330 of *LNCS*, pages 32–46. Springer, 1997.
11. M.A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118:73–84, 2003.

BlockSolve: A Bottom-Up Approach for Solving Quantified CSPs

Guillaume Verger and Christian Bessiere

LIRMM, CNRS/University of Montpellier, France
{verger, bessiere}@lirmm.fr

Abstract. Thanks to its extended expressiveness, the quantified constraint satisfaction problem (QCSP) can be used to model problems that are difficult to express in the standard CSP formalism. This is only recently that the constraint community got interested in QCSP and proposed algorithms to solve it. In this paper we propose BlockSolve, an algorithm for solving QCSPs that factorizes computations made in branches of the search tree. Instead of following the order of the variables in the quantification sequence, our technique searches for combinations of values for existential variables at the bottom of the tree that will work for (several) values of universal variables earlier in the sequence. An experimental study shows the good performance of BlockSolve compared to a state of the art QCSP solver.

1 Introduction

The quantified constraint satisfaction problem (QCSP) is an extension of the constraint satisfaction problem (CSP) in which variables are totally ordered and quantified either existentially or universally. This generalization provides a better expressiveness for modelling problems. Model Checking and planning under uncertainty are examples of problems that can nicely be modeled with QCSP. But such an expressiveness has a cost. Whereas CSP is in NP, QCSP is PSPACE-complete.

The SAT community has also done a similar generalization from the problem of satisfying a Boolean formula into the quantified Boolean formula problem (QBF). The most natural way to solve instances of QBF or QCSP is to instantiate variables from the outermost quantifier to the innermost. This approach is called *top-down*. Most QBF solvers implement top-down techniques. Those solvers lift SAT techniques to QBF. Nevertheless, Biere [1], or Pan and Vardi [2] proposed different techniques to solve QBF instances. Both try to eliminate variables from the innermost quantifier to the outermost quantifier, an approach called *bottom-up*. Biere uses expansion of universal variables into clauses to eliminate them, and Pan and Vardi use symbolic techniques. The bottom-up approach is motived by the fact that the efficiency of heuristics that are used in SAT is lost when following the ordering of the sequence of quantifiers. The drawback of bottom-up approaches is the cost in space.

The interest of the community in solving a QCSP is more recent than QBF, so there are few QCSP solvers. Gent, Nightingale and Stergiou [3] developed QCSP-Solve, a top-down solver that uses generalizations of well-known techniques in CSP like arc-consistency [4,5], intelligent backtracking, and some QBF techniques like the Pure Literal rule. This state-of-the-art solver is faster than previous approaches that transform the QCSP into a QBF problem before calling a QBF solver. Repair-based methods seem to be quite helpful as well, as shown by Stergiou in [6].

In this paper we introduce `BlockSolve`, the first bottom-up algorithm to solve QCSPs. `BlockSolve` instantiates variables from the innermost to the outermost. On the one hand, this permits to factorize equivalent subtrees during search. On the other hand, `BlockSolve` only uses standard CSP techniques, no need for generalizing them into QCSP techniques. The algorithm processes a problem as if it were composed of pieces of classical CSPs. Hence, `BlockSolve` uses the constraint propagation techniques of a standard CSP solver as long as it enforces at least forward checking (FC) [7]. The factorization technique used in `BlockSolve` is very close to that used by Fargier *et al.* for Mixed CSPs [8]. Mixed CSPs are QCSPs in which the sequence of variables is only composed of two consecutive sets, one universally quantified and the other existentially quantified. Fargier *et al.* decomposed Mixed CSPs to solve them using subproblem extraction as in [9]. `BlockSolve` uses this kind of technique, but extends it to deal with any number of alternations of existential and universal variables. Like QBF bottom-up algorithms, `BlockSolve` requires an exponential space to store combinations of values for universal variables that have been proved to extend to inner existential variables. However, storing them in a careful way dramatically decreases this space, as we observed in the experiments.

The rest of the paper is organized as follows. Section 2 defines the concepts that we will use during the paper. Section 3 describes `BlockSolve`, starting by an example and discusses its space complexity. Finally, Section 4 experimentally compares `BlockSolve` to the state-of-the-art QCSP solver QCSP-Solve and Section 5 contains a summary of this work and details for future work.

2 Preliminaries

In this section we define the basic concepts that we will use.

Definition 1 (Quantified Constraint Network). *A* quantified constraint network *is a formula $\mathcal{QC}$ in which:*

- *$\mathcal{Q}$ is a* **sequence** *of quantified variables $Q_i x_i, i \in [1..n]$, with $Q_i \in \{\exists, \forall\}$ and x_i a variable with a domain of values $D(x_i)$,*
- *$\mathcal{C}$ is a conjunction of constraints $(c_1 \wedge ... \wedge c_m)$ where each c_i involves some variables among $x_1, \ldots, x_n$.*

Now we define what is a solution tree of a quantified constraint network.

Definition 2 (Solution tree). *The* solution tree *of a quantified constraint network $\mathcal{QC}$ is a tree such that:*

- *the root node r has no label,*
- *every node s at distance i $(1 \leq i \leq n)$ from the root r is labelled by an instantiation $(x_i \leftarrow v)$ where $v \in D(x_i)$,*
- *for every node s at depth i, the number of successors of s in the tree is $|D(x_{i+1})|$ if x_{i+1} is a universal variable or 1 if x_{i+1} is an existential variable. When x_{i+1} is universal, every value w in $D(x_{i+1})$ appears in the label of one of the successors of s,*
- *for any leaf, the instantiation on $x_1, \ldots, x_n$ defined by the labels of nodes from r to the leaf satisfies all constraints in $\mathcal{C}$.*

It is important to notice that contrary to classical CSPs, variables are ordered as an input of the network. A different order in the sequence $\mathcal{Q}$ gives a different network.

Example 1. The network $\exists x_1 \forall x_2, x_1 \neq x_2, D(x_1) = D(x_2) = \{0, 1\}$ is inconsistent, there is no value for x_1 in $D(x_1)$ that is compatible with all values in $D(x_2)$ for x_2.

Example 2. The network $\forall x_2 \exists x_1, x_1 \neq x_2, D(x_1) = D(x_2) = \{0, 1\}$ has a solution: whatever the value of x_2 in $D(x_2)$, x_1 can be instantiated.

Notice that if all variables are existentially quantified, a solution to the quantified network is a classical instantiation. Hence, the network is a classical constraint network.

Definition 2 leads to the concept of quantified constraint satisfaction problem.

Definition 3 (QCSP). *A* quantified constraint satisfaction problem *(QCSP) is the problem of the existence of a solution to a quantified constraint network.*

We point out that this original definition of QCSP, though different in presentation, is equivalent to previous recursive definitions. The advantage of ours is that it formally specifies what a solution of a QCSP is.

Example 3. Consider the quantified network $\exists x_1 \exists x_2 \forall x_3 \forall x_4 \exists x_5 \exists x_6$, $(x_1 \neq x_5) \wedge (x_1 \neq x_6) \wedge (x_2 \neq x_6) \wedge (x_3 \neq x_5) \wedge (x_4 \neq x_6) \wedge (x_3 \neq x_6), D(x_i) = \{0, 1, 2, 3\}, \forall i$. Figure 1 shows a solution tree for this network.

We define the concept of block, which is the main concept handled by our algorithm `BlockSolve`.

Definition 4 (Block). *A block in a network $\mathcal{QC}$ is a maximal subsequence of variables in $\mathcal{Q}$ that have the same quantifier. We call a block that contains universal variables a* universal block*, and a block that contains existential variables an* existential block*.*

Inverting two variables of a same block does not change the problem, whereas inverting variables of two different blocks changes the problem. If x and y are variables in two different blocks, with x earlier in the sequence than y, we say that x is the *outer* variable and y is the *inner* variable. In this paper, we limit ourselves

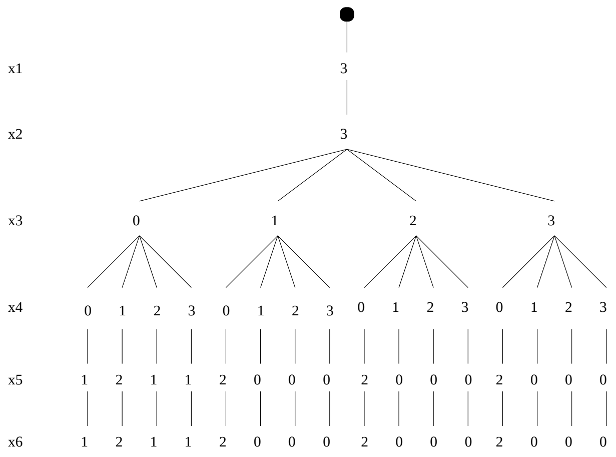


Fig. 1. A solution tree for Example 3

to binary constraints for simplicity of presentation: a constraint involving x_i and x_j is noted c_{ij}. Nevertheless, `BlockSolve` can handle non-binary constraints if they overlap at most two blocks and there is at most one variable in the outer block (if there are two blocks).

The concept of block can be used to define a solution block-tree of a QCSP. This is a compressed version of the solution tree defined above.

Definition 5 (Solution block-tree). *The solution of a quantified constraint network $\mathcal{QC}$ is a tree such that:*

- *the root node r has no label,*
- *every node s at distance i from the root represents the ith block in $\mathcal{Q}$,*
- *every node s at distance i from the root r is labelled by an instantiation of the variables in the ith block if it is an existential block, or by a union of Cartesian products of sub-domains of its variables if it is a universal block,*
- *for every node s at depth i, the number of successors of s in the tree is 1 if the i+1th block is existential, or possibly more than 1 if the i+1th block is universal. When the i+1th block is universal, every combination of values for its variables appears in the label of one of the successors of s,*
- *for any leaf, an instantiation on $x_1, \ldots, x_n$ defined by the labels of the existential nodes from r to the leaf and any of the labels of the universal nodes from r to the leaf satisfies all constraints in $\mathcal{C}$.*

Note that the root node is present only for having a tree and not a forest in cases where the first block is universal. Figure 2 is the block-based version of the solution tree in Fig. 1. The root node is not shown because the first block is existential. The problem is divided in three blocks, the first and the third blocks are existential whereas the second block is universal. Existential nodes are completely instantiated, it means that all variables of those blocks have a single value. The universal block is in three nodes, each one composed of the name of the variables and a union of Cartesian products of sub-domains. Each of the universal nodes represents as many nodes in the solution tree of Fig. 1 as there are tuples in the product. The block-tree in Figure 2 is a compressed version of the tree in Figure 1.

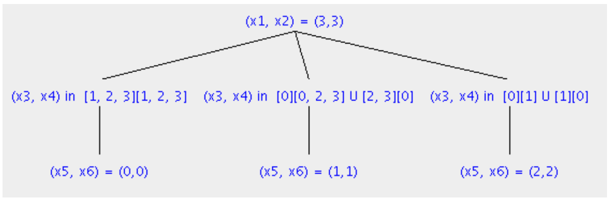

(x1, x2) = (3,3)
(x3, x4) in [1, 2, 3][1, 2, 3]
(x3, x4) in [0][0, 2, 3] U [2, 3][0]
(x3, x4) in [0][1] U [1][0]
(x5, x6) = (0,0)
(x5, x6) = (1,1)
(x5, x6) = (2,2)

Fig. 2. Solution block-tree of example 3

`BlockSolve` uses this concept of blocks for generating a solution and for solving the problem. Blocks divides the problem in levels.

Definition 6 (Level). *A network $\mathcal{P} = \mathcal{QC}$ is divided in p* levels *from* 1 *to p. Each level $k, 1 \leq k \leq p$, is composed of a universal block $block_\forall(k)$, and the following existential block in $\mathcal{Q}$, noted $block_\exists(k)$. If the first block in $\mathcal{Q}$ is existential, the first level contains only this block, and if the last block is universal, the last level contains only this block.*

We call $\mathcal{P}_k$ the subproblem that contains variables in levels k to p and constraints that are defined on those variables. $\mathcal{P}_1$ is the whole problem $\mathcal{P}$. The principle of `BlockSolve` is to solve $\mathcal{P}_p$ first, then using the result to solve $\mathcal{P}_{p-1}$, and so on until it solves $\mathcal{P}_1 = \mathcal{P}$.

3 The `BlockSolve` Algorithm

In this section we describe `BlockSolve`, our QCSP solving algorithm. First of all we run the algorithm on Example 3. Afterwards, we provide the general algorithm.

As done in QCSP-Solve, we start by a preprocessing that permanently removes constraints $\forall x_i \forall x_j \; c_{ij}$ and $\exists x_i \forall x_j \; c_{ij}$. Let us explain why these constraints can be completely removed. For constraints of type $\forall x_i \forall x_j \; c_{ij}$, if there exists a couple (v_i, v_j) of values for x_i and x_j that is forbidden by c_{ij}, then the whole problem is inconsistent. If not, the constraint will ever be satisfied, so we can remove it. For constraints of type $\exists x_i \forall x_j \; c_{ij}$, if there exists a couple (v_i, v_j) of values for x_i and x_j that is forbidden by c_{ij}, then x_i cannot take value v_i. So, we can remove it from the domain of x_i. If $D(x_i)$ becomes empty, the problem is inconsistent. Once all values in $D(x_i)$ have been checked, we can remove c_{ij}. Once the network has been preprocessed this way, the main algorithm can start. `BlockSolve` uses classical propagation techniques, and thus can be integrated into a CSP solver (like Choco [10]). It then inherits all propagation algorithms implemented in the solver. Let us illustrate the behavior of `BlockSolve` on the network of Example 3 before describing how it works.

3.1 Running `BlockSolve` on an Example

In this section we run the algorithm on the network of Example 3 whose solution is presented in Figure 2. The following pictures are an execution of `BlockSolve` on this example.

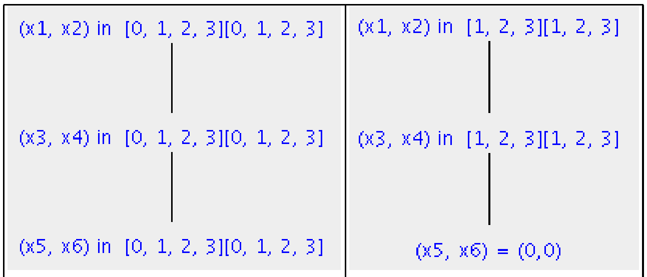
The main idea in `BlockSolve` is to instantiate existential variables of the last block, and to go up to the root instantiating all existential variables. Each assignment v_i of an existential variable x_i can lead to the deletion of inconsistent values of outer variables by propagation. (We illustrate here with FC).

Removing a value of an outer *existential* variable is similar to the CSP case. While the domains of variables are non empty, it is possible to continue instantiating variables. But if a domain is reduced to the empty set, it will be necessary to backtrack on previous choices on inner variables and to restore domains.

Removing a value of an outer *universal* variable implies that we will have to find another instantiation of inner variables that supports this value, because all tuples in universal blocks have to match to a partial solution of inner subproblem. But the instantiation that removes a value in the domain of an universal variable must not be rejected: it can be compatible with a subset of tuples of the universal block. The bigger the size of the subset, the better the grouping. Factorizing tuples of values for a universal block in large groups is a way for minimizing the number of times the algorithm has to solve subproblems. Each time an instantiation of inner variables is found consistent with a subset of tuples for a universal block, we must store this subset and solve again the inner subproblem wrt remaining tuples for the universal variables.

At level k, `BlockSolve` looks for a solution to $\mathcal{P}_{k+1}$, and then tries to solve $\mathcal{P}_k$. The first subproblem `BlockSolve` tries to solve is the innermost subproblem. In the example, `BlockSolve` will instantiate variables of the last block (x_5 and x_6) as if the problem was a classical CSP.

First step.

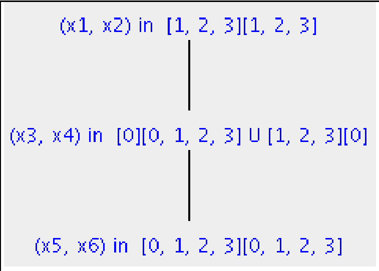

(x1, x2) in [0, 1, 2, 3][0, 1, 2, 3]
(x3, x4) in [0, 1, 2, 3][0, 1, 2, 3]
(x5, x6) in [0, 1, 2, 3][0, 1, 2, 3]
(x1, x2) in [1, 2, 3][1, 2, 3]
(x3, x4) in [1, 2, 3][1, 2, 3]
(x5, x6) = (0,0)

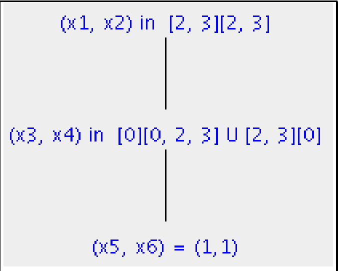
Before the instantiation	x_5 and x_6 instantiated

`BlockSolve` has found an instantiation ($(x_5, x_6) = (0, 0)$) which is consistent with all remaining values of the other variables (thanks to FC filtering). Thus, if there is a consistent assignment for (x_1, x_2) with their remaining values, it is consistent with values that we assigned to (x_5, x_6).

Here, FC removed value 0 for x_3 and x_4, and for x_1 and x_2. It means that BlockSolve has not found an instantiation for (x_5, x_6) that is consistent with tuples in $D(x_3) \times D(x_4)$ that contain 0 for x_3 or for x_4. So, in the next step, BlockSolve tries to find a partial solution on x_5 and x_6 that is consistent with some of the tuples in $\{0\} \times \{0, 1, 2, 3\} \cup \{1, 2, 3\} \times \{0\}$ for x_3 and x_4 (i.e., x_3 or x_4 is forced to take 0).

Second step.

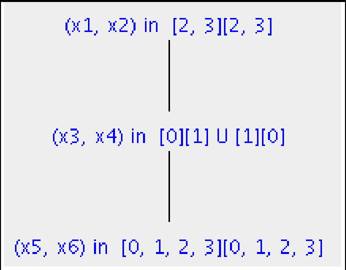

(x1, x2) in [1, 2, 3][1, 2, 3]
(x3, x4) in [0][0, 1, 2, 3] U [1, 2, 3][0]
(x5, x6) in [0, 1, 2, 3][0, 1, 2, 3]
Before the instantiation

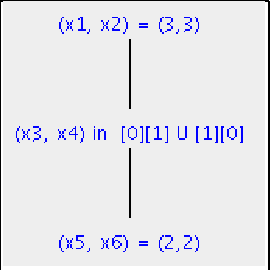


x_5 and x_6 instantiated

In the second step, BlockSolve has found the instantiation $(1, 1)$ for (x_5, x_6), which is consistent with some of the remaining tuples of x_3, x_4. This partial solution $(x_5, x_6) = (1, 1)$ is inconsistent with $(x_3, x_4) = (1, 0)$ and $(x_3, x_4) = (0, 1)$. Note that domains of x_1 and x_2 have been reduced as well.

Last step.


(x1, x2) in [2, 3][2, 3]
(x3, x4) in [0][1] U [1][0]
(x5, x6) in [0, 1, 2, 3][0, 1, 2, 3]
Before the instantiation



x_5 and x_6 instantiated

Finally, in the last step, we find the instantiation $(2, 2)$ for (x_5, x_6), which is consistent with the last remaining combinations for x_3, x_4 (namely, $(0, 1)$ and $(1, 0)$). At this point we know that any combination of values on (x_3, x_4) can be extended to x_5, x_6. The subproblem $\mathcal{P}_2$ is solved. During this process the domains of x_1 and x_2 have been successively reduced until they both reached the singleton $\{3\}$. These are the only values consistent with all instantiations found for x_5, x_6 in the three previous steps. These values 3 for x_1 and 3 for x_2 being compatible (there is no constraint between x_1 and x_2), we know that $\mathcal{P}_1$ $(= \mathcal{P})$ is satisfiable. The solution generated by BlockSolve is the one depicted in Figure 2.

3.2 Description of BlockSolve

In this section, we describe BlockSolve, presented as Algorithm 1. This is a recursive algorithm. BlockSolve(k) is the call of the algorithm at level k, which itself calls BlockSolve($k+1$). In the section above, we saw that it is necessary to keep in memory the tuples of each block. This is saved in two tables: $T_\forall[1..p]$ and $T_\exists[1..p]$ where p is the number of levels. BlockSolve(k) modifies the global tables $T_\forall[]$ and $T_\exists[]$ as side-effects. Local tables A and B are used to restore T adequately depending on success or failure in inner subproblems.

BlockSolve works as follows: for a level k starting from level 1, we try to solve the subproblem $\mathcal{P}_{k+1}$, keeping in mind that it must be compatible with all constraints in $\mathcal{P}$. If there is no solution for $\mathcal{P}_{k+1}$, it means that current values of existential variables in $block_\exists(k)$ do not lead to a solution. But it may be the case that previous choices in $\mathcal{P}_{k+1}$ provoked the removal of those values in $block_\exists(k)$ that lead to a solution with other values in $\mathcal{P}_{k+1}$. So we try to solve $\mathcal{P}_{k+1}$ again, after having removed tuples on $block_\exists(k)$ that led to failure. If there exists a solution for $\mathcal{P}_{k+1}$, we try to instantiate $block_\exists(k)$ with values consistent with some of the tuples on $block_\forall(k)$, exactly as if it was a classical CSP. If success, we remove from $T_\forall[k]$ the tuples on $block_\forall(k)$ that are known to extend on inner variables, and we start again the process on the not yet supported tuples of $block_\forall(k)$. The first call is made with these parameters: $\mathcal{P}_1$ which is the whole problem, and for each level k, the Cartesian products $T_\exists[k]$ and $T_\forall[k]$ of domains of variables in the blocks of level k.

Here we describe the main lines of the algorithm BlockSolve.

At line 1, BlockSolve returns *true* for the empty problem.

At line 2, we test if there remain tuples in $T_\forall[k]$, that is, tuples of $block_\forall(k)$ for which we have not yet found a partial solution in $\mathcal{P}_k$. If empty, it means we have found a partial solution tree for $\mathcal{P}_k$ and we can go up to level $k-1$ (line 13). We also test if we have tried all tuples in $T_\exists[k]$. If yes (i.e., $T_\exists[k] = \emptyset$), it means that $\mathcal{P}_k$ cannot be solved. In this case, we will go up to level $k-1$ (line 13), and we will have to try other values for variables of $block_\exists(k-1)$ (line 12).

At line 4, a call is made to solve $\mathcal{P}_{k+1}$. If the result is true, $T_\forall[i], \forall i \leq k$ and $T_\exists[i], \forall i \leq k$ are tables of tuples that are compatible with the partial solution of $\mathcal{P}_{k+1}$. If the result is false, there is no solution for $\mathcal{P}_{k+1}$ consistent with tuples in $T_\exists[k]$ for existential variables at level k.

At line 5, tuples on $block_\forall(k)$ and $block_\exists(k)$ compatible with $\mathcal{P}_{k+1}$ are saved in $B_\forall$ and $B_\exists$.

From line 6 to line 9, BlockSolve tries to instantiate variables of $block_\exists(k)$ consistently with tuples of $T_\forall[k]$, i.e., tuples of values of universal variables for which it has found a partial solution of $\mathcal{P}_{k+1}$. At line 7, BlockSolve calls solve-level(k) which is presented in Algorithm 2. This is a classical CSP solver that instantiates only existential variables at level k ($block_\exists(k)$) so that the instantiation is compatible with all constraints. This CSP solver has to propagate at least FC to ensure that values of outer variables that are inconsistent with the instantiation are removed. This is due to the fact that we limit constraints to constraints on two blocks, with only one variable in the outermost block. Hence

Algorithm 1. BlockSolve

```
in: $\mathcal{P}$, $k$, $T_\forall[k]$, $T_\exists[k]$
in/out: $T_\forall[1..k-1]$, $T_\exists[1..k-1]$
Result: $true$ if there exists a solution, $false$ otherwise
begin
1    if $k >$ number of levels then return $true$;
2    while $T_\forall[k] \neq \emptyset \wedge T_\exists[k] \neq \emptyset$ do
3        $A_\forall[1..k] \leftarrow T_\forall[1..k]$; $A_\exists[1..k] \leftarrow T_\exists[1..k]$;
4        $solved \leftarrow$ BlockSolve($k+1$);
5        $B_\exists \leftarrow T_\exists[k]$; $B_\forall \leftarrow T_\forall[k]$;
         if $solved$ then
6            repeat
7                $(inst, T_\forall^{Inc}) \leftarrow$solve-level($k$);
8                if $inst$ then $T_\forall[k] \leftarrow T_\forall^{Inc}$;
9            until $T_\forall[k] = \emptyset \vee \neg inst$ ;
10           $solved \leftarrow (B_\forall \neq T_\forall[k])$;
         if $solved$ then
11           $T_\exists[k] \leftarrow A_\exists[k]$; $T_\forall[k] \leftarrow (A_\forall[k] \setminus B_\forall) \cup T_\forall[k]$;
         else
12           $T_\exists[k] \leftarrow A_\exists[k] \setminus B_\exists$ ;
             $T_\exists[1..k-1] \leftarrow A_\exists[1..k-1]$; $T_\forall[1..k] \leftarrow A_\forall[1..k]$;
13   return $(T_\forall[k] = \emptyset)$;
end
```

we ensure that all variables but the outermost are instantiated when propagating a constraint. Each time it finds an instantiation, solve-level(k) removes from $T_\forall[k]$ the tuples not consistent with the instantiation of $block_\exists(k)$, and returns the table $T_\forall^{Inc}$ containing these tuples. This is the new value for $T_\forall[k]$ (line 8). BlockSolve will indeed try to find another instantiation on $block_\exists(k)$ as long as not all tuples in $block_\forall(k)$ compatible with $\mathcal{P}_{k+1}$ (those in $B_\forall$) have found extension to $block_\exists(k)$ or there is no more instantiation which is compatible with $T_\forall[k]$ (i.e., solve-level returns *false*).

If we have extended some new tuples in $block_\forall(k)$ since line 5 (test in line 10), then line 11 updates $T_\forall[k]$: there remains to consider all tuples that have been removed by BlockSolve($k+1$) or solve-level(k) since the beginning of the loop (line 3). For all these tuples of $block_\forall(k)$, BlockSolve has not yet found any partial solution of inner variables consistent with them. Remark that existential variables in $block_\exists(k)$ are restored to their state at the beginning of the loop (line 3).

At line 12, two cases: either *solved* has taken the value $false$ from the call to BlockSolve($k+1$) or it has taken the value $false$ because BlockSolve($k+1$) has found a solution but there was no possible instantiation of variables in $block_\exists(k)$ with a tuple of $B_\exists$ compatible with tuples of $B_\forall$. In both cases no tuple in $B_\exists$ can lead to a partial solution while universal variables of $block_\forall(k)$ have their values

Algorithm 2. `solve-level`(k)

```
in: $\mathcal{P}$,$k$, $T_\forall[k]$,$T_\exists[k]$
in/out: $T_\exists[1..k-1]$, $T_\forall[1..k-1]$
Result: a couple $(inst, T_\forall^{Inc})$:
begin
    Instantiate variables of $block_\exists(k)$ consistently with $T_\forall[]$ and $T_\exists[]$ and
    propagate the constraints;
    if success then
        $T_\forall^{Inc} \leftarrow$ tuples in $T_\forall[k]$ that are inconsistent with the instantiation;
        return (true, $T_\forall^{Inc}$);
    else return (false, —);
end
```

in $B_\forall$. But there might exist a solution on $B_\forall$ consistent with some other tuples of $A_\exists[k]$ (tuples that have been removed because of choices in `BlockSolve`$(k+1)$). We update $T_\exists[k]$ to contain them.

We should bear in mind that function `solve-level` (Algorithm 2) is a standard CSP solving algorithm that tries to instantiate existential variables of $block_\exists(k)$. If it is possible to instantiate them, *inst* is *true* and $T_\forall^{Inc}$ contains all tuples of $T_\forall[k]$ that are in conflict with the instantiation. Maintaining consistency with outer variables is done as side-effects on tables $T_\exists[]$ and $T_\forall[]$.

`BlockSolve` can give the solution block-tree as in Definition 5. Figure 2 shows the result given. In order to build such a tree, `BlockSolve` takes as parameter a node that corresponds to the existential block of the previous level (or the root for the first level). When solving $\mathcal{P}_{k+1}$, `BlockSolve` produces a tree that is plugged to the current existential node. Plugging the current sub-tree can be done after the call to `solve-level` (line 7), using the instantiation of the variables in $T_\exists[k]$, and the compatible tuples in $T_\forall[k]$. If solving $\mathcal{P}_{k+1}$ fails, nothing is plugged.

3.3 Spatial Complexity

`BlockSolve` needs more space than a top-down algorithm like QCSP-Solve. It keeps in memory all tuples of existential and universal blocks for which a solution has not yet been found. The size of such sets can be exponential in the number of variables of the block. But when solving a QCSP, the user usually prefers to obtain a solution tree than an answer: *"yes, it is satisfiable"*. Since a solution tree takes exponential space, any algorithm that returns a solution tree of a quantified network requires exponential space.[1]

`BlockSolve` keeps sets of tuples as unions of Cartesian products, which uses far less space than tuples in extension. In addition, computing the difference between two unions of Cartesian products is much faster than with tuples in extension.

[1] We can point out that a solution can be returned in polynomial space if we allow interactive computation: the values of the first existential block are returned, then, based on the values chosen adversarially for the first universal block, values of the second existential block are returned, and so on.

4 Experiments

In this section we compare QCSP-Solve and `BlockSolve` on random problems. The experiments show the differences between these two algorithms in CPU time and number of visited nodes.

4.1 Coding `BlockSolve`

`BlockSolve` is developed in Java using Choco as constraint library [10]. This library provides different propagation algorithms and a CSP solver. After loading the data of a problem, `BlockSolve` creates tables of sets of tuples for each block and finally launches the main function.

Heuristics. The algorithm uses a value ordering heuristic to increase efficiency. Because `BlockSolve` is able to factorize subtrees, the most efficient way to solve a QCSP is to minimize the number of subtrees during the search. One way to accomplish this is to select the value v of variable x that is compatible with the largest set of tuples of outer blocks. In order to determine which value is the best according to this criterion, the solver instantiates x to all its values successively. For each value it propagates the instantiation to others domains and computes the number of branches it is in (i.e., the number of compatible tuples in outer blocks). The larger the better.

4.2 The Random Problem Generator

Instances of QCSP presented in these experiments have been created with a generator based on that used in [3]. In this model, problems are composed of three blocks, the first is an existential block. It takes seven parameters as input $< n, n_\forall, n_\exists, d, p, q_{\forall\exists}, q_{\exists\exists} >$ where n is the total number of variables, $n_\forall$ is the number of universal variables, $n_\exists$ is the number of existential variables in the first block, d is the size of domains of variables (the same for each variable), p is the number of binary constraints as a fraction of all possible constraints. All constraints are $\forall x_i \exists x_j \; c_{ij}$ constraints or $\exists x_i \exists x_j \; c_{ij}$ constraints, other type of constraints that can be removed during the preprocessing are not generated. $q_{\forall\exists}$ and $q_{\exists\exists}$ are the looseness of constraints, i.e., the number of *goods* as a fraction of all possible couples of values. To avoid the flaw with which almost all problems are inconsistent, constraints $\forall x_i \exists x_j \; c_{ij}$ have very few forbidden couples.

We extended this generator to allow more than 3 blocks. We added an eighth parameter, $b_\forall$, that is the number of universal blocks. Variables are sequenced as follow: $n_\exists$ existential variables followed by $n_\forall$ universal variables, then again $n_\exists$ existential variables followed by $n_\forall$ universal variables.

4.3 Results

Now we present some results on randomly generated instances created by the generator. For each experiment, 100 instances were generated for each value of $q_{\exists\exists}$, from 0.05 to 0.95.

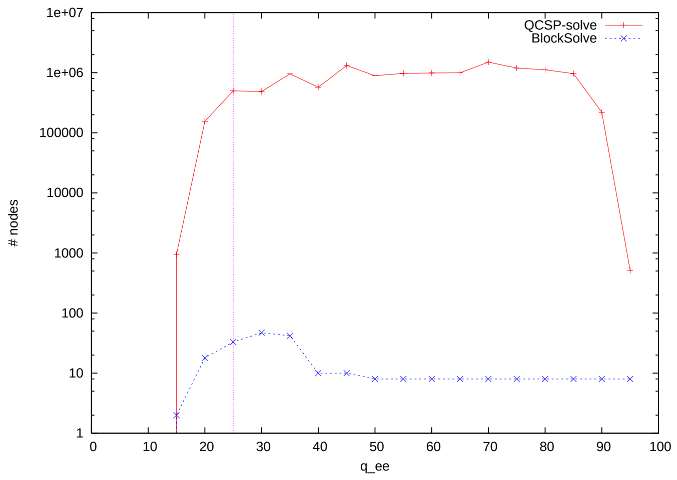


Fig. 3. Number of nodes for $n = 15, n_\forall = 7, n_\exists = 4, b_\forall = 1, d = 15, p = 30, q_{\forall\exists} = .50$ where $q_{\exists\exists}$ grows. (cross-over point at $q_{\exists\exists} = .25$).

The first three figures are from experiments on problems with these characteristics: each instance contains 15 variables, one block of 7 universal variables. Figure 3 shows the results in terms of number of nodes explored. In the leftmost part of the figure, both algorithms detect inconsistency before exploring any node. As we can see, `BlockSolve` traverses far less nodes than QCSP-Solve. Notice that `BlockSolve` explores only existential nodes, not universal ones. QCSP-Solve seems to have difficulties to solve problems that have solutions. On under-constrained problems, `BlockSolve` finds a solution without any backtrack. This means that for easy problems, there exists an instantiation of the innermost existential block that is consistent with all tuples of the universal block.

Figure 4 shows the results in term of CPU time. Comparing it to Figure 3, it is clear that `BlockSolve` takes a lot of time for exploring one node. This is because at each node `BlockSolve` looks for the best value for matching more tuples in outer universal blocks. This heuristic is quite long to compute compared to what QCSP-Solve does at a node. Note that QCSP-Solve determines faster than `BlockSolve` that a problem is inconsistent ($q_{\exists\exists} < .25$), but `BlockSolve` finds a solution much faster when it exists ($q_{\exists\exists} > .25$). For very high values of $q_{\exists\exists}$ ($> .90$), QCSP-Solve is more efficient than `BlockSolve`.

It is interesting to see that `BlockSolve` is more stable than QCSP-Solve, as shown in Figure 5. In this figure, each point represents an instance of problem. We see that in the satisfiable region, it is hard to predict how much time will take QCSP-Solve to solve an instance.

We ran both algorithms on instances that have more than three blocks. Figure 6 presents results for instances that have five blocks (∃∀∃∀∃) of five variables each (left graph), and instances that have seven blocks of four variables each (right

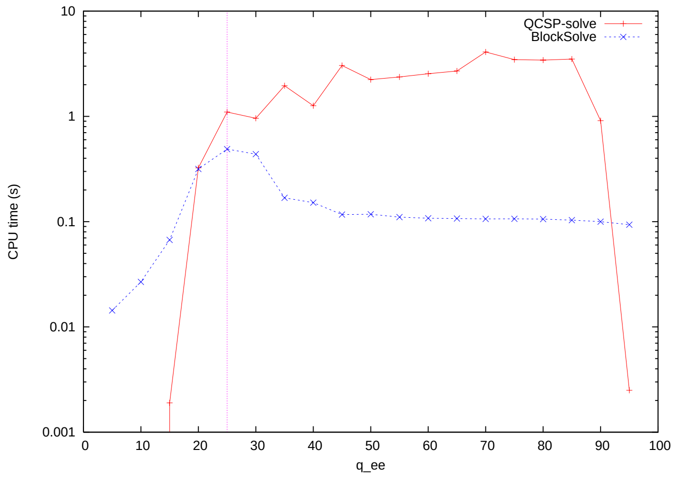


Fig. 4. cpu time for $n = 15, n_\forall = 7, n_\exists = 4, b_\forall = 1, d = 15, p = 30, q_{\forall\exists} = .50$ where $q_{\exists\exists}$ grows. (cross-over point at $q_{\exists\exists} = .25$).

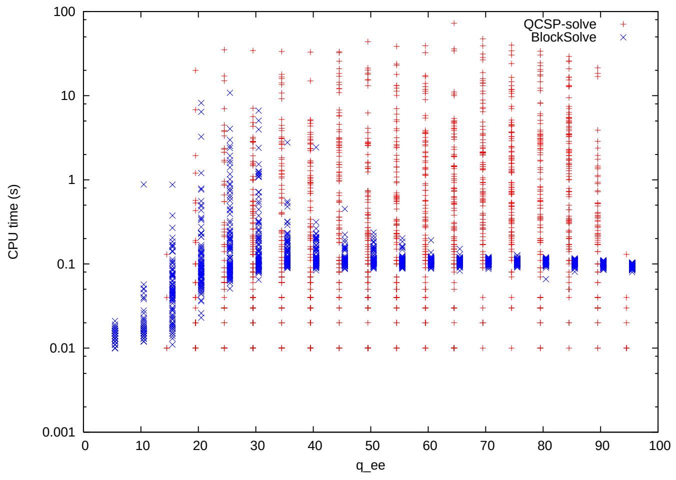
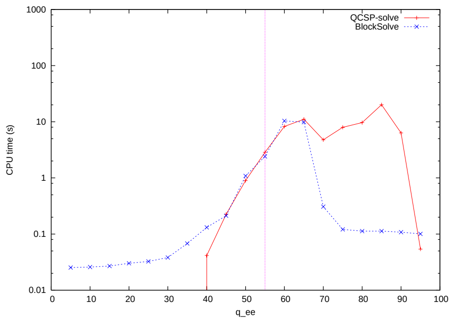
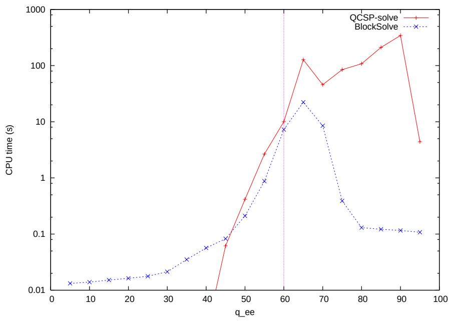


Fig. 5. Scattered plot for cpu time on $n = 15, n_\forall = 7, n_\exists = 4, b_\forall = 1, d = 15, p = 30,$ $q_{\forall\exists} = .50$

graph). These experiments show that the general behavior of both algorithms looks similar whatever the number of levels in the problem. In unsolvable problems, QCSP-Solve detects inconsistency before BlockSolve, whereas for solvable instances QCSP-Solve takes as much time as at the cross-over point (i.e., where there are as many satisfiable instances as unsatisfiable ones).



cpu time for $n = 25, n_\forall = 5, n_\exists = 5$,
$b_\forall = 2, d = 8, p = 20, q_{\forall\exists} = .50$
(cross-over point at $q_{\exists\exists} = .55$)



cpu time for $n = 28, n_\forall = 4, n_\exists = 4$,
$b_\forall = 3, d = 8, p = 20, q_{\forall\exists} = .50$
(cross-over point at $q_{\exists\exists} = .60$)

Fig. 6. Problems with 25 variables and 5 blocks (left) and 28 variables and 7 blocks (right)

5 Conclusions

In this paper we presented `BlockSolve`, a bottom-up QCSP solver that uses standard CSP techniques. Its specificity is that it treats variables from leaves to root in the search tree, and factorizes lower branches avoiding the search in subtrees that are equivalent. The larger this factorization, the better the algorithm, thus minimizing the number of nodes visited. Experiments show that grouping branches gives `BlockSolve` a great stability in time spent and in number of nodes visited. The number of nodes `BlockSolve` visits is much smaller than the number of nodes visited by QCSP-Solve in almost all instances.

Future work will focus on improving time efficiency of `BlockSolve`. Great improvements can probably be obtained by designing heuristics to efficiently prune subtrees that are inconsistent. Furthermore, most of the cpu time is spent updating and propagating tables of tuples on blocks. Finding better ways to represent them could significantly decrease the cpu time of `BlockSolve`. The current implementation of `BlockSolve` being far from being optimized, this leaves a lot of space for significant improvements. Finally, we plan to generalize `BlockSolve` to global constraints.

Acknowledgements

We are very grateful to Kostas Stergiou, Peter Nightingale and Ian Gent, who kindly provided us with the code of QCSP-Solve.

References

1. Biere, A.: Resolve and expand. In: Proceedings SAT'04, Vancouver BC (2004)
2. Pan, G., Vardi, M.: Symbolic decision procedures for QBF. In: Proceedings CP'04, Toronto, Canada (2004) 453–467

3. Gent, I., Nightingale, P., Stergiou, K.: QCSP-solve: A solver for quantified constraint satisfaction problems. In: Proceedings IJCAI'05, Edinburgh, Scotland (2005) 138–143
4. Bordeaux, L., Montfroy, E.: Beyond NP: Arc-consistency for quantified constraints. In: Proceedings CP'02, Ithaca NY (2002) 371–386
5. Mamoulis, N., Stergiou, K.: Algorithms for quantified constraint satisfaction problems. In: Proceedings CP'04, Toronto, Canada (2004) 752–756
6. Stergiou, K.: Repair-based methods for quantified csps. In: Proceedings CP'05, Sitges, Spain (2005) 652–666
7. Haralick, R., Elliott, G.: Increasing tree seach efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313
8. Fargier, H., Lang, J., Schiex, T.: Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. In: Proceedings AAAI'96, Portland OR (1996) 175–180
9. Freuder, E., Hubbe, P.: Extracting constraint satisfaction subproblems. In: Proceedings IJCAI'95, Montréal, Canada (1995) 548–557
10. Choco: A Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving. URL: http://choco-solver.net (2005)

General Symmetry Breaking Constraints

Toby Walsh

National ICT Australia and School of CSE,
University of New South Wales, Sydney, Australia
tw@cse.unsw.edu.au

Abstract. We describe some new propagators for breaking symmetries in constraint satisfaction problems. We also introduce symmetry breaking constraints to deal with symmetries acting simultaneously on variables and values, conditional symmetries, as well as symmeties acting on set and other types of variables.

1 Introduction

Symmetry occurs in many constraint satisfaction problems. We must deal with symmetry or we will waste much time visiting symmetric solutions, as well as parts of the search tree which are symmetric to already visited parts. One mechanism to deal with symmetry is to add constraints which eliminate symmetric solutions [1]. Crawford *et al.* have presented [2] a simple method for breaking any type of symmetry between variables. We pick an ordering on the variables, and then post symmetry breaking constraints to ensure that the final solution is lexicographically less than any symmetric re-ordering of the variables. That is, we select the "lex leader" assignment. Whilst this method was defined for symmetries of Boolean variables, it has been lifted to symmetries of non-Boolean variables, and symmetries acting independently on variables and values [3,4]. In this paper, we show how this basic method can be extended further to symmetries acting simultaneously on variables and values, conditional symmetries, as well as symmeties acting on set and other types of variables.

2 Background

A constraint satisfaction problem consists of a set of variables, each with a domain of values, and a set of constraints specifying allowed combinations of values for given subsets of variables. A solution is an assignment of values to variables satisfying the constraints. Finite domain variables take one value from a given finite set. Set variables take sets of such values and are typically defined by a lower bound on the definite elements and an upper bound on the definite and potential elements. We write $[X_1, \ldots, X_n]$ for the vector of *values* taken by variables X_1 to X_n. Systematic constraint solvers explore partial assignments enforcing some level of local consistency property. We consider two of the most common local consistencies: arc consistency and bound consistency. Given a constraint C on finite domain variables, a *support* is assignment to each variable of a value in its domain which satisfies C. A constraint C on finite domains variables is *generalized arc consistent* (*GAC*) iff for each variable, every value in its domain belongs to a support. Given a constraint C on set variables, a *bound support* on C is an assignment of a set to each set variable between its lower and upper bounds which satisfies C. A constraint C is *bound consistent* (*BC*) iff for each set variable S, the values in $ub(S)$ belong to S in at least one bound support and the values in $lb(S)$ belong to S in all bound supports.

3 Variable Symmetry Breaking

A variable symmetry σ is a bijection on variables that preserves solutions. That is, if $\{X_i = v_i \mid 1 \leq i \leq n\}$ is a solution, then $\{X_{\sigma(i)} = v_i \mid 1 \leq i \leq n\}$ is also. Crawford *et al.* [2] show how to break *all* variable symmetry by posting the constraint $\textsc{LexLeader}(\sigma, [X_1, \ldots, X_n])$ for each variable symmetry σ. This ensures:

$$[X_1, \ldots, X_n] \leq_{\text{lex}} [X_{\sigma(1)}, \ldots, X_{\sigma(n)}]$$

Where X_1 to X_n is any fixed ordering on the variables. To enforce GAC on such a constraint, we can use the lex propagator described in [5] which copes with repeated variables. This takes $O(nm)$ time where m is the maximum domain size. In general, this decomposition into a set of LexLeader constraints hinders propagation. There may be values which can be pruned because they do not occur in all lex leaders which we will not identify by looking at the lex leader constraints individually.

Theorem 1. *$GAC(\bigwedge_{\sigma \in \Sigma} \textsc{LexLeader}(\sigma, [X_1, \ldots, X_n]))$ is strictly stronger than $GAC(\textsc{LexLeader}(\sigma, [X_1, \ldots, X_n]))$ for all $\sigma \in \Sigma$.*

Proof: Clearly it is at least as strong. We show strictness with just two symmetries and four variables. Consider $X_1, X_2, X_4 \in \{0, 1\}$, $X_3 = 1$ and two symmetries defined by the cycles $(1\,4\,2\,3)$ and $(1\,2\,4\,3)$. Then $[X_1, X_2, X_3, X_4] \leq_{\text{lex}} [X_4, X_3, X_1, X_2]$ and $[X_1, X_2, X_3, X_4] \leq_{\text{lex}} [X_2, X_4, X_1, X_3]$) are GAC. However, enforcing GAC on their conjunction prunes 0 from X_4 as the only support for $X_3 = 1$ and $X_4 = 0$ that satisfies $[X_1, X_2, X_3, X_4] \leq_{\text{lex}} [X_4, X_3, X_1, X_2]$ is $X_1 = X_2 = 0$, whilst the only support for $X_3 = 1$ and $X_4 = 0$ that satisfies $[X_1, X_2, X_3, X_4] \leq_{\text{lex}} [X_2, X_4, X_1, X_3]$) is $X_1 = 0$ and $X_2 = 1$. ◇

Breaking all symmetry may require an exponential number of LexLeader constraints (e.g. the $n!$ symmetries of n indistinguishable variables). It may therefore be worth developing a global constraint that combines together several LexLeader constraints. For example, the lex chain constraint [6] breaks all row symmetries of a matrix model. This would require an exponential number of LexLeader constraints. Unfortunately, there are also cases where such a combined symmetry breaking constraint is intractable. For example, a global constraint that combines together all the LexLeader constraints for row and column symmetries of a matrix model is NP-hard to propagate completely [7].

4 Value Symmetry Breaking

A value symmetry σ is a bijection on values that preserves solutions. That is, if $\{X_i = v_i \mid 1 \leq i \leq n\}$ is a solution, then $\{X_i = \sigma(v_i) \mid 1 \leq i \leq n\}$ is also. The lex leader method can also be used to break value symmetries [3]. We simply post the constraint $\textsc{ValLexLeader}(\sigma, [X_1, \ldots, X_n])$ for each value symmetry σ. This holds iff:

$$[X_1, \ldots, X_n] \leq_{\text{lex}} [\sigma(X_1), \ldots, \sigma(X_n)]$$

Again X_1 to X_n is any fixed ordering on the variables. We first prove that such constraints break all symmetry.

Theorem 2. $\textsc{ValLexLeader}(\sigma, [X_1, \ldots, X_n])$ *for all* $\sigma \in \Sigma$ *leaves exactly one solution in each symmetry class.*

Proof: Immediate as lex is a total ordering. $\diamond$

For example, suppose we have a reflection symmetry on the values 1 to m defined by $\sigma_r(i) = m - i + 1$. We can break this symmetry with a single $\textsc{ValLexLeader}$ constraint. If m is even, $\textsc{ValLexLeader}(\sigma_r, [X_1, \ldots, X_n])$ simplifies to the logically equivalent constraint: $X_1 \leq \frac{m}{2}$. If m is odd, it simplifies into the constraints: $X_1 \leq \frac{m+1}{2}$, if $X_1 = \frac{m+1}{2}$ then $X_2 \leq \frac{m+1}{2}$, if $X_1 = X_2 = \frac{m+1}{2}$ then $X_3 \leq \frac{m+1}{2}, \ldots$, and if $X_1 = \ldots = X_{n-1} = \frac{m+1}{2}$ then $X_n \leq \frac{m+1}{2}$.

It is an interesting open question how to simplify $\textsc{ValLexLeader}$ constraints for other types of value symmetries. We conjecture that we may be able to apply some of Puget's ideas from [8]. When there are few value symmetries, Puget has proposed decomposing $\textsc{ValLexLeader}$ into $\textsc{Element}$ and lex ordering constraints [4]. However, enforcing GAC on such a decomposition does not achieve GAC on the $\textsc{ValLexLeader}$ constraint. We show here how to enforce GAC in $O(nm)$ time for each value symmetry. One possibility is to adapt the lex propagator from [9]. Alternatively we decompose $\textsc{ValLexLeader}$ into ternary constraints using a set of Boolean variables, B_i. We identify two important sets of values: $\mathcal{L}$ are those values for which $v < \sigma(v)$, whilst $\mathcal{E}$ are those values for which $v = \sigma(v)$. We post the sequence of constraints, $C(X_i, B_i, B_{i+1})$ for $1 \leq i \leq n$, where $B_1 = 0$ and $C(X_i, B_i, B_{i+1})$ holds iff $B_i = B_{i+1} = 0$ and $X_i \in \mathcal{E}$, or $B_i = 0$, $B_{i+1} = 1$ and $X_i \in \mathcal{L}$, or $B_i = B_{i+1} = 1$. We can enforce GAC on the ternary constraint C using a table constraint, GAC-schema or logical primitives like implication. As the constraint graph of the decomposition is Berge-acyclic, enforcing GAC on each ternary constraint achieves GAC on the whole $\textsc{ValLexLeader}$ constraint [10].

As with variable symmetry, decomposition into individual symmetry breaking constraints may hinder propagation.

Theorem 3. $GAC(\bigwedge_{\sigma \in \Sigma} \textsc{ValLexLeader}(\sigma, [X_1, \ldots, X_n]))$ *is strictly stronger than* $GAC(\textsc{ValLexLeader}(\sigma, [X_1, \ldots, X_n]))$ *for all* $\sigma \in \Sigma$.

Proof: Clearly it is at least as strong. We show strictness with just two symmetries and two variables. Suppose $X_1 \in \{0, 1\}$ and $X_2 \in \{0, 2\}$. Consider a value symmetry σ defined by the cycle $(0\,2)$. Then $\textsc{ValLexLeader}(\sigma, [X_1, X_2]))$ is GAC. Consider a second value symmetry σ' defined by the cycle $(1\,2)$. Then

ValLexLeader($\sigma', [X_1, X_2]$)) is GAC. However, enforcing GAC on the conjunction of these two symmetry breaking constraints prunes 2 from X_2. ◇

Breaking all value symmetry may again introduce an exponential number of symmetry breaking constraints (e.g. the $n!$ symmetries of n indistinguishable values). It may therefore be worth developing a specialized global constraint that combines together several ValLexLeader constraints. For example, as we discuss later, the global precedence constraint [11] combines together all the ValLexLeader constraints for indistinguishable values. As a second example, Puget has recently proposed a forward checking algorithm for any combination of value symmetries [4].

5 Variable and Value Symmetry Breaking

We may have both variable and value symmetries. Consider, for example, a model of the rehearsal problem (prob039 in CSPLib) in which variables represent time slots and values are the pieces practised in each time slot. This model has variable symmetry since any rehearsal can be reversed, as well as value symmetry since any piece requiring the same players is indistinguishable. Can we simply combine the appropriate LexLeader and ValLexLeader constraints?

If each symmetry breaking constraint considers the variables in different orders, it may not be safe to combine them. For example, if σ reflects 1 and 2, and $X_1, X_2 \in \{1, 2\}$ then LexLeader($\sigma, [X_1, X_2]$) and ValLexLeader($\sigma, [X_2, X_1]$) eliminate the assignment $X_1 = 1$ and $X_2 = 2$, as well as all its symmetries. We assume therefore in what follows that the lexicographical ordering within each symmetry breaking constraint considers the variables X_1 to X_n in the same ordering, and is the lifting of the same ordering on values to an ordering on tuples. Variable and value symmetry breaking constraints can then be combined safely.

Theorem 4. *If X_1 to X_n have a set of variable symmetries Σ and a set of value symmetries Σ' then posting* LexLeader($\sigma, [X_1, \dots, X_n]$) *for all $\sigma \in \Sigma$ and* ValLexLeader($\sigma', [X_1, \dots, X_n]$) *for all $\sigma' \in \Sigma'$ leaves one or more assignments in each equivalence class.*

Proof: Consider any assignment. Pick the lex leader under Σ of this assignment. By construction, this satisfies LexLeader($\sigma, [X_1, \dots, X_n]$) for all $\sigma \in \Sigma$. Now consider the lex leader under Σ' of this current assignment. By construction, this satisfies ValLexLeader($\sigma', [X_1, \dots, X_n]$) for all $\sigma' \in \Sigma'$. This also moves us down the lexicographical order on tuples. However, we may no longer satisfy LexLeader($\sigma, [X_1, \dots, X_n]$). We therefore pick the lex leader under Σ of our current assignment. We again must move down the lexicographical order on tuples. This process terminates as the lexicographical order is well founded and bounded by $[0, \dots, 0]$ where 0 is the least value. We terminate with an assignment that satisfies both the LexLeader and ValLexLeader constraints. ◇

Such symmetry breaking may leave one or *more* assignments in each equivalence class. Consider, for example, a Boolean problem in which both variables and values have reflection symmetry. Then the assignments $[0, 1, 1]$ and $[0, 0, 1]$ are symmetric, and both satisfy the appropriate LexLeader and ValLexLeader constraints. Thus, whilst we can post variable symmetry breaking constraints and value symmetry breaking constraints separately, they may not break all symmetry. We need to consider symmetry breaking constraints for the combined variable and value symmetries. We give such constraints in the next section.

6 Variable/Value Symmetry Breaking

In general, a symmetry σ is a bijection on assignments that preserves solutions [12]. We call this a variable/value symmetry to distinguish it from symmetries that act just on the variables or just on the values. Consider, for example, a model for the n-queens problem in which we have one variable for each row. This problem has a rotational symmetry $r90$ that maps $X_i = j$ onto $X_j = n - i + 1$. This is neither a variable nor a value symmetry as it acts on both variables and values together. With variable/value symmetries, we need to be careful that the image of an assignment is itself a proper assignment. We say that a complete assignment $[X_1, \ldots, X_n]$ is *admissible* for σ if the image under σ is also a complete assignment. In particular, the image should assign one value to each variable. Thus $[X_1, \ldots, X_n]$ is admissible iff $|\{k \mid X_i = j, \sigma(i, j) = (k, l)\}| = n$. To return to the 3-queens example, the assignment $[2, 3, 1]$ for $[X_1, X_2, X_3]$ is admissible for $r90$ as its image under $r90$ is $[1, 3, 2]$ which is a complete assignment. However, the assignment $[2, 3, 3]$ is not as its image tries to assign both 1 and 2 to X_3. If $[X_1, \ldots, X_n]$ is admissible for σ we write $\sigma([X_1, \ldots, X_n])$ for its image under σ. More precisely, $\sigma([X_1, \ldots, X_n])$ is the sequence $[Y_1, \ldots, Y_n]$ where for each $X_i = j$ and $\sigma(i, j) = (k, l)$, we have $Y_k = l$.

We now propose a *generic* method to break *all* variable/value symmetry. We simply post the constraint GenLexLeader$(\sigma, [X_1, \ldots, X_n])$ for each variable/value symmetry σ. This holds iff:

$$admissible(\sigma, [X1, \ldots, X_n]) \ \& \ [X_1, \ldots, X_n] \leq_{\text{lex}} \sigma([X_1, \ldots, X_n])$$

Again X_1 to X_n is any fixed ordering on the variables. If σ is a variable symmetry or a value symmetry, all assignments are admissible and we get the same symmetry breaking constraint as before. Consider the 3-queens problem and the $r90$ rotational symmetry. Suppose $X_1 = 2$, $X_2 \in \{1, 3\}$ and $X_3 \in \{1, 2, 3\}$. Then enforcing GAC on GenLexLeader$(r90, [X_1, X_2, X_3])$ prunes $X_3 = 2$ as this does not extend to an admissible assignment, as well as $X_2 = 3$ and $X_3 = 1$ as the image under $r90$ of any admissible assignment with $X_2 = 3$ or $X_3 = 1$ is smaller in the lexicographical order. As before, decomposition into individual symmetry breaking constraints may hinder propagation so it may be worth developing a specialized global constraint that combines several together.

One way to propagate an individual GenLexLeader constraint is to introduce variables Y_1 to Y_n for the image of X_1 to X_n. We post channelling constraints of the form $X_i = j$ iff $Y_k = l$ for each i, j where $\sigma(i, j) = (k, l)$. Finally, we post a lexicographical ordering constraint of the form $[X_1, \ldots, X_n] \leq_{\text{lex}}$

$[Y_1, \ldots, Y_n]$). Enforcing GAC on this decomposition takes just $O(nm)$ time. Unfortunately, this decomposition may hinder propagation.

Theorem 5. *$GAC(\textsc{GenLexLeader}(\sigma, [X_1, \ldots, X_n]))$ is strictly stronger than $GAC([X_1, \ldots, X_n] \leq_{\text{lex}} [Y_1, \ldots, Y_n])$ and $GAC(\forall i, j \ . \ X_i = j \text{ iff } Y_k = l)$ where $\sigma(i, j) = (k, l)$.*

Proof: Clearly it is at least as strong. To show strictness, consider the 3-queens problem and the symmetry $r90$. Suppose $X_1 \in \{1, 3\}$, $X_2 = 2$ and $X_3 \in \{1, 3\}$. Then enforcing GAC on $\textsc{GenLexLeader}(r90, [X_1, X_2, X_3])$ prunes $X_1 = 3$ and $X_3 = 1$ as these are not lex leaders. However, the decomposition is GAC. $\diamond$

To enforce GAC on a GenLexLeader constraint, we can adapt the lex propagator with repeated variables [5]. This give an $O(n^2)$ time propagator. The basic idea is simple but unfortunately we lack space to give full details. The propagator focuses on the first position in the lex constraint where the variables are not ground and equal. We need to test if the variables at this position can be strictly ordered or made equal consistent with admissibility, As in the lex propagator with repeated variables, they can only be made equal if admissibility does not then require the rest of the vector to be ordered the wrong way.

7 Conditional Symmetry Breaking

A conditional symmetry σ is a symmetry that preserves solutions subject to some condition [13]. This condition might be a partial assignment (e.g. that $X_1 = X_2 = 1$) or, more generally, an arbitrary constraint (e.g. that $X_1 \neq X_n$). More precisely, a conditional symmetry σ is a bijection on assignments such that if A is an admissible solution and A satisfies C_σ then $\sigma(A)$ is also a solution. Consider, for example, the all interval series problem (prob007 in CSPLib) in which we wish to find a permutation of the integers from 0 to n, such that the difference between adjacent numbers is itself a permutation from 1 to n. One model of this problem is a sequence of integer variables where X_i represents the number in position i. The problem has a variable reflection symmetry since we can invert any sequence. The problem also has a value reflection symmetry since we can map any value j to $n - j$. Finally, as noted in [13], the problem has a conditional symmetry since we can cycle a solution about a pivot to generate a new solution. Consider $n = 4$ and the solution $[3, 2, 0, 4, 1]$. The difference between first and last numbers in the sequence is 2. However, this is also the difference between the 2nd and 3rd numbers in the sequence. We can therefore rotate round to this point to give the new solution $[0, 4, 1, 3, 2]$. This symmetry is conditional since it depends on the values taken by neighbouring variables.

To break such conditional symmetry, we simply need to post a constraint:

$$C_\sigma(X_1, \ldots, X_n) \text{ implies } \textsc{GenLexLeader}(\sigma, [X_1, \ldots, X_n])$$

Consider again the all interval series problem. We can break the conditional symmetries in this problem by posting the constraints:

$$|X_1 - X_{n+1}| = |X_i - X_{i+1}| \text{ implies } \textsc{LexLeader}(rot_i, [X_1, \ldots, X_{n+1}])$$

Where $1 \leq i \leq n$ and rot_i is the symmetry that rotates a sequence by i elements. We can break other types of conditional symmetries in an identical way.

8 Indistinguishable Values

A common type of value symmetry in which symmetry breaking constraints are effective is when values are interchangeable. For example, in a model of the social golfer problem (prob010 in CSPLib) in which we assign groups to golfers in each week, all values are interchangeable. To break all such symmetry, Law and Lee [11] propose the value precedence constraint:

$$\textsc{Precedence}([v_1, \ldots, v_m], [X_1, \ldots, X_n])$$

This holds iff $\min\{i \mid X_i = v_i \vee i = n+1\} < \min\{i \mid X_i = v_j \vee i = n+2\}$ for all $1 \leq i < j < m$. To propagate this constraint, Law and Lee decompose it into pairwise value precedence constraints, and give a specialized algorithm for enforcing GAC on each pairwise constraint [11]. In [14], we show that this decomposition hinders propagation and give a simple ternary encoding which permits us to achieve GAC in linear time. It is not hard to show that a value precedence constraint combines together the exponential number of ValLexLeader constraints which break the symmetry of indistinguishable values.

9 All Different Problems

Another class of problems on which symmetry breaking constraints are especially effective are those like the all interval series problem in which variables must all take different values. On such problems, Puget has shown that the LexLeader constraints for any (possibly exponentially large) set of variable symmetries simplify down to a linear number of binary inequalities [15]. These can be determined by computing stabilizer chains and coset representatives. For example, on the all interval series problem, the variable reflection symmetry breaking constraint simplifies to $X_1 < X_{n+1}$.

Puget has also shown how to use this method to break value symmetries [8]. We assume we have a surjection problem in which each value is used at least once. (If this is not the case, we introduce "dummy" variables to take the unused values.) We then introduce index variables, Z_i where $Z_j = min\{i \mid X_i = j\}$. These variables are by construction all-different and have a variable symmetry. We can thus break the original value symmetry by adding just a linear number of binary inequalities to break this variable symmetry. For interchangeable values, Puget's method gives the binary symmetry breaking constraints: $X_i = j \rightarrow Z_j \leq i$, $Z_j = i \rightarrow X_i = j$, and $Z_k < Z_{k+1}$ for all $1 \leq i \leq n$, $1 \leq j \leq m$ and $1 \leq k < m$. We observe that these constraints break all value symmetries by ensuring value precedence [11]. However, this decomposition into binary constraints hinders propagation. Consider $X_1 = 1$, $X_2 \in \{1, 2\}$, $X_3 \in \{1, 3\}$, $X_4 \in \{3, 4\}$, $X_5 = 2$, $X_6 = 3$, $X_7 = 4$, $Z_1 = 1$, $Z_2 \in \{2, 5\}$, $Z_3 \in \{3, 4, 6\}$, and $Z_4 \in \{4, 7\}$. Now all the binary implications are arc consistent. However, we can prune $X_2 = 1$ as this violates value precedence.

If we have both variable and value symmetries and an all different problem, we can break variable symmetry with $O(n)$ binary inequalities on the primal model, and break value symmetry with $O(m)$ binary inequalities on the dual. This symmetry breaking on the dual is compatible with symmetry breaking on the primal [8]. However, as in the general case, this may not break all symmetry. Consider an all different problem with 3 variables and 3 values with variable rotation symmetry and value reflection symmetry. Then $[1, 2, 3]$ and $[1, 3, 2]$ are symmetric, but both satisfy the (simplified) variable symmetry breaking constraints that $X_1 < X_2$ and $X_1 < X_3$, and the (simplified) value symmetry breaking constraint (on the dual) that $Y_1 < Y_3$.

10 Set Variable Symmetry Breaking

There are many problems involving symmetry which are naturally modelled and effectively solved using set variables. Set variables can themselves eliminate some symmetry (in particular, the ordering between elements). However, we may still be left with symmetry. For example, another model of the social golfers problem introduces a 2 dimensional matrix of set variables, one for each group and each week. As groups and weeks are indistinguishable, these set variables have row and column symmetry. More generally, a set variable symmetry σ is a bijection on set variables, S_1 to S_n that preserves solutions. To break all such symmetry, we simply post the constraint $\textsc{SetLexLeader}(\sigma, [S_1, \ldots, S_n])$ for each set variable symmetry σ. This ensures:

$$[S_1, \ldots, S_n] \leq_{\text{lex}} [S_{\sigma(1)}, \ldots, S_{\sigma(n)}]$$

Again, S_1 to S_n is any fixed ordering on the set variables. The lexicographical ordering, $\leq_{\text{lex}}$ is the standard lifting of the multiset ordering on set variables to an ordering on tuples of set variables. The multiset ordering on sets is equivalent to the lexicographical ordering on the characteristic function.

We can propagate the $\textsc{SetLexLeader}$ constraint by adapting the propagator for the the lex constraint on finite domain variables described in [5] which copes with repeated variables. Alternatively, we can post a $\textsc{SetLexLeader}$ constraint by means of the following decomposition: $D(S_i, S_{\sigma(i)}, B_i, B_{i+1})$ for $1 \leq i \leq n$ where B_i are Booleans playing the role of α in the lex propagator, $B_1 = 0$, and $D(S_i, S_{\sigma(i)}, B_i, B_{i+1})$ holds iff $B_i = B_{i+1} = 0$ and $S_i = S_{\sigma(i)}$, or $B_i = 0$, $B_{i+1} = 1$ and $S_i <_{\text{mset}} S_{\sigma(i)}$, or $B_i = B_{i+1} = 1$, and $<_{\text{mset}}$ is the multiset ordering on sets.

We still need to provide a propagator for each quaternary constraint, D. This is not trivial as the quaternary constraint is over both set and Boolean variables, and involves notions like whether the sets are multiset ordered. We suppose the set variables are represented by their characteristic function using an m-tuple of Boolean variables. We will use the notation $S_{i,k}$ for the Boolean indicating whether $k \in S_i$. If $i = \sigma(i)$ then $D(S_i, S_{\sigma(i)}, B_i, B_{i+1})$ simplifies to the equality constraint: $B_i = B_{i+1}$. If $i \neq \sigma(i)$ then we give a further decomposition. This decomposition exploits the property that the multiset ordering on sets is identical to the lexicographical ordering on the characteristic function. More precisely, we decompose $D(S_i, S_j, B_i, B_{i+1})$ into $E(S_{i,k}, S_{j,k}, B_i, B_{i+1}, A_{i,k}, A_{i,k+1})$ for $1 \leq k \leq m$, and two implication constraints, $A_{i,m+1} = 0$ implies $B_i = B_{i+1}$ and $A_{i,m+1} = 1$ implies $B_{i+1} = 1$, where $A_{i,k}$ are Booleans (again playing a role similar to α in the lex propagator), and $E(S_{i,k}, S_{j,k}, B_i, B_{i+1}, A_{i,k}, A_{i,k+1})$ holds iff $B_i = A_{i,k} = 0$ implies $S_{i,k} \leq S_{j,k}$ and $S_{i,k} < S_{j,k}$ implies $A_{i,k+1} = 1$.

Our decomposition is a simple logical combinations of Booleans which is readily propagated by most solvers. As this decomposition of $D(S_i, S_j, B_i, B_{i+1})$ is Berge-acyclic, enforcing BC (=GAC) on the decomposition enforces BC on the original quaternary constraint [10]. This takes $O(m)$ time as we can enforce BC on each of the $O(m)$ Boolean constraints in constant time. Enforcing BC on the decomposition of SetLexLeader thus takes $O(nm)$ time.

As with finite domain variables, the decomposition into a set of SetLexLeader constraints hinders propagation.

Theorem 6. *$BC(\bigwedge_{\sigma \in \Sigma}$ SetLexLeader$(\sigma, [S_1, \ldots, S_n]))$ is strictly stronger than $BC($SetLexLeader$(\sigma, [S_1, \ldots, S_n]))$ for all $\sigma \in \Sigma$.*

Proof: Clearly it is at least as strong. We show strictness with just two symmetries. Consider $S_1, S_2 \subseteq \{0, 1\}$, $S_3 = \{0\}$, $S_4 = \{0\}$ and two symmetries defined by the cycles $(1\,2)(3\,4)$ and $(1\,3\,2\,4)$. $\diamond$

Decomposition may also introduce an exponential number of SetLexLeader constraints (e.g. the $n!$ symmetries of n indistinguishable set variables). It may therefore be worth developing specialized global constraints that combine together several SetLexLeader constraints.

11 Set Value Symmetry Breaking

The values used by some set variables might also be symmetric. Consider again the model of the social golfers problem with a 2 dimensional matrix of set variables, each containing the set of golfers playing in particular group in a given week. As the golfers are indistinguishable, any permutation of the values is also a solution. More precisely, a set value symmetry σ is a bijection on values that preserves solutions. That is, if $\{S_i = \{v_{ij} \mid 1 \leq j \leq m_i\} \mid 1 \leq i \leq n\}$ is a solution, then $\{S_i = \{\sigma(v_{ij}) \mid 1 \leq j \leq m_i\} \mid 1 \leq i \leq n\}$ is also. An even more general definition of symmetry would be a bijection on *sets* of values that preserves solutions. However, we focus here on what appears to be the more common case of a bijection on values. To break *all* such set value symmetry, we simply post the constraint SetValLexLeader$(\sigma, [S_1, \ldots, S_n])$ for each value symmetry σ. This constraint ensures:

$$[S_1, \ldots, S_n] \leq_{\text{lex}} [\sigma(S_1), \ldots, \sigma(S_n)]$$

Again, S_1 to S_n is any fixed ordering on the set variables, and $\leq_{\text{lex}}$ is the standard lifting of the multiset ordering on set variables to an ordering on tuples of set variables. To propagate SetValLexLeader we can adapt the lex propagator from [9]. Alternatively, we can use a simple decomposition somewhat similar to that used for ValLexLeader on finite domain variables. We decompose SetValLexLeader$(\sigma, [S_1, \ldots, S_n])$ into $F(S_i, B_i, B_{i+1})$ for $1 \leq i \leq n$ where B_i are Booleans playing the role of α in the lex propagator, $B_1 = 0$, and $F(X_i, B_i, B_{i+1})$ holds iff $B_i = B_{i+1} = 0$ and $S_i = \sigma(S_i)$, or $B_i = 0$, $B_{i+1} = 1$ and $S_i <_{\text{mset}} \sigma(S_i)$, or $B_i = B_{i+1} = 1$. Enforcing BC on this decomposition achieves BC on the SetValLexLeader constraint.

We still need to provide a propagator for each ternary constraint, F. This is again not trivial as it involves a set variable and notions like strict multiset ordering. We can again decompose it into some simple Boolean constraints on the characteristic function representation of the set variable. We identify two important sets of values: $\mathcal{L}$ are those values for which $v < \sigma(v)$, whilst $\mathcal{E}$ are those value for which $v = \sigma(v)$. We then decompose $F(S_i, B_i, B_{i+1})$ into $G(S_{i,k}, B_i, B_{i+1}, A_{i,k}, A_{i,k+1})$ for $1 \leq k \leq m$, and two implication constraints, $A_{i,m+1} = 0$ implies $B_i = B_{i+1}$ and $A_{i,m+1} = 1$ implies $B_{i+1} = 1$, where $A_{i,k}$ are Booleans (again playing a role similar to α in the lex propagator), and $G(S_{i,k}, B_i, B_{i+1}, A_{i,k}, A_{i,k+1})$ holds iff ($B_i = A_{i,k} = 0$ and $k \notin \mathcal{L} \cup \mathcal{E}$) implies $S_{i,k} = 0$ and ($k \in \mathcal{L}$ and $S_{i,k} = 1$) implies $A_{i,k+1} = 1$. Our decomposition is a simple logical combination of Booleans which is readily propagated by most solvers. As this decomposition is Berge-acyclic, enforcing BC (=GAC) on the decomposition enforces BC on the original ternary constraint [10]. This takes $O(m)$ time as we can enforce BC on the $O(m)$ Boolean implication constraints in constant time. Hence, enforcing BC on SetValLexLeader takes $O(nm)$ time. This is optimal. Nevertheless, it may be worth developing a specialized global constraint that combines together several SetValLexLeader constraints as decomposition into individual constraints may hinder propagation, and as there may be an exponential number of SetValLexLeader constraints. For example, the precedence constraint for set variables [11] combines together all the SetValLexLeader constraints for breaking the symmetry of indistinguishable set values.

12 Set Variable and Value Symmetry Breaking

Problems can contain both set variable and set value symmetry. Consider again the model of the social golfers problem with a 2 dimensional matrix of set variables, each containing the set of golfers playing in particular group in a given week. As the weeks, groups and golfers are all interchangeable, the set variables in this model have row and column symmetry, and their values have permutation symmetry. As with finite domain variables, we can combine together symmetry breaking constraints for set variables and set values, provided all the symmetry breaking constraints use the same ordering of set variables.

Theorem 7. *Suppose S_1 to S_n have set variable symmetries Σ and value symmetries Σ'. Then posting* SetLexLeader$(\sigma, [S_1, \ldots, S_n])$ *for all $\sigma \in \Sigma$ and* SetValLexLeader$(\sigma', [S_1, \ldots, S_n])$ *for all $\sigma' \in \Sigma'$ leaves one or more assignments in each equivalence class.*

Proof: Similar to the proof with finite domain variables. The only difference is that we now consider moving down the lexicographical order on tuples of sets. $\diamond$

In fact, complete symmetry breaking in this case is intractable in general (assuming P$\neq$NP). For instance, if we have set variables and values that are indistinguishable, then the problem is isomorphic to breaking all row and column symmetries of an 0/1 matrix model, and this is NP-hard [7].

13 Set Variable/Value Symmetry Breaking

Symmetries can act on set variables and their values simultaneously. Consider a model of the peaceable coexisting armies of queens problem [16] in which we have a set variable for each row of the chessboard containing the positions of the white queens along the row. As in [16], we do not place the black queens and just keep a count on the number of squares not attacked by white queens. This model has a rotational symmetry $r90'$ that maps $i \in S_j$ onto $n-i+1 \in S_i$. This symmetry acts on both set variables and values together. More generally, we will consider a set variable/value symmetry to be a bijection on set variable membership constraints that preserves solutions. That is, if $\{i \in S_j \mid 1 \leq j \leq n\}$ is a solution then $\{k \in S_l \mid i \in S_j,\ \sigma(i,j) = (k,l)\}$ is also. Note that the mapping of a value just depends on the set variable to which it is assigned. An even more general definition would be when the mapping depends on both the value, the set variable, and the other values assigned to the set variable.

Given a complete assignment to the sequence of set variables, $[S_1 \ldots, S_n]$ we write $\sigma([S_1, \ldots, S_n])$ for its image under σ. More precisely, $\sigma([S_1, \ldots, S_n])$ is the sequence $[T_1, \ldots, T_n]$ where for each $\sigma(i,j) = (k,l)$, we have $i \in S_j$ iff $k \in T_l$. To break all set variable/value symmetries, we simply post the constraint $\textsc{SetGenLexLeader}(\sigma, [S_1, \ldots, S_n])$ for each such symmetry σ. This ensures:

$$[S_1, \ldots, S_n] \leq_{\text{lex}} \sigma([S_1, \ldots, S_n])$$

Consider the 5 by 5 peaceable armies of queens problem in which $\{1,4\} \subseteq S_1 \subseteq \{1,4,5\}$, $S_2 = \{2\}$, $S_3 = \{\}$, $\{\} \subseteq S_4 \subseteq \{1\}$ and $\{1\} \subseteq S_5 \subseteq \{1,2,3,4\}$. Enforcing BC on $\textsc{SetGenLexLeader}(r90', [S_1, S_2, S_3, S_4, S_5])$ will reduce the upper bound on S_1 to its lower bound, $\{1,4\}$ to ensure that the placement of white queens is ordered less than its rotation.

As set variable/value symmetry generalizes both set variable and set value symmetry, decomposition into individual symmetry breaking constraints may hinder propagation. We observe, however, that breaking all such symmetry is intractable in general. To propagate an individual $\textsc{SetGenLexLeader}$ constraint, we introduce variables T_1 to T_n for the image of S_1 to S_n. We post channelling constraints of the form $i \in S_j$ iff $k \in T_l$ for each i and j, where $\sigma(i,j) = (k,l)$. Finally we post a lexicographical ordering constraint on the set variables: $[S_1, \ldots, S_n] \leq_{\text{lex}} [T_1, \ldots, T_n]$. We can again propagate this by adapting the propagator for the lex constraint on finite domain variables to set variables [5]. Alternatively, we can use a decomposition similar to that used for the

SetLexLeader constraint. Enforcing BC on this decomposition takes $O(nm)$. Unfortunately, decomposition hinders propagation as it ignores the repeated variables in the lexicographical ordering constraint.

Theorem 8. *$BC(\text{SetGenLexLeader}(\sigma, [S_1, \ldots, S_n]))$ is strictly stronger than $BC(i \in S_j \text{ iff } k \in T_l)$ for each i, j where $\sigma(i, j) = (k, l)$, and $BC([S_1, \ldots, S_n] \leq_{\text{lex}} [T_1, \ldots, T_n])$.*

Proof: Clearly it is at least as strong. To show strictness, consider $\{\} \subseteq S_1, S_3 \subseteq \{1\}$, $S_2 = \{1\}$ and the rotational set variable symmetry, *rot* which maps S_1 to S_3, S_2 to S_1 and S_3 to S_2. Enforcing BC on $\text{SetGenLexLeader}(rot, [S_1, S_2, S_3])$ reduces the upper bound on S_3 to $\{\}$. However, the decomposition is BC. $\diamond$

14 Symmetry Breaking for Other Representations of Sets

We can represent other information about a set variable besides the possible and necessary elements in the set. For example, many solvers include an integer variable to record the cardinality of the set. Such cardinality information can be used when symmetry breaking to permit additional pruning. For instance, suppose we have a set variable, $\{\} \subseteq S_1 \subseteq \{3, 4\}$ with a cardinality of 1 and the permutation set value symmetry, σ_{34} that swaps the values 3 and 4. If we post the symmetry breaking constraint, $\text{SetValLexLeader}(\sigma_{34}, [S_1])$ then a propagator can exploit the cardinality information to prune the upper and lower bounds on S_1 to give S_1 the unique assignment up to set value symmetry of $\{3\}$.

Sadler and Gervet have proposed maintaining upper and lower bounds on a set variable according to the lexicographical ordering [17]. That is, they maintain a set which is lexicographically larger than all possible assignments, as well as one which is lexicographically smaller than all possible assignments. Such lexicographical bounds fit well with the symmetry breaking methods proposed here. For example, suppose we have a pair of set variables, S_1 and S_2 and the permutation set variable symmetry, σ_{12} that swaps S_1 and S_2. Then the symmetry breaking constraint, $\text{SetLexLeader}(\sigma_{12}, [S_1, S_2])$ simplifies to $S_1 \leq_{\text{lex}} S_2$. We can enforce lexicographical bounds consistency on this ordering constraint simply by making the lexicographical upper bound on S_1 equal to the smaller of the two upper bounds, and the lexicographical lower bound on S_2 equal to the larger of the two lower bounds.

Another representation used for a set variable is a binary decision diagram [18]. Whilst such a representation requires exponential space in the worst case, it is often manageable in practice and permits the maximum possible pruning. Whilst set variable symmetry breaking constraints fit well with such a decision diagram representation, set value symmetry breaking constraints do not. For set variable symmetry, we can use the same ordering for values within the decision diagram as within the symmetry breaking constraint. This will lead to a compact representation for the symmetry broken set variables. For set value symmetry, different symmetries can map values far apart in the ordering used in the decision diagram. Adding set value symmetry breaking constraints may therefore give exponentially large decision diagrams.

15 Symmetry Breaking on Other Variable Types

These symmetry breaking methods lift to other types of variables. We consider here multiset variables [19]. Multiset (or bag) variables are useful to model a range of problems. As with set variables, their use can eliminate some but not necessarily all symmetry in the model of a problem. Thus, we may need to break symmetries in models involving multiset variables. For example, one model of the template design problem (prob002 in CSPLib) introduces a multiset variable for each template containing the number of each design. Such multiset variables allow us to ignore the ordering of designs on the template. However, as templates with the same production run length are indistinguishable, certain multiset variables can still be permuted. We thus may have a permutation symmetry, σ over some of the multiset variables. To break all such symmetry, we simply post the constraint MSetLexLeader($\sigma, [M_1, \ldots, M_n]$) for each multiset variable symmetry σ. This constraint ensures:

$$[M_1, \ldots, M_n] \leq_{\text{lex}} [M_{\sigma(1)}, \ldots, M_{\sigma(n)}]$$

Again, M_1 to M_n is any fixed ordering on the multiset variables. The lexicographical ordering, $\leq_{\text{lex}}$ is the lifting of the multiset ordering on multiset variables to an ordering on tuples of set variables. Note that the multiset ordering is equivalent to the lexicographical ordering on the occurrence representation of the multiset. We can propagate the MSetLexLeader constraint by adapting the propagator for the lex constraint on finite domain variables [5] or with an encoding similar to that proposed for the SetLexLeader constraint. The extensions to multiset value and multiset variable/value symmetries are similar. Another promising computation domain for constraint programming is the domain of graphs. Graph variables can be used in domains like bioinformatics to represent combinatorial graph problems [20]. These symmetry breaking methods also lift to deal with symmetries of such graph variables.

16 Related Work

Puget proved that symmetries can always be eliminated by the additional of suitable constraints [1]. Crawford *et al.* presented the first general method for constructing such symmetry breaking constraints [2]. Petrie and Smith applied this lex-leader method to value symmetries [3]. Puget has recently proposed a simple method to propagate such lex-leader constraints when symmetries are the product of variable and value symmetries [4]. He has also proposed a forward checking algorithm for a set of ValLexLeader constraints [4]. To reduce the number of lex-leader constraints used, Aloul *et al.* suggest breaking only those symmetries corresponding to generators of the group [21]. Aloul, Sakallah and

Markov exploit the cyclic structure of generators to reduce the size of lex-leader constraints from quadratic to linear in the number of variables [22]. The full set of lex-leader constraints can often be simplified. For example, in matrix models with row symmetry, the exponential number of lex-leader constraints simplifies to a linear number of lex row ordering constraints [23,24]. For matrix models with both row and column symmetry, it is unlikely that we can break all row and column symmetry using a polynomial number of symmetry breaking constraints as this is NP-hard [7]. However, we can break most row and column symmetry using lex constraints to order rows and columns [23,24]. As a second example, for problems where variables must take all different values, Puget has shown that the lex-leader constraints simplify to just a linear number of binary inequality constraints [15]. Finally, an alternative way to break value symmetry is to convert it into a variable symmetry by channeling into a dual viewpoint and using lex-leader constraints on this dual view [24,25].

17 Conclusions

We have presented some new propagators to break symmetries in constraint satisfaction problems. Our symmetry breaking method works with symmetries acting simultaneously on both variables and values, conditional symmetries, as well as symmetries acting on set and other types of variables. There exist a number of promising areas for future work. Are they efficient ways to combine together these symmetry breaking constraints for particular types of symmetries (just as value precedence combines together an exponential number of value symmetry breaking constraints)? Are there useful subsets of these symmetry breaking constraints when there are too many to post individually? Are there other types of problems and symmetries where these symmetry breaking constraints simplify dramatically (just as Puget has shown for all different problems)? Finally, can these symmetry breaking methods be used to improve symmetry breaking methods like SBDS and SBDD that work during search?

References

1. Puget, J.F.: On the satisfiability of symmetrical constrained satisfaction problems. In ISMIS'93. (1993) 350–361
2. Crawford, J., Luks, G., Ginsberg, M., Roy, A.: Symmetry breaking predicates for search problems. In 5th Int. Conf. on Knowledge Representation and Reasoning, (KR '96). (1996) 148–159
3. Petrie, K.E., Smith, B.M.: Symmetry Breaking in Graceful Graphs. Technical Report APES-56a-2003, APES Research Group (2003)
4. Puget, J.F.: An efficient way of breaking value symmetries. In 21st National Conf. on AI, (2006)
5. Kiziltan, Z.: Symmetry Breaking Ordering Constraints. PhD thesis, Dept. of Information Science, Uppsala University (2004)
6. Carlsson, M., Beldiceanu, N.: Arc-consistency for a chain of lexicographic ordering constraints. Technical report T2002-18, Swedish Institute of Computer Science (2002)

7. Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of global constraints. In Proc. of the 19th National Conf. on AI, (2004)
8. Puget, J.F.: Breaking all value symmetries in surjection problems. In 11th Int. Conf. on Principles and Practice of Constraint Programming (CP2005), (2005)
9. Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Global constraints for lexicographic orderings. In 8th Int. Conf. on Principles and Practices of Constraint Programming (CP-2002), (2002)
10. Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence **38** (1989) 353–366
11. Law, Y., Lee, J.: Global constraints for integer and set value precedence. In 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004) 362–376
12. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K., Smith, B.: Symmetry definitions for constraint satisfaction problems. In 11th Int. Conf. on Principles and Practice of Constraint Programming (CP2005), (2005) 17–31
13. Gent, I., Kelsey, T., Linton, S., McDonald, I., Miguel, I., Smith, B.: Conditional symmetry breaking. In 11th Int. Conf. on Principles and Practice of Constraint Programming (CP2005), (2005)
14. Walsh, T.: Symmetry breaking using value precedence. In 17th ECAI, IOS Press (2006)
15. Puget, J.F.: Breaking symmetries in all different problems. In 19th IJCAI, (2005) 272–277
16. Smith, B., Petrie, K., Gent, I.: Models and symmetry breaking for peacable armies of queens. In Proc. of the 1st Int. Conf. on Integration of AI and OR Techniques in Constraint Programming (CP-AI-OR), (2004) 271–286.
17. Sadler, A., Gervet, C.: Hybrid set domains to strengthen constraint propagation and reduce symmetries. In 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004)
18. Hawkins, P., Lagoon, V., Stuckey, P.: Solving set constraint satisfaction problems using ROBDDs. Journal of Artificial Intelligence Research **24** (2005) 109–156
19. Walsh, T.: Consistency and propagation with multiset constraints: A formal viewpoint. In 9th Int. Conf. on Principles and Practices of Constraint Programming (CP-2003), (2003)
20. Dooms, G., Deville, Y., Dupont, P.: CP(Graph): Introducing a graph computation domain in constraint programming. In 11th Int. Conf. on Principles and Practice of Constraint Programming (CP2005), (2005)
21. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Solving difficult SAT instances in the presence of symmetries. In Proc. of the Design Automation Conf.. (2002) 731–736
22. Aloul, F., Sakallah, K., Markov, I.: Efficient symmetry breaking for Boolean satisfiability. In 18th IJCAI (2003) 271–276
23. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. In Proc. of LICS workshop on Theory and Applications of Satisfiability Testing (SAT 2001). (2001)
24. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetry in matrix models. In 8th Int. Conf. on Principles and Practices of Constraint Programming (CP-2002), (2002)
25. Law, Y., Lee, J.: Symmetry Breaking Constraints for Value Symmetries in Constraint Satisfaction. Constraints (2006) to appear.

Inferring Variable Conflicts for Local Search[⋆]

Magnus Ågren, Pierre Flener, and Justin Pearson

Department of Information Technology, Uppsala University, Sweden
{agren, pierref, justin}@it.uu.se

Abstract. For efficiency reasons, neighbourhoods in local search are often shrunk by only considering moves modifying variables that actually contribute to the overall penalty. These are known as conflicting variables. We propose a new definition for measuring the conflict of a variable in a model and apply it to the set variables of models expressed in existential second-order logic extended with counting ($\exists$SOL$^+$). Such a variable conflict can be automatically and incrementally evaluated. Furthermore, this measure is lower-bounded by an intuitive conflict measure, and upper-bounded by the penalty of the model. We also demonstrate the usefulness of the approach by replacing a built-in global constraint by an $\exists$SOL$^+$ version thereof, while still obtaining competitive results.

1 Introduction

In local search, it is often important to limit the size of the neighbourhood by only considering moves modifying conflicting variables, i.e., variables that actually contribute to the overall penalty. See [4,6,8], for example.

We address the inference of variable conflicts from a formulation of a constraint. After giving necessary background information in Section 2, we propose in Section 3 a new definition for measuring the conflict of a variable and apply it to the *set* variables of models expressed in existential second-order logic extended with counting ($\exists$SOL$^+$) [5]. Such a variable conflict can be automatically and incrementally evaluated. The calculated value is lower-bounded by an intuitive target value, namely the maximum penalty decrease of the model that may be achieved by only changing the given variable, and upper-bounded by the penalty of the model. We demonstrate the usefulness of the approach in Section 4 by replacing a built-in constraint by an $\exists$SOL$^+$ version, while still obtaining competitive results.

2 Preliminaries

As usual, a *constraint satisfaction problem* (*CSP*) is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a finite set of variables, $\mathcal{D}$ is a finite set of domains, each $D_x \in \mathcal{D}$ containing the set of possible values for $x \in \mathcal{X}$, and $\mathcal{C}$ is a finite set of constraints, each being defined on a subset of $\mathcal{X}$ and specifying their valid combinations of values.

⋆ This research was partially funded by EuroControl project C/1.246/HQ/JC/04.

A variable $S \in \mathcal{X}$ is a *set variable* if its corresponding domain D_S is $2^{\mathcal{U}}$, where $\mathcal{U}$ is a common finite set of values of some type, called the *universe*.

Local search iteratively makes a small change to a current assignment of values to *all* variables (configuration), upon examining the merits of many such changes, until a solution is found or allocated resources have been exhausted. The configurations examined constitute the neighbourhood of the current configuration, crucial guidance being provided by penalties and variable conflicts.

Definition 1. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP. A* configuration *for P (or $\mathcal{X}$) is a total function $k : \mathcal{X} \to \bigcup_{D \in \mathcal{D}} D$. We use $\mathcal{K}$ to denote the* set of all configurations *for a given CSP or set of variables, depending on the context. A* neighbourhood function *for P is a function $n : \mathcal{K} \to 2^{\mathcal{K}}$. The* neighbourhood *of P with respect to (w.r.t.) a configuration $k \in \mathcal{K}$ and n is the set $n(k)$. The* variable neighbourhood *for $x \in \mathcal{X}$ w.r.t. k is the subset of $\mathcal{K}$ reachable from k by changing $k(x)$ only: $n_x(k) = \{\ell \in \mathcal{K} \mid \forall y \in \mathcal{X} : y \neq x \to k(y) = \ell(y)\}$. A* penalty function *of a constraint $c \in \mathcal{C}$ is a function $penalty(c) : \mathcal{K} \to \mathbb{N}$ such that (s.t.) $penalty(c)(k) = 0$ if and only if (iff) c is satisfied w.r.t. k. The* penalty *of c w.r.t. k is $penalty(c)(k)$. A* conflict function *of c is a function $conflict(c) : \mathcal{X} \times \mathcal{K} \to \mathbb{N}$ s.t. if $conflict(c)(x, k) = 0$ then $\forall \ell \in n_x(k) : penalty(c)(k) \leq penalty(c)(\ell)$. The* conflict *of x w.r.t. c and k is $conflict(c)(x, k)$.*

Example 1. Let $P = \langle \{S, T\}, \{D_S, D_T\}, \{S \subset T\} \rangle$ where $D_S = D_T = 2^{\mathcal{U}}$ and $\mathcal{U} = \{a, b, c\}$. A configuration for P is given by $k(S) = \{a, b\}$ and $k(T) = \emptyset$, or equivalently by $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$. The neighbourhood of P w.r.t. k and the neighbourhood function for P that moves an element from S to T is the set $\{k_a = \{S \mapsto \{b\}, T \mapsto \{a\}\}, k_b = \{S \mapsto \{a\}, T \mapsto \{b\}\}$. The variable neighbourhood for S w.r.t. k is the set $n_S(k) = \{k, k_1 = \{S \mapsto \emptyset, T \mapsto \emptyset\}, k_2 = \{S \mapsto \{a\}, T \mapsto \emptyset\}, k_3 = \{S \mapsto \{b\}, T \mapsto \emptyset\}, k_4 = \{S \mapsto \{c\}, T \mapsto \emptyset\}, k_5 = \{S \mapsto \{a, c\}, T \mapsto \emptyset\}, k_6 = \{S \mapsto \{b, c\}, T \mapsto \emptyset\}, k_7 = \{S \mapsto \{a, b, c\}, T \mapsto \emptyset\}\}$. Let the penalty and conflict functions of $S \subset T$ be defined by:

$$penalty(S \subset T)(k) = |k(S) \setminus k(T)| + \begin{cases} 1, & \text{if } k(T) \subseteq k(S) \\ 0, & \text{otherwise} \end{cases}$$

$$conflict(S \subset T)(Q, k) = |k(S) \setminus k(T)| + \begin{cases} 1, & \text{if } Q = T \text{ and } k(T) \subseteq k(S) \\ 1, & \text{if } Q = S \text{ and } k(S) \cap k(T) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

We have that $penalty(S \subset T)(k) = 3$. Indeed, we may satisfy P w.r.t. k by, e.g., adding the three values a, b, and c to T. We also have that $conflict(S \subset T)(S, k) = 2$ and $conflict(S \subset T)(T, k) = 3$. Indeed, by removing the values a and b from S, we may decrease the penalty of P by two. Similarly, by adding the values a, b, and c to T, we may decrease the penalty of P by three.

We use existential second-order logic extended with counting ($\exists$SOL$^+$) for modelling set constraints [1]. In the BNF below, the non-terminal symbol $\langle S \rangle$ denotes an identifier for a bound set variable S such that $S \subseteq \mathcal{U}$, while $\langle x \rangle$ and $\langle y \rangle$ denote identifiers for bound variables x and y such that $x, y \in \mathcal{U}$, and $\langle a \rangle$ denotes a natural number constant:

$$
\begin{array}{lll}
\langle \mathit{Constraint} \rangle & ::= & (\underline{\exists}\ \langle S \rangle)^{+}\ \langle \mathit{Formula} \rangle \\
\langle \mathit{Formula} \rangle & ::= & \underline{(}\langle \mathit{Formula} \rangle\underline{)} \mid (\underline{\forall} \mid \underline{\exists})\langle x \rangle\ \langle \mathit{Formula} \rangle \\
 & \mid & \langle \mathit{Formula} \rangle\ (\underline{\wedge} \mid \underline{\vee})\ \langle \mathit{Formula} \rangle \mid \langle \mathit{Literal} \rangle \\
\langle \mathit{Literal} \rangle & ::= & \langle x \rangle\ (\underline{\in} \mid \underline{\notin})\ \langle S \rangle \\
 & \mid & \langle x \rangle\ (\underline{<} \mid \underline{\leq} \mid \underline{=} \mid \underline{\neq} \mid \underline{\geq} \mid \underline{>})\ \langle y \rangle \\
 & \mid & \underline{|}\langle S \rangle\underline{|}\ (\underline{<} \mid \underline{\leq} \mid \underline{=} \mid \underline{\neq} \mid \underline{\geq} \mid \underline{>})\ \langle a \rangle
\end{array}
$$

As a running example, consider the constraint $S \subset T$ of Ex. 1. This may be expressed in $\exists$SOL$^+$ by $\Omega = \exists S \exists T((\forall x(x \notin S \vee x \in T)) \wedge (\exists x(x \in T \wedge x \notin S)))$.

We proposed a penalty function for $\exists$SOL$^+$ formulas in [1], which was inspired by [9]. For example, the penalty of a literal $x \in S$ w.r.t. a configuration k is 0 if $k(x) \in k(S)$ and 1, otherwise. The penalty of a conjunction (disjunction) is the sum (minimum) of the penalties of its conjuncts (disjuncts). The penalty of a universal (existential) quantification is the sum (minimum) of the penalties of the quantified formula where the occurrences of the bound variable are replaced by each value in the universe.

Example 2. Recall $k = \{S \mapsto \{a,b\}, T \mapsto \emptyset\}$ of Ex. 1. Then $\mathit{penalty}(\Omega)(k) = 3$.

3 Variable Conflicts of an $\exists$SOL$^+$ Formula

The notion of abstract conflict measures the maximum possible penalty decrease obtainable by only changing the value of the given variable. It is uniquely determined by the chosen penalty function:

Definition 2. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP and let $c \in \mathcal{C}$. The* abstract conflict function *of c w.r.t. $\mathit{penalty}(c)$ is the function $\mathit{abstractConflict}(c) : \mathcal{X} \times \mathcal{K} \to \mathbb{N}$ s.t. $\mathit{abstractConflict}(c)(x,k) = \max\{\mathit{penalty}(c)(k) - \mathit{penalty}(c)(\ell) \mid \ell \in n_x(k)\}$. The* abstract conflict *of $x \in \mathcal{X}$ w.r.t. c and $k \in \mathcal{K}$ is $\mathit{abstractConflict}(c)(x,k)$.*

Example 3. The function $\mathit{conflict}(S \subset T)$ of Ex. 1 gives abstract conflicts.

Similarly to our penalty function in [1], it is important to stress that the calculation of the variable conflict defined next is automatable and feasible incrementally [3], as it is based only on the syntax of the formula and the semantics of the quantifiers, connectives, and relational operators of $\exists$SOL$^+$, but not on the intended semantics of the formula.

Definition 3. *Let $\mathcal{F} \in \exists$SOL$^+$, let $S \in \mathit{vars}(\mathcal{F})$, and let k be a configuration for $\mathit{vars}(\mathcal{F})$. The* conflict *of S w.r.t. $\mathcal{F}$ and k is defined by:*

(a) $\mathit{conflict}(\exists S_1 \cdots \exists S_n \phi)(S,k) = \mathit{conflict}(\phi)(S,k)$

(b) $\mathit{conflict}(\forall x \phi)(S,k) = \sum_{u \in \mathcal{U}} \mathit{conflict}(\phi)(S, k \cup \{x \mapsto u\})$

(c) $\mathit{conflict}(\exists x \phi)(S,k) =$
$\max\{0\} \cup \{\mathit{penalty}(\exists x \phi)(k) - (\mathit{penalty}(\phi)(k \cup \{x \mapsto u\}) - \mathit{conflict}(\phi)(S, k \cup \{x \mapsto u\})) \mid u \in \mathcal{U}\}$

(d) $\mathit{conflict}(\phi \wedge \psi)(S,k) = \sum\{\mathit{conflict}(\gamma)(S,k) \mid \gamma \in \{\phi, \psi\} \wedge S \in \mathit{vars}(\gamma)\}$

(e) $\mathit{conflict}(\phi \vee \psi)(S,k) = \max\{0\} \cup \{\mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\gamma)(k) - \mathit{conflict}(\gamma)(S,k)) \mid \gamma \in \{\phi, \psi\} \wedge S \in \mathit{vars}(\gamma)\}$

(f) $\mathit{conflict}(|S| \leq c)(S,k) = \mathit{penalty}(|S| \leq c)(k)$

(g) $\mathit{conflict}(x \in S)(S,k) = \mathit{penalty}(x \in S)(k)$

We only show cases for subformulas of the form $|S| \lozenge c$ and $x \vartriangle S$ where $\lozenge \in \{\leq\}$ and $\vartriangle \in \{\in\}$. The other cases are defined similarly.

Example 4. Recall once again $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ of Ex. 1. According to Def. 3, we have that $\mathit{conflict}(\Omega)(S, k) = 2$ and $\mathit{conflict}(\Omega)(T, k) = 3$, i.e., the same values as obtained by the handcrafted $\mathit{conflict}(S \subset T)$ of Ex. 1.

The novelty of Def. 3 compared to the one in [8] lies in rules (c) and (e) for disjunctive formulas, due to the different abstract conflict that we target (see [3] for more details). The following example clarifies these rules in terms of (e).

Example 5. Consider $\mathcal{F} = (|S| = 5 \vee (|T| = 3 \wedge |S| = 6))$ and let k_1 be a configuration s.t. $|k_1(S)| = 6$ and $|k_1(T)| = 4$. Then $\mathit{penalty}(\mathcal{F})(k_1) = 1$ and we have $\mathit{conflict}(|S| = 5)(S, k_1) = 1$ and $\mathit{conflict}(|T| = 3 \ \wedge \ |S| = 6)(S, k_1) = 0$. Rule ($e$) applies for calculating $\mathit{conflict}(\mathcal{F})(S, k_1)$, which, for each disjunct, gives the *maximum possible penalty decrease* one may obtain by changing $k_1(S)$. This is 1 for the first disjunct since we may decrease $\mathit{penalty}(\mathcal{F})(k_1)$ by 1 by changing $k_1(S)$ as witnessed by $\mathit{penalty}(\mathcal{F})(k_1) - (\mathit{penalty}(|S| = 5)(k_1) - \mathit{conflict}(|S| = 5)(S, k_1)) = 1 - (1 - 1) = 1$. It is 0 for the second disjunct since we cannot decrease $\mathit{penalty}(\mathcal{F})(k_1)$ by changing $k_1(S)$ as witnessed by $\mathit{penalty}(\mathcal{F})(k_1) - (\mathit{penalty}(|T| = 3 \wedge |S| = 6)(k_1) - \mathit{conflict}(|T| = 3 \wedge |S| = 6)(S, k_1) = 1 - (1 - 0) = 0$. The maximum value of these is 1 and hence $\mathit{conflict}(\mathcal{F})(S, k_1) = 1$.

Consider now k_2 s.t. $|k_2(S)| = 4$ and $|k_2(T)| = 4$. Then $\mathit{penalty}(\mathcal{F})(k_2) = 1$ and $\mathit{conflict}(|T| = 3 \ \wedge \ |S| = 6)(T, k_2) = 1$. The maximum possible penalty decrease one may obtain by changing $k_2(T)$ in the only disjunct for T is -1 as witnessed by $\mathit{penalty}(\mathcal{F})(k_2) - (\mathit{penalty}(|T| = 3 \wedge |S| = 6)(k_2) - \mathit{conflict}(|T| = 3 \wedge |S| = 6)(T, k_2) = 1 - (3 - 1) = -1$. But we may not have a negative conflict, hence the union with $\{0\}$ in (e). Indeed, we cannot decrease $\mathit{penalty}(\mathcal{F})(k)$ by changing $k_2(T)$ since even if we satisfy $|k_2(T) = 3|$, the conjunct $|k_2(S) = 6|$ implies a penalty larger than 1 which is the minimum penalty of the two disjuncts.

We now state some properties of variable conflicts compared to the abstract conflict of Def. 2 and the formula penalty [1]. The proofs can be found in [3].

Proposition 1. *Let $\mathcal{F} \in \exists\mathrm{SOL}^+$, let k be a configuration for $\mathit{vars}(\mathcal{F})$, and let $S \in \mathit{vars}(\mathcal{F})$. Then $\mathit{abstractConflict}(\mathcal{F})(S, k) \leq \mathit{conflict}(\mathcal{F})(S, k) \leq \mathit{penalty}(\mathcal{F})(k)$.*

Corollary 1. *The function induced by Def. 3 is a conflict function w.r.t. Def. 1.*

4 Practical Results and Conclusion

The *progressive party problem* [7] is about timetabling a party at a yacht club, where the crews of certain boats (the guest boats) party at other boats (the host boats) over a number of periods. The crew of a guest boat must party at some host boat in each period. The spare capacity of a host boat is never to be exceeded. The crew of a guest boat may visit a particular host boat at most once. The crews of two distinct guest boats may meet at most once.

We use the same set-based model and local search algorithm as we did in [2]. The model includes *AllDisjoint*$(\mathcal{X})(k)$ constraints that hold iff no two distinct set variables in $\mathcal{X} = \{S_1, \ldots, S_n\}$ overlap. Assuming that this global constraint is not built-in, we may use the following $\exists$SOL$^+$ version instead:

$$\begin{aligned}\exists S_1 \cdots \exists S_n \forall x \; (& (x \notin S_1 \vee (x \notin S_2 \wedge \cdots \wedge x \notin S_n)) \wedge \\ & (x \notin S_2 \vee (x \notin S_3 \wedge \cdots \wedge x \notin S_n)) \wedge \cdots \wedge (x \notin S_{n-1} \vee x \notin S_n))\end{aligned}$$

We have run the same classical instances as we did in [2], on a 2.4GHz/512MB Linux machine. The following table shows the results for the $\exists$SOL$^+$ and built-in versions of the *AllDisjoint* constraint (mean run time in seconds of successful runs out of 100 and the number of unsuccessful runs, if any, in parentheses).

	$\exists$SOL$^+$ *AllDisjoint*					Built-in *AllDisjoint*				
H/periods (fails)	6	7	8	9	10	6	7	8	9	10
1-12,16			1.3	3.5	42.0			1.2	2.3	21.0
1-13			16.5	239.3				7.0	90.5	
1,3-13,19			18.9	273.2 (3)				7.2	128.4 (4)	
3-13,25,26			36.5	405.5 (16)				13.9	170.0 (17)	
1-11,19,21	19.8	186.7				10.3	83.0 (1)			
1-9,16-19	32.2	320.0 (12)				18.2	160.6 (22)			

The run times for the $\exists$SOL$^+$ version are only 2 to 3 times higher, though it must be noted that *efforts such as designing penalty and conflict functions as well as incremental maintenance algorithms for AllDisjoint were not necessary.* Note also that the robustness of the local search algorithm does not degrade for the $\exists$SOL$^+$ version, as witnessed by the number of solved instances.

To conclude, we proposed a new definition for inferring the conflict of a variable in a model and proved that any inferred variable conflict is lower-bounded by the targeted value, and upper-bounded by the inferred penalty. *The search is indeed directed towards interesting neighbourhoods, as a built-in constraint can be replaced without too high losses in run-time, nor any losses in robustness.*

References

1. M. Ågren, P. Flener, and J. Pearson. Incremental algorithms for local search from existential second-order logic. *Proceedings of CP'05*. Springer-Verlag, 2005.
2. M. Ågren, P. Flener, and J. Pearson. Set variables and local search. *Proceedings of CP-AI-OR'05*. Springer-Verlag, 2005.
3. M. Ågren, P. Flener, and J. Pearson. Inferring variable conflicts for local search. Tech. Rep. 2006-005, Dept. of Information Technology, Uppsala University, 2006.
4. P. Galinier and J.-K. Hao. A general approach for constraint solving by local search. *Proceedings of CP-AI-OR'00*, 2000.
5. N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
6. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. *Proceedings of OOPSLA'02*, 2002.
7. B. M. Smith *et al.* The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.
8. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, 2005.
9. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-based combinators for local search. *Proceedings of CP'04*. Springer-Verlag, 2004.

Reasoning by Dominance in Not-Equals Binary Constraint Networks

Belaïd Benhamou and Mohamed Réda Saïdi

Laboratoire des Sciences de l'Information et des Systmes (LSIS)
Centre de Mathmatiques et d'Informatique
39, rue Joliot Curie - 13453 Marseille cedex 13, France
Belaid.Benhamou@cmi.univ-mrs.fr, saidi@cmi.univ-mrs.fr

Abstract. In this paper, we extend the principle of symmetry to dominance in *Not-Equals Constraint Networks* and show how dominated values are detected and eliminated efficiently at each node of the search tree.

1 Introduction

As far as we know the principle of symmetry is first introduced by [1] to improve resolution in propositional logic. Symmetry for boolean constraints is studied in [2] where the authors showed that its exploitation is a real improvement for several automated deduction algorithms' efficiency. Symmetry in CSPs is studied in [3,4]. Since that, many research works on symmetry appeared.

Recently a method which breaks symmetries between the variables of an Alldiff constraint is studied in [5], a method which eliminates all value symmetries in surjection problems is given in [6], and a work gathering the different symmetry definitions is done in [7].

We investigate in this article the principle of dominance in *Not-Equals binary Constraint Networks (notation NECSPs).* A CSP is an object $P = (X, D, C)$ where: X is a finite set of variables; D is the set of finite discrete domains associated to the CSP variables; C is a finite set of constraints each involving some subsets of the CSP variables. A constraint is binary when it involves two variables. A binary constraint is called a Not-Equal constraint if it forces the two variables X_i and X_j to take different values (it is denoted by $X_i \neq X_j$). A binary Not-Equal CSP (NECSP) is a CSP whose all constraints are binary Not-Equal constraints. Dominance is a weak symmetry principle which extend the Full substitutability notion [8]. Dominance is first introduced in [9] for general CSPs, but it is shown that its detection is harder than symmetry. Here, we show how dominance is adapted, detected, and exploited efficiently in NECSPs. Of course, the NECSPs is a limited framework, but in theory, this restriction remains NP-complete. Indeed, Graph coloring fits in the NECSPs framework and is NP-complete, thus solving Not-Equals CSPs is in general a NP-complete problem. Besides, in practice, this framework is quite expressive, it covers a broad range of problems in artificial intelligence, such as Time-tabling and Scheduling, Register Allocation in compilation, and Cartography.

For space reason, the work is not developped in depth here. A more complete version of this paper is given in [10].

2 Dominance in NECSPs

Benhamou in [9] defined a weak symmetry called *Dominance* which extends the Full substitutability notion [8]. Here we adapt the principle of dominance to NECSPs and give some sufficient conditions which leads to a linear algorithm for dominance detection.

2.1 The Principle of Dominance

Definition 1. *[Semantic Dominance] A value a_i dominates another value b_i for a CSP variable $v_i \in V$ (notation $a_i \succeq b_i$) iff [There exist a solution of the CSP which assigns the value b_i to the variable $v_i \Rightarrow$ there exist a solution of the CSP which assigns the value a_i to v_i].*

The value b_i participates in a solution if the value a_i does; otherwise it does not. The value b_i can be removed from D_i without affecting the CSP consistency.

Proposition 1. *If $a_i \succeq b_i$ and a_i doesn't participate in any solution of $\mathcal{P}$, then b_i doesn't participate in any solution of $\mathcal{P}$.*

2.2 A Sufficient Condition for Dominance

Now, we give the sufficient conditions for dominance which represent the main key of this work.

Theorem 1. *Let a_i and b_i be two values of a domain D_i corresponding to a variable X_i of a Not-Equals CSP $\mathcal{P}$, I a partial instantiation of $\mathcal{P}$, and Y the set of un-instantiated variables of $\mathcal{P}$. If the two following conditions:*

1. *$a_i \in D_j \Rightarrow b_i \in D_j$, for all $X_j \in Y$ such that X_j shares a constraint with X_i.*
2. *$a_i \in D_j \Leftrightarrow b_i \in D_j$ for each variable X_j of Y which does not share a constraint with X_i*

hold, then a_i dominates b_i in $\mathcal{P}$.

2.3 The Weakened Dominance Sufficient Conditions

Before introducing the weakened sufficient conditions, we define the notion of assignment trees and failure trees corresponding to the enumerative search method used to prove the consistency of the considered CSP.

Definition 2. *We call an assignment tree of a CSP $\mathcal{P}$ corresponding to a given search method and a fixed variable ordering, a tree which gathers the history of all the variable assignments made during its consistency proof, where the nodes represent the variables of the CSP and where the edges out coming from a node*

X_i are labeled by the different values used to instantiate the corresponding CSP variable X_i.

Definition 3. *Let T be an assignment tree corresponding to a consistency proof of a CSP $\mathcal{P}$, $I = (a_1, a_2, ..., a_i)$ an inconsistent partial instantiation of the variables X_1,X_2,...,X_i corresponding to the path $\{X_1, X_2, ..., X_i\}$ in T. We call a failure tree of the instantiation I, the sub-tree of T noted by $T_{I=(a_1,a_2,...,a_i)}$ such that:*

1. *The root of the tree T and the root of the sub-tree $T_{I=(a_1,a_2,...,a_i)}$ are joined by the path corresponding to the instantiation I;*
2. *All the CSP variables corresponding to the leaf nodes of $T_{I=(a_1,a_2,...,a_i)}$ have empty domains.*

We can now give the weakened sufficient conditions of dominance. That is, the conditions of theorem 1 are restricted to only the variables involved in the failure tree among the un-instantiated ones.

Theorem 2. *Let $\mathcal{P}(X, D, C)$ be a CSP, $a_i \in D_i$ and $b_i \in D_i$ two values of the domain D_i of the current CSP variable X_i under instantiation, $I_0 = (a_1, ..., a_{i-1})$ a partial instantiation of the $i-1$ variables instantiated before X_i such that the extension $I = I_0 \bigcup \{a_i\} = (a_1, ..., a_{i-1}, a_i)$ is inconsistent, $T_{I=(a_1,...,a_{i-1},a_i)}$ is the failure tree of I and $Var(T_{I=(a_1,...,a_{i-1},a_i)})$ the set of variables corresponding to the nodes of $T_{I=(a_1,...,a_{i-1},a_i)}$. If the two following conditions:*

1. *$b_i \in D_j \Rightarrow a_i \in D_j$, for all $X_j \in Var(T_{I=(a_1,...,a_{i-1},a_i)})$ such that X_j shares a constraint with X_i.*
2. *$a_i \in D_j \Leftrightarrow b_j \in D_i$ for each other variable X_j of $Var(T_{I=(a_1,...,a_{i-1},a_i)})$ which do not share a constraint with X_i*

hold, then the extension $J = I_0 \bigcup \{b_i\} = (a_1, ..., a_{i-1}, b_i)$ is inconsistent.

2.4 Dominance Detection and Exploitation

Dominance detection is based on the conditions of theorem 2. The algorithm sketched in Figure 1 computes the values dominated by a value a_i of a given domain D_i *w.r.t* the conditions of theorem 2.

The algorithm computes the class $cl(d_i)$ of dominated values by a_i with a complexity $\mathcal{O}(nd)$ in the worst case. It has a linear complexity *w.r.t* the NECSP size.

Theorem 2 allows to prune k-1 branches in the search tree if there are k dominated values by a dominant value which is shown to not participating in any solution.

This property can be exploited in all enumerative resolution methods. Here we use it in a *Simplified Forward Checking* method (denoted by SFC) adapted to NECSPs where the filtering consists only in removing the value d_i from the domains of the future variables having a constraint with the current variable v_i under instantiation with d_i.

```
procedure $weak\_dominance(a_i \in D_i, Var(T_{I=(a_1,\ldots,a_i)}))$, var cl($a_i$):class);
input: a value $a_i \in D_i$, a set of variables $Var(T_{I=(a_1,\ldots,a_i)})$
Output: the class $cl(a_i)$ of the dominated values by $a_i$.
begin
    cl($a_i$):={$a_i$}
    for each $d_i \in D_i$-{$a_i$} do
        for each domain $D_k$ of variables of $Var(T_{I=(a_1,\ldots,a_i)})$ do
            if ($c_{ik} \in C$ and ($a_i \in D_k \Rightarrow d_i \in D_k$)) or
            ($c_{ik} \notin C$ and ($a_i \in D_k \Leftrightarrow d_i \in D_k$)) then cl($a_i$):=cl($a_i$)∪{$d_i$}
end
```

Fig. 1. The algorithm of dominance search in NECSPs

3 Experiments

We tested and compared the SFC augmented by the symmetry property defined in [11] (SFC-sym), the SFC augmented by the advantage of the dominance property of theorem 2 (SFC-weak-dom), and an improved version [12] of the well known method DSATUR. The source code is written in C and compiled on a P4 2.8 GHz - RAM 1 Go.

Table 1. Dimacs graph coloring benchmarks

Pb	k	DSATUR		SFC-SYM		SFC-dom-weak		Pb	k	DSATUR		SFC-SYM		SFC-dom-weak	
instances		*N*	*T*	*N*	*T*	*N*	*T*	*instances*		*N*	*T*	*N*	*T*	*N*	*T*
queen8_8	9	1581661	4.3	1368441	6.1	1353680	6.1	myciel5	6	378310	0.6	72966	0.2	21278	0.0
queen8_12	12	162	0.0	460	0.0	460	0.0	myciel6	7	-	-	83157279	556.0	29754513	190.2
le450_5a	5	-	-	1408	0.1	1395	0.1	le450_25a	25	425	0.1	450	0.1	450	0.1
le450_5b	5	-	-	19884	0.6	19763	0.5	le450_25b	25	425	0.0	450	0.1	450	0.1
qg.order30	30	1680	0.2	1169	0.2	1162	0.2	school.1	14	371	0.2	568	0.4	555	0.4
qg.order40	40	-	-	12089785	302.0	10814593	266.6	school_nsh	14	338	0.2	352	0.4	352	0.4
1-FullIns_3	4	37	0.0	51	0.0	50	0.0	mug88_25	4	-	-	22643	0.0	1631	0.0
2-FullIns_3	5	156663424	193.6	678	0.0	359	0.0	mug100_25	4	-	-	99917	0.2	515	0.0
ash608	4	10242	0.5	1742	0.5	1707	0.5	R125.5	36	1357573	9.1	55952	0.4	1051	0.0
ash958	4	-	-	10252	2.2	7167	1.6	R500.1c	84	-	-	-	-	28044984	3096.0

Table 1 shows the results of the methods on some graph coloring benchmarks of Dimacs[1]. It gives the number of nodes of the search tree and the CPU time for each method. We seek for each of them the chromatic number[2] k.

We can remark that only SFC-weak-dom is able to solve the known hard problem "DSJR500.1c" which, as far as, we know it has never been solved by an exact method. We can see that SFC-dom-weak outperforms both DSATUR, and SFC-sym is better than DSATUR on these problems. Theses methods are also tested and compared on random graph coloring problems which are not reported here, and the results confirme that SFC-weak-dom has the best performance in the average.

4 Conclusion

In this work we extended the symmetry principle to dominance and weakened the symmetry/dominance sufficient conditions in Not-Equals constraint networks

[1] http://dimacs.rutgers.edu/Challenges/

[2] The minimal number of colors needed to color the vertices of the corresponding graph.

when an inconsistent partial instantiation is generated. We implemented a more efficient dominance search algorithm which detects both symmetry and which captures the dominance. We exploited the new dominance property in a backtracking algorithm which we use to solve NECSPs. Experiments are carried, and the obtained results show that reasoning by dominance is profitable for solving NECSPs.

References

1. Krishnamurty, B.: Short proofs for tricky formulas. Acta informatica (22) (1985) 253–275
2. Benhamou, B., Sais, L.: Theoretical study of symmetries in propositional calculus and application. CADE-11, Saratoga Springs,NY, USA (1992)
3. Puget, J.F.: On the satisfiability of symmetrical constrained satisfaction problems. In: ISMIS. (1993)
4. Benhamou, B.: Study of symmetry in constraint satisfaction problems. In PPCP'94 (1994)
5. Puget, J.F.: Breaking symmetries in all different problems. In, proceedings of IJCAI (2005) 272–277
6. Puget, J.F.: Breaking all value symmetries in surjection problems. In, proceedings of CP (2005) 490–504
7. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K., Smith, B.: Symmetry definitions for constraint satisfaction problems. In, proceedings of CP (2005) 17–31
8. Freuder, E.: Eliminating interchangeable values in constraints satisfaction problems. Proc AAAI-91 (1991) 227–233
9. Benhamou, B.: Theoretical study of dominance in constraint satisfaction problems. 6th International Conference on Artificial Intelligence: Methodology, Systems and Applications (AIMSA-94), Sofia, Bulgaria, september (1994) 91–97
10. Benhamou, B., Saïdi, M.R.: Reasoning by dominance in not-equals binary constraint networks. Technical report, LSIS (www.lsis.org) (2006)
11. Benhamou, B.: Symmetry in not-equals binary constraint networks. SymCon'04: 4th International Workshop on Symmetry and Constraint Satisfaction Problems (2004)
12. Sewell, E.C.: An improved algorithm for exact graph coloring. DIMACS series on Discrete Mathematics and Theorical Computer Science (1995)

Distributed Stable Matching Problems with Ties and Incomplete Lists*

Ismel Brito and Pedro Meseguer

Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain
{ismel, pedro}@iiia.csic.es

Abstract. We consider the Stable Marriage Problem and the Stable Roommates Problem in presence of ties and incomplete preference lists. They can be solved by centralized algorithms, but this requires to make public preference lists, something that members would prefer to avoid for privacy reasons. This motivates a distributed formulation to keep privacy. We propose a distributed constraint approach that solves all the considered problems, keeping privacy.

1 The Stable Marriage Problem

The Stable Marriage Problem (*SM*) [5] involves n men and n women. Each man m_i ranks women forming his preference list, and each woman w_j ranks men forming hers. A matching M is a one-to-one mapping between the two sexes. M is *stable* when there is no blocking pair (m, w) such that m and w, no partners in M, prefer one another to his/her partner in M. A *solution* is a stable matching.

There are several *SM* versions. With Incomplete Lists (*SMI*) some people may consider unacceptable some members of the other sex. With Ties (*SMT*), some people may consider equally acceptable some members of the other sex, so there is a tie among them. For SMT, three stability types have studied: weak, strong and super [6]. (m, w) is a weak blocking pair for M if m and w are not partners in M, and each of whom strictly prefers the other to his/her partner in M. (m, w) is a strong blocking pair for M if m and w are not partners in M, and one strictly prefers the other to his/her partner in M and the other is at least indifferent between them. (m, w) is a super blocking pair for M if m and w are not partners in M, and each of whom either strictly prefers the other to his/her partner in M or it is indifferent between them. With Ties and Incomplete Lists ($SMTI$), some person may consider as equally acceptable some members of the other sex, while others are unacceptable. The three stability types apply here.

Solvability conditions, complexity and solving algorithms of each *SM* version appear in Table 1. For *SMTI-weak*, different solutions may exist with different lengths, so it is of interest to find the matching of maximum cardinality. This is *SMTI-weak-max*, an optimization problem that is NP-hard.

* Supported by the Spanish project TIN2005-09312-C03-01.

Table 1. Solvability conditions, solving algorithm and complexity for the *SM* versions

SM version	∃ solution?	Size	All solutions		Algorithm	Compl
			Length	Partners		
SM	always	n	same	same	*EGS* [5]	poly
SMI	always	$\leq n$	same	same	*EGS* [5]	poly
SMT-weak	always	n	same	same	break ties + *EGS* [6]	poly
SMT-strong	not always	n	same	same	*STRONG* [6]	poly
SMT-super	not always	n	same	same	*SUPER* [6]	poly
SMTI-weak	always	$\leq n$	diff	diff	break ties + *EGS* [7]	poly
SMTI-strong	not always	$\leq n$	same	same	*STRONG2* [7]	poly
SMTI-super	not always	$\leq n$	same	same	*SUPER2* [7]	poly
SMTI-weak-max	always	$\leq n$	same	diff	break ties in all possible ways + *EGS* [7]	NP-hard

Table 2. Algorithms for solving *SM* and *DisSM* versions

DisSM problem	Centralized Algorithm	Extension to the distributed case, keeping privacy
DisSMT-weak	break ties + *EGS* [6]	break ties arbitrary + *DisEGS* [3]
DisSMT-strong	*STRONG* [6]	No extension [2]
DisSMT-super	*SUPER* [6]	Extension [2]
DisSMTI-weak	break ties + *EGS* [7]	break ties arbitrary + *DisEGS* [3]
DisSMTI-strong	*STRONG2* [7]	No extension [2]
DisSMTI-super	*SUPER2* [7]	No extension [2]
DisSMTI-weak-max	break ties in all possible ways + *EGS* [7]	Discussion [2]

SM appears to be naturally distributed. Each person would like to keep his/her preference lists private, which is not possible in the centralized case. This motivates the Distributed Stable Marriage (*DisSM*) [3], defined as *SM* plus a set of $2n$ agents. Each agent owns exactly one person. An agent knows all the information of its owned person, but it cannot access the information of people owned by other agents. A solution is a stable matching. Similarly, we define here the distributed versions with incomplete lists (*DisSMI*), with ties (*DisSMT*) and with ties and incomplete lists (*DisSMTI*) [1].

Is it possible to extend the centralized algorithms to the distributed setting keeping privacy? *DisEGS* is a distributed version of Extended Gale-Shapley (*EGS*) that maintains privacy. It was used to solve *DisSM* and *DisSMI* [3]. Here we focus on *DisSMT* and *DisSMTI*, that jointly with the three stability types, produce six decision problems plus one optimization problem. The extension of centralized algorithms to the distributed case appear in Table 2. Only three out of the six decision problems can be solved by extending the centralized algorithms to the distributed case while keeping preferences private. Details appear in [2].

2 Constraint Formulation

In [4], *SM* is modeled and solved as a binary CSP with $2n$ variables. Variable domains are the preference lists. Constraints are defined between men and women: C_{ij} is a table with all possible partial matchings involving man i and woman j. For any pair k, l ($k \in Dom(i)$, $l \in Dom(j)$), the element $C_{ij}[k, l]$ represents the partial matching $(m_i, w_k)(m_l, w_j)$; which could be: **A**llowed, **I**llegal, **B**locked or **S**upport. Tables with A, I, B, S are passed into 1/0 by A, S → 1, I, B → 0.

[1] [9] proposes a solving method based on encryption techniques.

For instances with ties, we consider the different definitions of stability. This affects the usage of **B**locked pair in C_{ij}. The definition given in [4] is valid for weak stability. For strong and super stability, we replace **B** definition in [4] by,

- $C_{ij}[k,l]$ = **B**locked if (m_i, w_j) is a strong blocking pair for (m_l, w_k).
- $C_{ij}[k,l]$ = **B**locked if (m_i, w_j) is a super blocking pair for (m_l, w_k).

Privacy is one of the main motivations for distributed *CSP* (*DisCSP*). We differentiate between value privacy and constraint privacy [1]. Value privacy implies that agents are not aware of other agent's values during the solving process and in the final solution. On constraint privacy, the Partially Known Constraints (*PKC*) model was presented. It assumes that when two agents i, j share a constraint C_{ij}, none of them fully knows it. Each agent knows the part of the constraint that it is able to build, using its own information. We say that agent i knows $C_{i(j)}$, and j knows $C_{(i)j}$. Similarly to [3], we use the constraint formulation of Section 2 to solve *DisSMT* and *DisSMTI* by the *DisFC* algorithm [1], keeping privacy of preference lists using the *PKC* model.

How can i build $C_{i(j)}$? Ordering rows of $C_{i(j)}$ following his preference list, all elements in rows above w_j are 1 (except m_i^{th} column that are 0). All elements in rows below w_j may be 1 or 0, depending on the ordering of columns (except m_i^{th} column that are 0). Since x_i does not know the preference list of y_j, columns are ordered lexicographically, and the elements below w_j row are ? (undecided). If there is a tie between w_j and $w_{j'}$ elements in $w_{j'}$ row are # (tie). $C_{i(j)}$ is,

$$C_{i(j)} = \begin{array}{cccccccc}
 & m_1 & \dots & & m_i & & \dots & m_n \\
w_{i_1} & 1 & \dots & 1 & 0 & 1 & \dots & 1 \\
 & & \dots & & & & & \\
 & 1 & \dots & 1 & 0 & 1 & \dots & 1 \\
w_j & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\
w_{j'} & \# & \dots & \# & 0 & \# & \dots & \# \\
 & ? & \dots & ? & 0 & ? & \dots & ? \\
 & & \dots & & & & & \\
w_{i_n} & ? & \dots & ? & 0 & ? & \dots & ?
\end{array}$$

A property of these tables is that all columns of $C_{i(j)}$ are equal, except m_i column. All rows of $C_{(i)j}$ are equal except w_j row [3]. $C_{ij} = C_{i(j)} \diamond C_{(i)j}$, $\diamond$ operates component to component. $\diamond$ depends on each type of stability.

How does a distributed algorithm achieve a global solution?. *DisFC* does it, using phase I only. *DisFC* instead of sending the assigned value to lower priority agents, it sends the domain subset that is compatible with the assigned value. Also, it replaces actual values by sequence numbers. After assignment, each man agent sends the domain that each woman may take to the women involved. For example, when an agent i assigns value k to x_i, it sends to j the row of $C_{i(j)}$ corresponding to value k. This row may contain 1's, 0's, ?'s or #'s. For each received domain, the agents search for a compatible value. If the domain contains ? or # entries, they are disambiguated, following the rules of $\diamond$ operation specific for each type of stability. When j receives a domain with ? or # values, it performs the $\diamond$ operation with a row of $C_{(i)j}$ different from i. Since all rows in $C_{(i)j}$ are equal, except i row, they will all give the same result. The $\diamond$ operation j will compute the corresponding row in the complete constraint C_{ij}, although j does not know to which value this row corresponds. After this operation the resulting domain will contain neither ? nor # values, and the receiver will have no ambiguity. If it finds no compatible value, it performs backtracking. The process repeats until finding a solution or detecting that no solution exits.

The $\diamond$ operation depends on the stability type. *Weak Stability.* From *weak blocking pair* definition, we realize that no matched pair (m, w) will be blocked for any other pair (m',w'), such that either m is indifferent between w and w' or w is indifferent between m and m'. So #'s are replaced by 1's in tables. Rules for the $\diamond$ operation of [3] apply. For *DisSMTI-weak*, not all stable matchings have the same length. One may desire to find a matching of maximum cardinality. With this aim, we consider the question 'Is there a *weakly stable* matching of size k?', where k starts with value n. If a weakly stable matching exits, it will be of maximum cardinality. Otherwise, the value k is decreased by one, and the problem is reconsidered. Modeling this idea with constraints, we add n variables, $u_1, u_2, \ldots, u_n$, plus an extra variable z, with the domains: $D(u_i) = \{0, 1\}, 1 \leq i \leq n$, $D(z) = \{k\}$. New constraints are: if $x_i < n + 1$ then $u_i = 1$ else $u_i = 0, 1 \leq i \leq n$ and $z = \sum_{i=1}^{n} u_i$. The agent that owns x_i also owns u_i. An extra agent owns z. *Strong Stability.* $\diamond$ includes #: $\# \diamond \# = 1$, $1 \diamond \# = 1$, $\# \diamond ? = 0$, $0 \diamond \# = 0$. *Super Stability.* $\diamond$ has a single change from strong stability:$\# \diamond \# = 0$.

Experimentally, we observe that when it is possible to extend an specialized centralized algorithm to the distributed case keeping privacy, the resulting algorithm is more efficient than *DisFC*. However, only a fraction of the *SM* versions can be solved in this way, while all of them can be solved by the generic distributed constraint formulation.

3 The Stable Roommates Problem

The Stable Roommates Problem (*SR*) consists of $2n$ participants, each ranks *all* other participants in strict order according to his/her preferences [5]. A matching is a set of n disjointed participant pairs. A matching is *stable* if it contains no *blocking pair*. A pair (p_i, p_j) is a blocking pair for M if p_i prefers p_j to his/her partner in M and p_j prefers p_i to his/her partner in M. There are instances that admit no stable matching. The goal is to determine whether an *SR* instance is solvable, and if so, find a stable matching. Like *SM*, there are versions with Incomplete Lists (*SRI*), with Ties (*SRT*), and with Ties and Incomplete Lists (*SRTI*). For versions including Ties, there are three stability types, weak, strong and super. Solvability conditions, complexity and solving algorithms of each *SR* version appear in Table 3. Considering *SRTI-weak*, different solutions may exist with different lengths, so it is of interest to find the matching of maximum cardinality. This is *SRTI-weak-max*, an optimization problem that is NP-hard.

With a motivation similar to *SM*, the Distributed Stable Roommates Problem (*DisSR*) [3] is defined by $2n$ persons plus a set of agents. Each person ranks all the other in his/her preference list. Each agent owns exactly one person. An agent knows the preference list of its owned person, but it does not know others' preferences. A solution is to find a stable matching, if it exists. Similarly, we

Table 3. Solvability conditions, solving algorithm and complexity for the *SR* versions

SR version	∃ solution?	Size	All solutions		Algorithm	Compl
			Length	Partners		
SR	not always	n	same	same	*Stable Roommates* [5]	poly
SRI	not always	$\leq n$	same	same	*Stable Roommates* [5]	poly
SRT-weak	not always	n	same	same	break ties in all possible ways + *Stable Roommates*, until finding a solution [8]	NP-comp
SRT-strong	not always	n	same	same	? [8]	?
SRT-super	not always	n	same	same	*SRT-super* [8]	poly
SRTI-weak	not always	$\leq n$	diff	diff	break ties in all possible ways + *Stable Roommates*, until finding a solution [8]	NP-comp
SRTI-strong	not always	$\leq n$	same	same	? [8]	?
SRTI-super	not always	$\leq n$	same	same	*SRTI-super* [8]	poly
SRTI-weak-max	not always	$\leq n$	same	diff	break ties in all possible ways + *Stable Roommates* [8]	NP-hard

Table 4. Algorithms for solving *SR* and *DisSR* versions

DisSR problem	Centralized Algorithm	Extension to the distributed case, keeping privacy
DisSRT-weak	break ties in all possible ways + *Stable Roommates*, until finding a solution [8]	No extension
DisSRT-strong	? [8]	No extension
DisSRT-super	*SRT-super* [8]	Extension
DisSRTI-weak	break ties in all possible ways + *Stable Roommates*, until finding a solution [8]	No extension
DisSRTI-strong	? [8]	No extension
DisSRTI-super	*SRTI-super* [8]	Extension
DisSRTI-weak-max	break ties in all possible ways + *Stable Roommates* [8]	No extension

define the distributed versions with incomplete lists (*DisSRI*), with ties (*DisSRT*) and with ties and incomplete lists (*DisSRTI*).

We investigate if centralized algorithms can be extended to the distributed case keeping privacy, focusing on *DisSRT* and *DisSRTI.* Their resolution is summarized in Table 4. We conclude that only two decision problems can be solved by extending the centralized algorithms to the distributed case while keeping privacy. Experimentally, we get similar results to those obtained for *SM*.

References

1. Brito I., Meseguer P. Distributed Forward Checking. *Proc. CP-03*, 801–806, 2003.
2. Brito I., Meseguer P. Distributed Stable Marriage with Ties and Incomplete Lists. *Proc. ECAI-06 Workshop on Distributed Constraint Satisfaction*, 2006.
3. Brito I., Meseguer P. Distributed Stable Matching Problem.*Proc.CP-05*,152–166.
4. Gent I., Irving R. W., Manlove D. F., Prosser P., Smith B. M. A constraint programming approach to the stable marriage problem. *Proc. CP-01*, 225-239, 2001.
5. Gusfield D., Irving R. W. *The Stable Marriage Problem: Structure and Algorithms.* The MIT Press, 1989.
6. Irving R.W.Stable Marriage and Indifference.*Discr. Appl. Maths.*,48:261–272, 1994.
7. Manlove D.F. Stable marriage with Ties and Unacceptable Partners. *TR-1999-29*, Dep. Computing Science, Univ. Glasgow, 1999.
8. Irving, R. W., Manlove, D. F. The roommates problem with ties. *J. Algorithms*,43(1):85-105, 2002.
9. Silaghi, M.C. and Zanker,M. and Bartak,R., Desk-mates (Stable Matching) with Privacy of Preferences, and a new Distributed CSP Framework, *Proc. of CP'2004 Workshop on Immediate Applications of Constraint Programming*, 2004

Soft Arc Consistency Applied to Optimal Planning

Martin Cooper, Sylvain Cussat-Blanc, Marie de Roquemaurel, and Pierre Régnier

IRIT, 118 route de Narbonne, 31062 Toulouse cedex 9, France
{cooper, cussat, deroquemaurel, regnier}@irit.fr

Abstract. We show in this article[1] how the Weighted CSP framework can be used to solve an optimisation version of numerical planning. The WCSP finds an optimal plan in the planning graph containing all solution plans of minimum length. Experimental trials were performed to study the impact of soft arc consistency techniques (FDAC and EDAC) on the efficiency of the search for an optimal plan in this graph. We conclude by giving a possible theoretical explanation for the fact that we were able to solve optimisation problems involving several hundred variables.

1 Introduction

In the field of planning, one of today's challenges is the solution of numerical problems to optimality. Some numerical planners perform heuristic choices, aiming simply to produce a good quality plan [5] while others merely compute *a posteriori* the cost of the solution-plan [6]. Some planners already use CSP to encode the planning graph [4], but never in a numerical approch to planning. We use the WCSP [7] framework to find a minimum cost plan, allowing the representation of strict constraints and an optimisation criterion expressed as the aggregation of cost functions.

2 WCSP and Numerical Planning

2.1 Numerical Planning Graph

A numerical planning problem is a triple $\langle A, I, G\rangle$ such that the initial state I is a finite set of propositional and numerical variables (or fluents) with their initial assignments, A is a set of actions, i.e. triples $\langle prec(a), effect(a), cost(a)\rangle$, where $prec(a)$ is the set of preconditions of action a, $effect(a)$ is the set of effects of a (Adds, Deletes and Modifiers of fluents), $cost(a)$ is the cost of applying a. An action a is applicable in a state S iff its preconditions are satisfied. A proposition p is satisfied in S iff $p \in S$; a numerical condition c is satisfied in state S iff the numerical variables of c are defined in S and verify c. G is the set of propositional and numerical goals to be satisfied. One of the most efficient and

[1] An extended version of this paper is available at http://www.irit.fr/recherches/RPDMP/persos/Regnier/PLANIF/index.html

influential algorithms in the field of planning is that of GRAPHPLAN [1]. We had to adapt the construction of the planning graph in order to solve numerical planning problems. The numerical planning graph G_p is a disjunctive graph which can be considered as a compact summary of all solution-plans up to a given maximal length. Actions can be applied in parallel but with three restrictions (called *interference*): (1) none deletes a precondition or an add-effect of another, (2) they have no numerical variable in common and (3) for each pair of distinct preconditions of actions at level i, there is at least one pair of non-interfering actions at the level $i-1$ which produce them.

In GRAPHPLAN, the planning graph G_p is extended level by level until either the extraction of a solution-plan is successful or the planning graph levels off. Level-off occurs when the sets of actions, fluents and mutexes (mutual exclusion constraints) are identical at levels i and $i+1$. It can be shown that if no solution has been found in a planning graph that has levelled off, then no solution exists. The first level of the graph consists simply of fluents corresponding to the initial state I. The next levels $i > 0$ are developed using the following algorithm:

1. We find all actions whose preconditions are satisfied in level $i-1$. To maintain at level i a fluent f present at level $i-1$, there is a noop action which has f as its only precondition and its only effect.
2. Next, we have to calculate the mutual exclusion relations (or mutexes) between actions. Two actions are mutex iff one interferes with the other.
3. We can now add to the graph all the fluents produced by these level i actions. As for the actions, we have to search for interferences between fluents: Two fluents are mutex at a given level i if there is no couple of non-mutex actions at the same level i that add these fluents.
4. The final step of level construction is to check if the goal is satisfied in the current state. If this is the case, the algorithm halts and we extract a solution plan from the graph. Otherwise, we go back to step 1.

The quality of a numerical plan can be estimated through a function known as a plan metric. We consider only the problem of minimising a linear additive metric, the sum of the costs of the actions in a plan P.

After having constructed the numerical planning graph, we reduce it by eliminating actions which cannot possibly be part of a solution-plan, ie all actions and fluents that do not have a path to any goal fluent. In preparation for coding the numerical planning graph as a WCSP (see Section 2.2), we rename the fluents $f_1, f_2, \ldots$ and renumber the actions 1,2,..., starting at the last level.

2.2 Coding the Planning Graph as a WCSP

Once the numerical planning graph has been constructed and reduced, we code it as a WCSP as follows:

1. *Creation of variables and domains:* for each fluent (not present in the initial state), we create a variable whose domain is the set of actions which produce this fluent. For each fluent (not present in the goal state) we add the value -1 to represent its non-activation.

2. *Translation of mutexes between fluents:* for all mutex fluents f_i and f_j, we code the fact that f_i et f_j cannot both be activated: $(f_i = -1) \vee (f_j = -1)$.
3. *Translation of mutexes between actions:* for all mutex actions a and b with (respective) effects f_i and f_j $(f_i \neq f_j)$, this mutual exclusion is coded by a constraint which states that f_i and f_j cannot simultaneously be triggered by the actions a and b: $\neg((f_i = a) \wedge (f_j = b))$.
4. *Translation of activity arcs:* the activation of a fluent f_i produced by an action a implies the activation of the preconditions of this action. This is coded by activity constraints: $\forall f_j \in prec(a)$, $(f_i = a) \Rightarrow (f_j \neq -1)$.
5. *Translation of the cost of actions:* For each value a, we add a unary constraint for $f = a$ corresponding to the cost of the action. No-ops have cost 0.
6. *Actions with multiple effects:* when an action a produces several fluents $f_i \in effect(a)$, the cost of this action could be counted several times. To avoid this problem, we create an intermediary fluent f^{int} with domain $\{a, -1\}$. Furthermore, the action a is replaced by a false action a^{int} (cost 0) in the domains of each f_i. We add the activity constraint $(f_i = a^{int}) \Rightarrow (f^{int} = a)$ between the fluent f^{int} and each of the fluents $f_i \in effect(a)$.

A limitation of our approach is that we only search for the optimal plan among parallel plans of length L, where L is the length of a shortest parallel solution-plan. Once this optimal plan among shortest-length plans has been discovered, it could, of course, be used as a good lower bound in the search for an optimal plan of any fixed length. Note, however, that the search space of all numerical plans is, in the worst case, infinite.

2.3 Search for an Optimal Solution to the WCSP

To simplify the WCSP [7] before solving it, we can apply soft arc consistency algorithms. Node consistency (NC) corresponds to taking the sum of minimum unary costs at each variable as a lower bound.

Propagating all infinite costs (between unary and binary constraints) and projecting binary costs until convergence establishes (soft) arc consistency (SAC). In order to have non-zero costs available as soon as possible during search, directional arc consistency (DAC) always chooses to send costs (via project and extend operations) towards variables which occur earlier in the instantation order. Full directional arc consistency (FDAC) [2] is the combination of directional arc consistency and arc consistency.

FDAC has recently been extended to existential directional arc consistency (EDAC) [3] which also performs the following operation: if for *each* value a in the domain of variable X_i, there exists a neighbouring variable X_j such that it is possible to increase $c_i(a)$ by sending costs from c_j and c_{ij}, then perform all these operations and then establish NC at X_i.

3 Experimental Trials

To test the utility of this approach, we carried out a large number of trials on different benchmark problems from IPC (International Planning Competition)

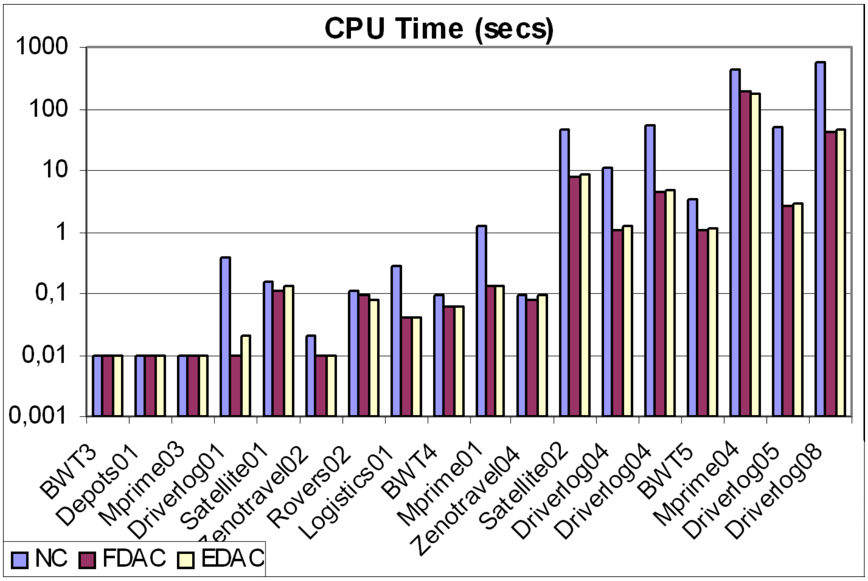


Fig. 1. Results of the trials using NC, FDAC and EDAC

covering diverse application areas (Blocksworld (bwt), Depots, Driverlog, Logistics, Mprime, Rover, Satellite and Zenotravel). We used the Toolbar library[2] [3] to solve the WCSP derived from the numerical planning graph[3]. The IPC benchmarks are derived from real problems and are highly structured. For each different domain, there is a series of problems of increasing difficulty, this being a function of the number of available actions and the number of fluents in the initial and goal states. The solution-plans for the hardest problems consist of thousands of actions. These benchmarks can be found at the IPC website[4].

We tested different soft arc consistency algorithms (NC, FDAC and EDAC) on different problems from the benchmark domains. NC, FDAC or EDAC was established at each node of the branch-and-bound search. Fig 1 compares the performances of these three soft arc consistency algorithms in terms of CPU time.

Firstly, comparing NC and FDAC (Figure 1), we observed that the number of nodes visited was always less with FDAC and that, on average, FDAC visited 17 times less nodes than NC. As for CPU time, FDAC is on average 7 times faster than NC. We can conclude that the use of FDAC significantly improves the time to extract an optimal solution from the numerical planning graph coded as a WCSP. Figure 1 also allows us to compare FDAC and EDAC. We found no significant difference between the two techniques. On average, EDAC visited

[2] http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro

[3] wcsp files are available here: http://mulcyber.toulouse.inra.fr/plugins/scmcvs/cvsweb.php/benchs/planning/?cvsroot=toolbar

[4] http://ipc.icaps-conference.org/

5% less nodes but used 6% more CPU time. We can therefore conclude that EDAC, unlike on random problems of similar density [3], does not provide an improvement compared with FDAC on the problems tested.

The most striking result of our experimental trials is that we were able to solve real optimal planning problems coded as WCSPs with several hundred variables (maximum search space size about 10^{376}), even though previous trials on random WCSPs had indicated a practical upper limit of about 50 variables (search space size 10^{45}) [3]. The good performance of intelligent branch and bound search on planning problems can probably be explained by the existence of many crisp constraints and the structure of the constraint graph: the variables can be divided into levels (corresponding to the levels of the planning graph) and binary constraints only exist between variables at the same or adjacent levels. If there are L levels, then at level $\frac{L}{2}$, i.e. after instantiating half of the problem variables, we can already apply approximately half of the constraints. In a random problem, when half of the variables have been instantiated, we can only apply approximately one quarter of the constraints. We formalize this idea in the following definition.

Definition 1. *A WCSP is* linearly incremental *under an ordering $X_1, \ldots, X_n$ of its variables if, for all $p \in \{1, \ldots, n\}$, the number c_p of constraints whose scopes are subsets of $\{X_1, \ldots, X_p\}$ satisfies $c_p = \frac{c}{n}(p + \circ(n))$, where c is the total number of constraints.*

A random problem is not linearly incremental, since in this case $c_p = \frac{cp(p-1)/2}{n(n-1)/2}$ $= O(\frac{cp^2}{n^2})$. In the optimal planning problem under consideration in this paper, assuming for simplicity that there are the same number of variables in each of L levels and that L, n/L are both $\circ(n)$, we have, for p a multiple of n/L: $c_p = \frac{c(\frac{pL}{n}-1)}{L-1} = \frac{c}{n}(p + \circ(n))$ and hence the problem is linearly incremental.

References

1. A. Blum & M. Furst, "Fast planning through planning graph analysis", *AI* 90 p.281-300 (1997).
2. M.C. Cooper, "Reduction operations in fuzzy and valued constraint satisfaction", *Fuzzy Sets and Systems* 134, p.311-342 (2003).
3. S. de Givry, F. Heras, M. Zytnicki & J. Larrosa, "Existential arc consistency: Getting closer to full arc consistency in weighted CSPs", *IJCAI 2005* p.84-89.
4. M. Do & S. Kambhampati, "Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP", *AI* 132 p.151-182 (2001)
5. P. Haslum & H. Geffner, "Heuristic Planning with Time and Resources", *European Conference on Planning* (2001).
6. J. Hoffmann, "The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables", *JAIR* 20 p.291-341 (2003).
7. J. Larrosa & T. Schiex, "In the quest of the best form of local consistency for Weighted CSP", *IJCAI 2003* p.239-244.

A Note on Low Autocorrelation Binary Sequences

Iván Dotú[1] and Pascal Van Hentenryck[2]

[1] Departamento De Ingeniería Informática, Universidad Autónoma de Madrid
[2] Brown University, Box 1910, Providence, RI 02912

Abstract. The Low Autocorrelation Binary Sequences problem (LABS) is problem 005 in the CSPLIB library, where it is stated that "these problems pose a significant challenge to local search methods". This paper presents a straighforward tabu search that systematically finds the optimal solutions for all tested instances.

1 Introduction

Sequences $S = \{s_1 =, \ldots, s_N\}$ ($s_i \in \{-1, +1\}$) with low off-peak autocorrelations

$$C_k(S) = \sum_{i=1}^{N-k} s_i s_{i+k}$$

have applications in many communication and electrical engineering problems [14], including high-precision interplanetary radar measurements [15]. The The *Low Autocorrelation Binary Sequences* (LABS) Problem consists in finding a sequence S with minimal energy

$$E(S) = \sum_{k=1}^{N-1} C_k^2(S).$$

A well-known measure of optimality for a sequence, known as the Bernasconi model [2], is given by $F \approx \frac{N^2}{2E}$ and is estimated to be about $F \approx 9.3$ [9].

The LABS problem is a challenging problem for constraint programming and is listed as problem 005 in the CSPLIB library [4]. The difficulty comes from its symmetric nature and the limited propagation [5]. The optimal sequences for up to $N = 50$ are known and given in [4]). It is also claimed on the CSPLIB web site that these problems pose a significant challenge to local search methods.

This paper proposes a trivial tabu-search algorithm with restarts that finds optimal sequences for up to $N = 48$. The rest of the paper is organized as follows: Section 2 reviews related work. Section 3 presents the algorithm. Section 4 presents the experimental results and Section 5 concludes the paper and discusses future work.

2 Related Work

The LABS problem attracted the attention of many communities since the early 70s. Physicists and mathematicians studied the structure of its search space. [7] found sequences with optimal energy up to $N = 32$ using exhaustive search and [9] found optimal solutions up to $N = 48$ by means of branch-and-bound and symmetry breaking. Although this latter approach clearly outperformed the former, the algorithm took about 1 hour to solve $N = 39$ using a Sun Sparc 20 workstation with 4 processors.

A substantial piece of work has been conducted on a special restricted type of sequence: the *skew-symmetric sequences* of odd length $N = 2n - 1$, which satisfy

$$s_{n+l} = (-1)^l s_{n-l}, \text{ where } l = 1, \ldots, n-1.$$

which guarantees that all C_k with odd k have a 0 value. Restricting attention to skew-symmetric sequences allow for larger values of N (up to 71) but optimal solutions are not guaranteed to be skew-symmetric [6,8]. Many heuristic procedures were developed to find skew-symmetric sequences [1], including simulated annealing ([2]), evolutionary algorithms [8], and techniques inspired by molecular evolution ([16]). These approaches on skew-symmetric sequences yield sequences with a *merit factor* $F \approx 6$ and thus very far from optimal solutions. ([2,7]). The conclusion of that research line was that optimal sequences must be extremely isolated. In particularm, it is said that stochastic search procedures are not well suited to find this "golf holes" [9], an observation also made on the CSPLIB site.

Constraint programming was also applied to LABS problems. Gent and Smith [5] used symmetric breaking and reported CPU times around $45,780$ seconds for $N = 35$. More recently, a hybrid branch & bound and local search technique was proposed by Prestwich [11]. This Constrained Local Search (CLS) algorithm assigns unassigned variables with consistent values until a failure occurs. Then CLS unassigns B variables either randomly or heuristically. In this case, B is set to 1, variables are selected randomly, and random restarts are used. The starting point of Prestwich's CLS is a simplified variant of the branch-and-bound technique presented in [9] but without symmetry breaking. CLS is able to find optimal sequences up to $N = 48$ and is faster than earlier approaches. Recently, reference [3] proposed an effective local search algorithm based on Kerninghan and Lin's meta-heuristic.

3 A Tabu-Search Algorithm for LABS

The Modeling The model is straightforward: there is a decision variable with every element in the sequence. A configuration σ is a complete assignment of variables to values and $\sigma(i)$ denotes the value assigned to variable s_i in configuration σ. There are no constraints in this problem but the algorithm maintains every C_k incrementally to compute and update the energy E. The energy of a configuration σ is denoted by $E(\sigma)$. The problem thus consists of finding a configuration σ minimizing $E(\sigma)$.

The Neighborhood. The neighborhood of the local search is also straightforward: it simply consists of flipping the value of a variable. Thus, the neighbors of a configuration σ can be defined as

$$\mathcal{M}(\sigma) = \{\sigma' \mid i \in 0..N-1 \;\&\; \forall j \neq i : \sigma'(j) = \sigma(j) \;\&\; \sigma'(i) = \sigma(i) \times (-1)\}.$$

```
1.  LABSLS($N$)
2.     $\sigma \leftarrow$ random configuration;
3.     $\sigma^* \leftarrow \sigma$;
4.     $k \leftarrow 0$;
5.     $s \leftarrow 0$;
6.     while $k \leq maxIt$ do
7.        select $\sigma' \in \mathcal{M}^*(\sigma, \sigma^*)$ minimizing $E(\sigma')$;
8.        $\sigma \leftarrow \sigma'$;
9.        if $E(\sigma) < E(\sigma^*)$ then
10.          $\sigma^* \leftarrow \sigma$;
11.          $s \leftarrow 0$;
12.       else if $s > maxStable$ then
13.          $\sigma \leftarrow$random configuration;
14.          $s \leftarrow 0$;
15.       else
16.          $s$++;
17.       $k$++;
```

Fig. 1. Algorithm for Low Autocorrelation Binary Sequences

The Tabu Component. The tabu component is again straithforward: it maintains a fixed-length tabu list to avoid flipping recently considered variable. It also uses an aspiration criterion to overwrite the tabu status when the move would lead to the best solution found so far. The so-obtained neighborhood is denoted by $\mathcal{M}^*(\sigma, \sigma^*)$ for a configuration σ and a best-found solution σ^*.

The Tabu-Search Algorithm. Figure 1 depicts the tabu search with a restarting component. The initial configuration is randomly generated. Each iteration selects the best move in the neighborhood (line 7). The restarting component simply reinitializes the search from a random configuration whenever the best configuration found so far has not been improved upon for *maxStable* iterations.

4 Experimental Results

This section reports our experimental results to find optimal LABSs up to $N = 48$ (the only results presented in [11]). A table of optimal solutions for $N = 17 - 50$ can be found on the CSPLIB website [4]. The goal is simply to show the effectiveness of the simple tabu search, not to provide a systematic comparison with other algorithms.

Table 1 reports the mean times in seconds for a set of 10 runs for each $N = 21 - 47$ and 4 runs for $N = 48$, and compares them with one of the fastest approaches (CLS from [11] on a 300 MHz DEC Alpha Server 1000A 5/300). The energy E and *merit factor* F are also depicted for every instance. Experiments were run on a 3.01GHz PC under linux. The algorithm was run with a $maxStable$ factor of $1,000$ iterations and maximum limit of iterations of $10,000,000$. Depicted mean times correspond to the times to find the optimal solution. The table also shows median and mean times and iterations.

Table 1. Experimental Results on the LABS Problem

N	E	F	CLS	LABSLS	speedup	medianIts	meanIts	medianT	meanT
21	26	8.48	3.85	0.23	16.74	5572	5768	0.22	0.23
22	39	6.21	0.74	0.02	37.00	2389	2964	0.03	0.02
23	47	5.63	2.21	0.12	18.42	793	2399	0.04	0.12
24	36	8.00	9.55	0.17	56.18	2389	2964	0.14	0.17
25	36	8.68	1.72	0.62	2.78	6311	9677	0.41	0.62
26	45	7.51	3.12	0.23	13.57	2287	3247	0.16	0.23
27	37	9.85	33.9	1.77	19.16	19138	22499	1.51	1.77
28	50	7.84	37.2	0.96	38.75	8685	11076	0.75	0.96
29	62	6.78	43.3	1.24	34.92	12920	12894	1.24	1.24
30	59	7.63	25.3	3.08	8.21	28951	29382	3.04	3.08
31	67	7.17	117	2.59	45.17	16019	22599	1.84	2.59
32	64	8.00	319	6.47	49.30	44544	51687	5.58	6.47
33	64	8.51	482	17.80	27.08	100278	130509	13.67	17.80
34	65	8.89	422	14.80	28.51	125260	100000	18.51	14.80
35	73	8.39	709	44.85	15.81	107415	277547	17.22	44.85
36	82	7.90	981	53.21	18.44	205506	305737	36.52	53.21
37	86	7.96	2096	78.92	26.56	354694	417485	66.29	78.92
38	87	8.30	1679	147.17	11.41	578751	724586	117.20	147.17
39	99	7.68	5300	138.99	39.56	666494	640236	144.15	138.99
40	108	7.41	14305	260.11	55.00	787803	1128835	183.66	260.11
41	108	7.78	21224	460.26	46.11	1618612	1857771	404.25	460.26
42	101	8.73	4890	466.73	10.48	1639049	1722619	446.95	466.73
43	109	8.48	46168	1600.63	28.84	4608116	5757618	1286.87	1600.63
44	122	7.93	27422	764.66	35.86	1930648	2547498	588.25	764.66
45	118	8.58	52920	1103.48	47.96	3536722	3448381	1188.24	1103.48
46	131	8.08	5184	703.32	7.37	2154834	2053875	728.30	703.32
47	135	8.18	26280	1005.03	26.15	2416869	2749650	863.86	1005.03
48	140	8.23	12096	964.13	12.55	2024970	2553812	759.83	964.13

Observe that this simple tabu search algorithm quickly finds solutions for instances up to $N = 32$ (a few seconds or less), and finds solutions in reasonable time for the remaining instances. The algorithm is consistently 8 to 55 times faster than CLS, even for $N = 43 - 48$ were CLS reports only the times for a single run. This is particularly remarkable, since the energy is not specified in our approach, while it is set to the known optimal value plus 1 in CLS. Note also that the means and medians are very similar indicating the robustness of the algorithm.

Every sequence is symmetric in up to 8 ways. However, no symmetries have been exploited in this approach. Adding symmetries for local search has been found to be unlikely to improve performance for several problems ([12,13]).

5 Conclusion and Future Work

This paper proposed a simple local-search algorithm for LABS problems that was shown to be effective and robust. In particular, it finds optimal sequences for up to

$N = 48$. These results are quite surprising given the claim (on the CSPLIB website and in [9]) that "these problems pose a significant challenge to local search methods". This seems to suggest (once again) that it is not easy to predict what makes a problem hard for local search methods.

Future work includes pushing the $N = 48$ limit, along with the study of novel hybridizations. A complete local search (exploring the search space exhaustively) has been developed, although it cannot (yet) compete with the local search presented here. A version of the complete local search with limited discrepancies produces similar results in quality to the tabu-search, indicating the potential for obtaining a complete and effective search procedure for LABS problems.

Acknowledgment

Thanks to the second reviewer for pointing out reference [3].

References

1. G. Beenker, T. Claasen and P. Hermens. Binary Sequences with a Maximally Flat Amplitude Spectrum. *Philips Journal of Research*, 40:289–304, 1985.
2. J. Bernasconi. Low Autocorrelation Binary Sequences: Statistical Mechanics and Configuration Space Analysis. *Physique*, 48:559, 1987.
3. F. Brglez, X. Li, M. Stallman and B. Militzer. Reliable Cost Prediction for Finding Optimal Solutions to LABS Problem: Evolutionary and Alternative Algorithms. Fifth International Workshop on Frontiers in Evolutionary Algorithms, Cary, NC, USA, 2003.
4. I. Gent and T. Walsh. CSPLIB: A Benchmark Library for Constraints. http://csplib.org
5. I. Gent and B. Smith. Symmetry Breaking During Search in Constraint Programming. *Research Report*, 99.02, 1999.
6. M.J. Golay. Sieves for Low Autocorrelation Binary Sequences. *IEEE Transactions on Information Theory*, 23:43–41, 1977.
7. M.J. Golay. The Merit Factor of Long Low Autocorrelation Binary Sequences. *IEEE Transactions on Information Theory*, 28:543–549, 1982.
8. C. de Groot, D. Wurtz, K. Hoffmann. Low Autocorrelation Binary Sequences: Exact Enumeration and Optimization by Evolutionary Strategies. *Optimization*, 23:369–384, 1992.
9. S. Mertens. Exhaustive Search for Low Autocorrelation Binary Sequences. *J. Phys. A: Math*, 29:473–481, 1996.
10. S.D. Prestwich. A Hybrid Local Search for Low Autocorrelation Binary Sequences. *Technical Report*, TR-00-01, 2000.
11. S.D. Prestwich. A Hybrid Search Architecture Applied to Hard Random 3-SAT and Low-Autocorrelation Binary Sequences. *CP 2000*, 337-352, 2000.
12. S.D. Prestwich. Negative Effects of Modeling Techniques on Search Performance. *Annals of Operations Research*, 118:137–150, 2003.
13. S. Prestwich and A. Roli Symmetry breaking and local search spaces. *In CPAIOR 2005*, volume 3524 of LNCS, Springer-Verlag, 2005.
14. M.R. Schroeder. Number theory in Science and Communication. *Berlin: Springer*, 118:137–150, 2003.
15. I. Shapiro, G. Pettengil, M. Ash, M. Stone, W. Smith, R. Ingalls and R. Brockelman. Fourth Test of General Relativity. *Phys. rev. Lett.*, 20:1265–1269, 1968.
16. Q. Wang. Optimization by Simulating Molecular Evolution. *Biol. Cybern*, 57:95–101, 1987.

Relaxations and Explanations for Quantified Constraint Satisfaction Problems*

Alex Ferguson and Barry O'Sullivan

[1] Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
[2] Centre for Telecommunications Value-Chain Research
{a.ferguson, b.osullivan}@4c.ucc.ie

1 Introduction

The Quantified CSP (QCSP) is a generalisation of the classical CSP in which some of the variables are universally quantified [3]. We consider the problem of relaxing an instance of the QCSP when it is unsatisfiable. We propose several novel forms of problem relaxations for the QCSP and present an algorithm for generating conflict-based explanations of inconsistency. Our motivation comes from problems in supply-chain management and conformant planning.

Definition 1 (Quantified CSP). *A* QCSP, ϕ, *has the form*

$$\phi \mathrel{\hat{=}} \mathcal{Q}.\mathcal{C} = Q_1 x_1 \in D(x_1) \cdots Q_n x_n \in D(x_n).\mathcal{C}(x_1, \ldots, x_n)$$

where $D(x_i)$ is the domain of x_i; $\mathcal{C}$ is a set of constraints defined over the variables $x_1 \ldots x_n$; and $\mathcal{Q}$ is a sequence of quantifiers over the variables $x_1 \ldots x_n$ where each Q_i $(1 \leq i \leq n)$ is either an existential, $\exists$, or a universal, $\forall$, quantifier[1]*. The expression $\exists x_i.c$ means that "there exists a value $a \in D(x_i)$ such that the assignment (x_i, a) satisfies c". Similarly, the expression $\forall x_i.c$ means that "for every value $a \in D(x_i)$, (x_i, a) satisfies c".*

Example 1 (Quantified CSP). Consider a QCSP defined on the variables x_1 and x_2 such that $D(x_1) = \{1, 2\}$ and $D(x_2) = \{1, 2, 3\}$ as follows: $\exists x_1 \forall x_2.\{x_1 < x_2\}$. This QCSP is false. This is because for any choice of value for variable x_1 there is at least one value in the domain of x_2 that ensures that the constraint $x_1 < x_2$ will be violated. ▲

2 Relaxation

We observe that many real world problems are over-constrained, and that we are often interested in relaxing a problem so that we find a solution that is at least relatively satisfactory. Problem relaxations give us a language with which we can *explain* the over-constrainedness of a problem.

* This work was supported by Science Foundation Ireland (Grant 03/CE3/I405). The authors thank Hubie Chen, Helene Fargier and Nic Wilson for their helpful comments.

[1] When the domain of a variable is clear from context we often write $Q_i x_i$ rather than $Q_i x_i \in D(x_i)$ in the quantifier sequence.

2.1 Relaxation for the Quantified CSP

Partial Constraint Satisfaction [4] identifies a number of relaxations for classical CSP, all of which can be regarded as enlarging either constraint relations or variable domains. We identity five classes of relaxation for QCSPs: the first two are familiar from CSP, the remaining three particular to QCSPs. We claim that these are *comprehensive* in representing all 'natural' relaxations of a given QCSP.

Single Constraint Relaxation. This consists of adding allowed tuples to an extensionally represented constraint, and is equivalent to the corresponding relaxation operation in classical CSP.

Example 2 (Single Constraint Relaxation). Consider the following QCSP, which is false:

$$\exists x_1 \in \{1, 2\} \forall x_2 \in \{1, 2, 3\}.\{x_1 < x_2\}.$$

If we relax the constraint between x_1 and x_2 from $<$ to $\leq$ the QCSP becomes true. This is because if x_1 is assigned 1, the constraint cannot be violated by any value of x_2. ▲

Relaxation of Existentially Quantified Domains. Adding values to the domain of an existentially quantified variable corresponds to the CSP case of enlarging a variable domain.

Example 3 (Relaxation of Existential Domains). Consider the QCSP in Example 2. A suitable relaxation is the following, in which the value 0 is added to the domain of x_1:

$$\exists x_1 \in \{0, 1, 2\} \forall x_2 \in \{1, 2, 3\}.\{x_1 < x_2\};$$

the relaxed QCSP is true since setting x_1 to 0 ensures that any assignment to x_2 satisfies the constraint. ▲

Relaxation of Universally Quantified Domains. A class of relaxation particular to QCSPs is to *remove* values from the domains of *universally* quantified variables.

Example 4 (Relaxation of Universal Domains). Returning to the QCSP of Example 2: this is false, as 1 can be assigned to x_2, for which there is no satisfying counter-assignment. However, if we relax the domain of the universally quantified variable x_2 so that it no longer contains the value 1 as follows:

$$\exists x_1 \in \{1, 2\} \forall x_2 \in \{2, 3\}.\{x_1 < x_2\},$$

then the relaxed QCSP is true. ▲

Quantifier Relaxation. A fourth form of relaxation, also with no direct equivalent in classical CSP, is to *reverse* the quantification of a variable from universal to existential.

Example 5 (Quantifier Relaxation). We revisit the QCSP presented in Example 2, which is false. If we relax the universal quantifier on variable x_2 to be existential to get:

$$\exists x_1 \in \{1, 2\} \exists x_2 \in \{1, 2, 3\}.\{x_1 < x_2\},$$

then the relaxed QCSP is true. This is valid if the associated domain is non-empty. ▲

Quantifier Moving. A fourth class of relaxation corresponds to moving an existentially quantified variable to the right in the sequence of quantifiers.

Example 6 (Quantifier Moving). Consider the following QCSP, which is false:

$$\exists x_1 \in \{\mathsf{true}, \mathsf{false}\} \forall x_2 \in \{\mathsf{true}, \mathsf{false}\}.\{(x_1 \vee x_2), (\neg x_1 \vee \neg x_2)\}.$$

We relax this problem by moving the quantification $\exists x_1$ to the right, giving sequence $\forall x_2 \in \{\mathsf{true}, \mathsf{false}\} \exists x_1 \in \{\mathsf{true}, \mathsf{false}\}$. The resulting QCSP is true. This corresponds to delaying a key decision until better information is available about assignments to universal variables. ▲

2.2 QCSP Relaxation as Requirement Relaxation

We now present a uniform treatment of the relaxation of both quantifiers and constraints. *Requirements* correspond to either a constraint in the QCSP, or a universal quantifier, and we frame relaxation of each as instances of *requirement relaxation*, over a partial order defined for that purpose.

Definition 2 (Relaxation of a QCSP). *We define the relaxation $\phi[r']$ of a* QCSP *ϕ on a single requirement, r, to be its replacement with the given requirement r'. This can then be extended to a set of relaxations R: $\phi[\emptyset] = \phi$, $\phi[\{r'\} \cup R] = (\phi[r'])[R]$.*

Definition 3 (Ordering over Requirement Relaxations). *Given the set of possible relaxations, $\mathcal{R}(r)$, of a requirement r, we say that $(r_1 \in \mathcal{R}(r)) \sqsubseteq (r_2 \in \mathcal{R}(r))$ iff for any problem ϕ, if $\phi[r_1]$ is satisfiable then $\phi[r_2]$ is, necessarily. We further require that this partial order also be a* meet-semilattice*. That is to say,* greatest lower bounds *are guaranteed to exist: if $r_1, r_2 \in \mathcal{R}(r)$, then $r_1 \sqcap r_2$ is well-defined. This corresponds to the unique requirement relaxation which is as constraining as both of its arguments, but no more.*

We now consider universal quantifier relaxation. Informally, the space of possible relaxations for a universal quantifier corresponds to restricting the quantifier to any one of the exponentially many subsets of the domain of the quantified variable, narrowing to a single choice of value, and thereafter widening to *existentially* quantified subsets. Of course, in practice, one can limit this powerset to a subset of tractable size.

Definition 4 (Requirement Relaxation for Universals). *Given a requirement on an universally quantified variable x, i.e. $r \hat{=} \forall x \in D(x)$, the set of relaxations $\mathcal{R}(\forall x \in D(x))$ is defined as:*

$$\mathcal{R}(r) \hat{=} \{(\forall x \in D'(x)) : \emptyset \subseteq D'(x) \subseteq D(x)\} \cup \{(\exists x \in D'(x)) : \emptyset \subseteq D'(x) \subseteq D(x)\}$$

The elements of $\mathcal{R}(\forall x \in D(x))$ form the following meet-semilattice:

$$\begin{aligned}
&(\forall x \in D(x)) \sqcap (\forall x \in D'(x)) \hat{=} (\forall x \in (D(x) \cup D'(x)));\\
&(\exists x \in D(x)) \sqcap (\exists x \in D'(x)) \hat{=} (\exists x \in (D(x) \cap D'(x)));\\
&(\forall x \in D(x)) \sqcap (\exists x \in D'(x)) \hat{=} (\forall x \in D(x)), \textit{if } D(x) \cap D'(x) \neq \emptyset;\\
&(\forall x \in D(x)) \sqcap (\exists x \in D'(x)) \hat{=} (\exists x \in \emptyset), \textit{if } D(x) \cap D'(x) = \emptyset.
\end{aligned}$$

Note the existence of unique top and bottom points, corresponding to trivially- and un-satisfiable quantifications: $\top = (\forall x \in \emptyset), \bot = (\exists x \in \emptyset)$.

We also define the space of relaxations, $\mathcal{R}(c)$, for a constraint c. The elements of $\mathcal{R}(c)$ form the usual lattice using intersection, that is, $\sqcap \hat{=} \cap$.

Definition 5 (Requirement Relaxation for Constraints). *Given a constraint c with scope $var(c)$ and relation $sol(c)$, we define its relaxations in terms of adding additional allowed tuples to $sol(c)$ as follows: $\mathcal{R}(c) \hat{=} \{sol'(c) : sol(c) \subseteq sol'(c) \subseteq \Pi_{x \in var(c)} D(x)\}$.*

We can now define the space of possible relaxations of a given QCSP in terms of its constituent requirements in the natural way, in terms of the cross-product of the available relaxations for each, and similarly the associated comparison and meet operations.

3 From Relaxations to Explanations

A familiar notion of explanation is that based on *minimal conflicts* [6], which we generalise to *minimally conflicting explanations*, or alternatively, *maximally relaxed conflict-based explanations*. We define explanation with respect to a (typically incomplete) consistency propagation method Π, such as QAC [2], in a similar way to Junker [6].

Definition 6 (Maximally Relaxed Explanation). *Given a consistency propagator Π, a* maximally relaxed (Π-conflict-based) explanation *of a Π-inconsistent* QCSP *ϕ, is a maximal* QCSP*, X, that is inconsistent with respect to Π; i.e. such that $\phi \sqsubseteq X$, $\bot \in \Pi(X)$ and $\forall X'$ such that $X \sqsubset X', \bot \notin \Pi(X')$.*

We adopt a scheme similar to that of the QUICKXPLAIN 'family' of algorithms [6], in particular REPLAYXPLAIN. We present QUANTIFIEDXPLAIN (Algorithm 1), which is parameterised by a semilattice of available relaxations for each requirement. We begin with a maximal relaxation of *each* requirement, and progressively tighten these one by one, by a minimal amount to ensure maximality of the final relaxation. Each time an inconsistency is detected, we eliminate all relaxations *tighter* than the current approximation. At the same time, we take the last-relaxed requirement to be *no more* relaxed than the last-chosen value that produced the inconsistency. Eventually only one possibility remains from each relaxation, thus fully determining the chosen explanation.

Theorem 1 (Maximally Relaxed Explanations using QUANTIFIEDXPLAIN). *If $\bot \in \Pi(\phi)$, and $\mathcal{R}$ is a set of available relaxations of some, all or none of the requirements of ϕ, and* QUANTIFIEDXPLAIN *(ϕ, $\mathcal{R}$) = X, then: $\bot \in \Pi(X)$, and if $X' = X[r]$ for some $r \in \mathcal{R}$ such that $X \sqsubset X'$, then $\bot \notin \Pi(X')$.*

Proof. (Sketch) If $\bot \in \Pi(\phi[R])$ where $R = \{r_1, \ldots, r_{m+n}\}$, and each r_i is maximal in $\mathcal{R}_i$, then $\phi[R]$ is a maximally relaxed explanation. If $\bot \notin \Pi(\phi)$ then no conflict exists. If $\bot \in \Pi(\phi[r_i])$, and X is a maximally relaxed explanation for ϕ, then X is a maximally relaxed explanation for ϕ where $\mathcal{R}'_i = \{r \in \mathcal{R}_i, r_i \sqsubseteq r\}$. If $\bot \in \Pi(\phi[r_i])$ and $\bot \notin \Pi(\phi[r'_i])$, where $r'_i \in \text{minima}\{r'' : r'' \in \mathcal{R}_i, r_i \sqsubset r''\}$, then if X is a maximally relaxed explanation for ϕ, then X is a maximally relaxed explanation for ϕ where $\mathcal{R}'_i = \{r \in \mathcal{R}_i, r \sqsubseteq r_i\}$. □

Algorithm 1. QUANTIFIEDXPLAIN($\phi, \mathcal{R}$)

```
Input  : A QCSP $\phi$; a set of relaxation spaces for each quantifier and constraint, $\mathcal{R} \subseteq \mathcal{R}(\phi)$).
Output : A maximally relaxed conflict-based explanation for $\phi$.

if $\bot \notin \Pi(\phi)$ then return exception "no conflict";
enumerate $\mathcal{R}$ as $\mathcal{R}_1 \ldots \mathcal{R}_{m+n}$;
if $\forall i \in [1, m+n].|\mathcal{R}_i| = 1$ then return $\phi$;
while $\exists i : |\mathcal{R}_i| > 1$ do
    foreach $\mathcal{R}_i$ do select an $r_i$ from maxima($\mathcal{R}_i$);
    if $\bot \in \Pi(\phi)$ then return $\phi[\{r_1, \ldots, r_{m+n}\}]$;
    while $\bot \notin \Pi(\phi[\{r_1, \ldots, r_{m+n}\}])$ do
        select (any) $i$ s.t. $r_i \neq \bigsqcap(\mathcal{R}_i)$;
        choose an $r'$ from maxima$\{r : r \in \mathcal{R}_i, r_i \not\sqsubseteq r\}$;
        $r_i \leftarrow r' \sqcap r_i$;
    $\mathcal{R}_i \leftarrow \{r : r \in \mathcal{R}_i, r_i \sqsubseteq r \sqsubseteq r'\}$;
    foreach $\mathcal{R}_j, j \neq i$ do $\mathcal{R}_j \leftarrow \{r : r \in \mathcal{R}_j, r_j \sqsubseteq r\}$;
return $\phi[\{r_1, \ldots, r_{m+n}\}]$;
```

4 Related Work and Conclusions

Classical CSP concepts such as arc-consistency, satisfiability, interchangeability and symmetry have been extended to the QCSP [1, 7]. A number of techniques for solving the QCSP have also been proposed [5, 8].

Our work on relaxation and explanation of inconsistency is the first on these important aspects of reasoning about QCSPs. We have defined a number of different forms of relaxation not available in classical CSP, and presented an algorithm called QUANTIFIEDXPLAIN for computing minimally conflicting explanations for QCSPs.

References

1. L. Bordeaux, M. Cadoli, and T. Mancini. CSP properties for quantified constraints: Definitions and complexity. In *AAAI*, pages 360–365, 2005.
2. L. Bordeaux and E. Monfroy. Beyond NP: Arc-consistency for quantified constraints. In *CP*, pages 371–386, 2002.
3. H. Chen. *The Computational Complexity of Quantified Constraint Satisfaction.* PhD thesis, Cornell University, August 2004.
4. E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artif. Intell.*, 58(1-3):21–70, 1992.
5. I.P. Gent, P. Nightingale, and K. Stergiou. QCSP-solve: A solver for quantified constraint satisfaction problems. In *IJCAI*, pages 138–143, 2005.
6. U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.
7. N. Mamoulis and K. Stergiou. Algorithms for quantified constraint satisfaction problems. In *CP*, pages 752–756, 2004.
8. K. Stergiou. Repair-based methods for quantified CSPs. In *CP*, pages 652–666, 2005.

Static and Dynamic Structural Symmetry Breaking

Pierre Flener[1], Justin Pearson[1], Meinolf Sellmann[2], and Pascal Van Hentenryck[2]

[1] Dept of Information Technology, Uppsala University, Box 337, 751 05 Uppsala, Sweden
[2] Dept of Computer Science, Brown University, Box 1910, Providence, RI 02912, USA
{pierref, justin}@it.uu.se, {sello, pvh}@cs.brown.edu

Abstract. We reconsider the idea of structural symmetry breaking (SSB) for constraint satisfaction problems (CSPs). We show that the dynamic dominance checks used in symmetry breaking by dominance-detection search for CSPs with piecewise variable *and* value symmetries have a static counterpart: there exists a set of constraints that can be posted at the root node and that breaks *all* these symmetries. The amount of these symmetry-breaking constraints is *linear* in the size of the problem, but they possibly remove a super-exponential number of symmetries on both values and variables. Moreover, static and dynamic structural symmetry breaking coincide for static variable and value orderings.

1 Introduction

Symmetry breaking has been the topic of intense research in recent years. Substantial progress was achieved in many directions, often exhibiting significant speedups for complex real-life problems arising, say, in configuration and network design. One of the interesting recent developments has been the design of general symmetry-breaking schemes such as symmetry breaking by dominance detection (SBDD) and symmetry breaking during search (SBDS). SBDD [1,2] is particularly appealing as it combines low memory requirements with a number of dominance checks at each node linearly proportional to the depth of the search tree. It then became natural to study which classes of symmetries for CSPs admit polynomial-time dominance-checking algorithms. This issue was first studied in [9], where symmetry breaking for various classes of value symmetries was shown to take constant time and space (see also [7] for an elegant generalization to all value symmetries). It was revisited for CSPs with piecewise variable and value symmetry in [8], where a polynomial-time dominance-checking algorithm was given and the name 'structural symmetry breaking' (SSB) was coined. In parallel, researchers have investigated for many years (e.g., [4]) static symmetry breaking, which consists in adding constraints to the CSP in order to remove symmetries.

In this paper, after reviewing the basic concepts in Section 2, we show in Section 3 that the polynomial-time dominance-checking algorithm of [8] has a static counterpart, namely that there exists a static set of constraints for CSPs with piecewise symmetric variables and values that, when added to the CSP, results in a symmetry-free search tree. The amount of symmetry-breaking constraints is *linear* in the size of the problem, but possibly removes a super-exponential number of symmetries on both values and variables. In Section 4, we establish a clear link between static (SSSB) and dynamic structural symmetry breaking (DSSB) by showing that the obtained SSSB scheme explores the same tree as DSSB [8] whenever the variable and value orderings are fixed.

2 Basic Concepts

Definition 1 (CSP, Assignment, Solution). *A* constraint satisfaction problem (CSP) *is a triplet $\langle V, D, C\rangle$, where V denotes the set of variables, D denotes the set of possible values for these variables and is called their* domain, *and $C : (V \rightarrow D) \rightarrow Bool$ is a constraint that specifies which assignments of values to the variables are solutions. An* assignment *for a CSP $\mathcal{P} = \langle V, D, C\rangle$ is a function $\alpha : V \rightarrow D$. A* partial assignment *for a CSP $\mathcal{P} = \langle V, D, C\rangle$ is a function $\alpha : W \rightarrow D$, where $W \subseteq V$. The* scope *of α, denoted by $scope(\alpha)$, is W. A* solution *to a CSP $\mathcal{P} = \langle V, D, C\rangle$ is an assignment σ for $\mathcal{P}$ such that $C(\sigma) = \mathbf{true}$. The set of all solutions to a CSP $\mathcal{P}$ is denoted by $Sol(\mathcal{P})$.*

Definition 2 (Partition, Piecewise Bijection). *Given a set S and a set of sets $P = \{P_1, \ldots, P_n\}$ such that $S = \bigcup_i P_i$ and the P_i are pairwise non-overlapping, we say that P is a* partition *of S and that each P_i is a* component, *and we write $S = \sum_i P_i$. A bijection $b : S \rightarrow S$ is a* piecewise bijection *over $\sum_i P_i$ iff $\{b(e) \mid e \in P_i\} = P_i$.*

Definition 3 (Piecewise Symmetric CSP). *A CSP $\mathcal{P} = \langle \sum_k V_k, \sum_\ell D_\ell, C\rangle$ is a* piecewise symmetric CSP *iff, for each solution $\alpha \in Sol(\mathcal{P})$, each piecewise bijection τ over $\sum_\ell D_\ell$, and each piecewise bijection σ over $\sum_k V_k$, we have $\tau \circ \alpha \circ \sigma \in Sol(\mathcal{P})$.*

Definition 4 (Dominance Detection). *Given two partial assignments α and β for a piecewise symmetric CSP $\mathcal{P} = \langle \sum_k V_k, \sum_\ell D_\ell, C\rangle$, we say that α* dominates *β iff there exist piecewise bijections σ over $\sum_k V_k$ and τ over $\sum_\ell D_\ell$ such that $\alpha(v) = \tau \circ \beta \circ \sigma(v)$ for all $v \in scope(\alpha)$.*

Dominance detection constitutes the core operation of symmetry breaking by dominance detection (SBDD) [1,2], and its tractability immediately implies that we can efficiently limit ourselves to the exploration of symmetry-free search trees only. For piecewise symmetric CSPs, [8] showed that dominance detection is tractable.

3 Static SSB for Piecewise Symmetric CSPs

When we assume a total ordering of the variables $V = \{v_1, \ldots, v_n\}$ and the values $D = \{d_1, \ldots, d_m\}$, we can break the variable symmetries within each variable component as usual, by requiring that earlier variables take smaller or equal values. To break the value symmetries, we resort to structural abstractions, so-called *signatures*, which generalize from an exact assignment of values to variables by quantifying how often a given value is assigned to variables in each component. Let the frequency $f_h^k = |\{v_g \in V_k \mid v_g \in scope(\alpha) \ \& \ \alpha(v_g) = d_h\}|$ denote how often each value d_h is taken under partial assignment α by the variables in each variable component V_k. For a partial assignment α, we then denote by $\mathrm{sig}_\alpha(d_h) := (f_h^1, \ldots, f_h^a)$ the *signature* of d_h under α. Then, for all consecutive values d_h, d_{h+1} in the same value component, we require that their signatures are lexicographically non-increasing, i.e., $\mathrm{sig}_\alpha(d_h) \geq_{lex} \mathrm{sig}_\alpha(d_{h+1})$. So the problem boils down to computing the signatures of values efficiently. Fortunately, this is an easy task when using the existing global cardinality constraint (gcc) [6]. We thus propose to add the following static set of constraints to a piecewise symmetric CSP $\langle \sum_{k=1}^{a} V_k, \sum_{\ell=1}^{b} D_\ell, C \rangle$ with $V_k = \{v_{i(k)}, \dots, v_{i(k+1)-1}\}$ and $D_\ell = \{d_{j(\ell)}, \dots, d_{j(\ell+1)-1}\}$:

$$\begin{array}{l}
\forall\, 1 \le k \le a : \forall\, i(k) \le h < i(k+1) - 1 : v_h \le v_{h+1} \\
\forall\, 1 \le k \le a : \mathrm{gcc}(v_{i(k)}, \dots, v_{i(k+1)-1}, d_1, \dots, d_m, f_1^k, \dots, f_m^k) \\
\forall\, 1 \le \ell \le b : \forall\, j(\ell) \le h < j(\ell+1) - 1 : (f_h^1, \dots, f_h^a) \ge_{lex} (f_{h+1}^1, \dots, f_{h+1}^a)
\end{array}$$

where $i(k)$ denotes the index in $\{1, \dots, n\}$ of the first variable in V of variable component V_k, with $i(a+1) = n+1$, and $j(\ell)$ denotes the index in $\{1, \dots, m\}$ of the first value in D of value component D_ℓ, with $j(b+1) = m+1$.

Example. Consider scheduling study groups for two sets of five indistinguishable students each. There are six identical tables with four seats each. Let $\{v_1, \dots, v_5\} + \{v_6, \dots, v_{10}\}$ be the partitioned set of piecewise interchangeable variables, one for each student. Let the domain $\{t_1, \dots, t_6\}$ denote the set of tables, which are fully interchangeable. The static structural symmetry-breaking constraints are:

$$\begin{array}{l}
v_1 \le v_2 \le v_3 \le v_4 \le v_5,\ \ v_6 \le v_7 \le v_8 \le v_9 \le v_{10}, \\
\mathrm{gcc}(v_1, \dots, v_5,\ t_1, \dots, t_6,\ f_1^1, \dots, f_6^1),\ \ \mathrm{gcc}(v_6, \dots, v_{10},\ t_1, \dots, t_6,\ f_1^2, \dots, f_6^2), \\
(f_1^1, f_1^2) \ge_{lex} (f_2^1, f_2^2) \ge_{lex} \cdots \ge_{lex} (f_6^1, f_6^2)
\end{array}$$

Consider the assignment $\alpha = \{v_1 \mapsto t_1, v_2 \mapsto t_1, v_3 \mapsto t_2, v_4 \mapsto t_2, v_5 \mapsto t_3\} \cup \{v_6 \mapsto t_1, v_7 \mapsto t_2, v_8 \mapsto t_3, v_9 \mapsto t_4, v_{10} \mapsto t_5\}$. Within each variable component, the $\le$ ordering constraints are satisfied. Having determined the frequencies using the gcc constraints, we observe that the $\ge_{lex}$ constraints are satisfied, because $(2,1) \ge_{lex} (2,1) \ge_{lex} (1,1) \ge_{lex} (0,1) \ge_{lex} (0,1) \ge_{lex} (0,0)$. If student 10 moves from table 5 to table 6, producing a symmetrically equivalent assignment because the tables are fully interchangeable, the $\ge_{lex}$ constraints are no longer satisfied, because $(2,1) \ge_{lex} (2,1) \ge_{lex} (1,1) \ge_{lex} (0,1) \ge_{lex} (0,0) \not\ge_{lex} (0,1)$.

Theorem 1. *For every solution α to a piecewise symmetric CSP, there exists exactly one symmetric solution that obeys the static structural symmetry-breaking constraints.*

Proof. (a) We show that there exists at least one symmetric solution that obeys all the symmetry-breaking constraints. Denote by $\tau_\alpha^\ell : \{j(\ell), \dots, j(\ell+1) - 1\} \to \{j(\ell), \dots, j(\ell+1) - 1\}$ the function that ranks the values in D_ℓ according to the signatures over some solution α, i.e., $\mathrm{sig}_\alpha(d_{\tau_\alpha^\ell(h)}) \ge \mathrm{sig}_\alpha(d_{\tau_\alpha^\ell(h+1)})$ for all $j(\ell) \le h < j(\ell+1) - 1$. We obtain a symmetric solution β where we re-order the values in each D_ℓ according to τ_α^ℓ. Then, when we denote by $\sigma_\beta^k : \{i(k), \dots, i(k+1) - 1\} \to \{i(k), \dots, i(k+1) - 1\}$ the function that ranks the variables in V_k according to β, i.e., $\beta(v_{\sigma_\beta^k(h)}) \le \beta(v_{\sigma_\beta^k(h+1)})$ for all $i(k) \le h < i(k+1) - 1$, we can re-order the variables in each V_k according to σ_β^k, and we get a new symmetric solution γ. Note that the re-ordering of the variables within each component has no effect on the signatures of the values, i.e., $\mathrm{sig}_\gamma(d) = \mathrm{sig}_\beta(d)$ for all $d \in D$. Thus, γ obeys all the symmetry-breaking constraints.

(b) Now assume there are two symmetric solutions α and β to the piecewise symmetric CSP that both obey all the symmetry-breaking constraints. Denote by τ^ℓ the re-ordering of the values in D_ℓ and denote by σ^k the re-ordering of the variables in V_k. Then, we denote by τ the piecewise bijection over the values based on the τ^ℓ, and by σ

the piecewise bijection over the variables based on the σ^k, such that $\alpha = \tau \circ \beta \circ \sigma$. The first thing to note is that the application of the piecewise bijection σ on the variables has no effect on the signatures of the values, i.e., $\mathrm{sig}_\beta(d) = \mathrm{sig}_{\beta\circ\sigma}(d)$ for all $d \in D$. Consequently, the total lexicographic ordering constraints on the signatures of each value d and its image $\tau(d)$ require that $\mathrm{sig}_\beta(d) = \mathrm{sig}_\beta(\tau(d)) = \mathrm{sig}_{\tau\circ\beta}(d) = \mathrm{sig}_{\tau\circ\beta\circ\sigma}(d) = \mathrm{sig}_\alpha(d)$. Thus, the signatures under α and β are identical. However, with the signatures of all the values fixed and with the ordering on the variables, there exists exactly one assignment that gives these signatures, so α and β must be identical. □

4 Static Versus Dynamic SSB for Piecewise Symmetric CSPs

The advantage of a static symmetry-breaking method lies mainly in its ease of use and its moderate costs per search node. The number of constraints added is *linear* in the size of the problem, unlike the general method in [5], but they may break super-exponentially many variable and value symmetries. Constraint propagation and incrementality are inherited from the existing lex-ordering and gcc constraints. However, it is well-known that static symmetry breaking can collide with dynamic variable and value orderings, whereas dynamic methods such as SBDD do not suffer from this drawback.

Theorem 2. *Given static variable and value orderings, static (SSSB) and dynamic SSB (DSSB) explore identical search trees for piecewise symmetric CSPs.*

Proof. (a) Proof by contradiction. Assume there exists a node in the SSSB search tree that is pruned by DSSB. Without loss of generality, we may consider the first node in a depth-first search tree where this occurs. We identify this node with the assignment $\beta := \{v_1, \dots, v_t\} \to D$, and the node that dominates β is identified with the assignment $\alpha := \{v_1, \dots, v_s\} \to D$, for some $1 \leq s \leq t \leq n$. By the definition of DSSB, we have that $\alpha(v_i) = \beta(v_i)$ for all $1 \leq i < s$ (since every no-good considered by SBDD differs in exactly its last variable assignment from the current search node), and $\alpha(v_s) < \beta(v_s)$.

First consider $s = t$. Assume the dominance check between α and β is successful. Then, $\mathrm{sig}_\beta(\beta(v_s)) = \mathrm{sig}_\alpha(\alpha(v_s)) \gneq \mathrm{sig}_\beta(\alpha(v_s))$. However, since $\alpha(v_s) < \beta(v_s)$, it must also hold that $\mathrm{sig}_\beta(\alpha(v_s)) \geq \mathrm{sig}_\beta(\beta(v_s))$. Contradiction.

Now consider $s < t$. Since the parent of β is not dominated by α, as β was chosen minimally, we know that v_t must be interchangeable with some v_p with $p \leq s < t$. If we denote the component of v_t by $\{v_q, \dots, v_t, \dots, v_u\}$, we can deduce that $q \leq s < t \leq u$, i.e., v_s and v_t must belong to the same component. By definition of SSSB, we also know that $\alpha(v_q) \leq \cdots \leq \alpha(v_s) < \beta(v_s) \leq \cdots \leq \beta(v_t)$. Moreover, we know that $\beta(v_t)$ and $\alpha(v_p)$ must be interchangeable. Consequently, $\alpha(v_s)$ and $\beta(v_s)$ are also interchangeable. Now, since setting $\alpha(v_s)$ and $\beta(v_s)$ to v_s was not considered symmetric by DSSB, together with $\beta(v_s) > \alpha(v_s)$, we know that $\mathrm{sig}_\beta(\alpha(v_s)) \lneq \mathrm{sig}_\alpha(\alpha(v_s))$. It follows that $\mathrm{sig}_\alpha(\alpha(v_s)) \gneq \mathrm{sig}_\beta(\alpha(v_s)) \geq \mathrm{sig}_\beta(\beta(v_s))$ (1). When $\alpha(v_s)$ is matched with $\beta(v_i)$, for $q \leq i \leq t$, by the successful dominance check of α and β, then it must hold that $i < s$ as otherwise $\mathrm{sig}_\alpha(\alpha(v_s)) \leq \mathrm{sig}_\beta(\beta(v_i)) \leq \mathrm{sig}_\beta(\beta(v_s))$, which is in conflict with (1). This implies that $\beta(v_t)$ must be matched with some $\alpha(v_r)$ for $q \leq r < s$ by the successful dominance check. Hence all the values in $\{\alpha(v_r), \dots, \alpha(v_s), \beta(v_s), \dots, \beta(v_t)\}$ are pairwise interchangeable. But then $\mathrm{sig}_\alpha(\alpha(v_r)) \leq \mathrm{sig}_\beta(\beta(v_t)) \leq \mathrm{sig}_\beta(\beta(v_s)) \lneq \mathrm{sig}_\alpha(\alpha(v_s)) \leq \mathrm{sig}_\alpha(\alpha(v_r))$. Contradiction.

(b) Assume there exists a node in the DSSB search tree that is pruned by SSSB. Without loss of generality, we may consider the first node in a depth-first search tree where this occurs. We identify this node with the assignment $\beta := \{v_1, \dots, v_t\} \rightarrow D$.

First assume a variable ordering constraint is violated, i.e., $\beta(v_j) > \beta(v_i)$ for some $1 \leq i < j \leq t$ where v_i and v_j are interchangeable. Consider $\alpha : \{v_1, \dots, v_i\} \rightarrow D$ such that $\alpha(v_k) := \beta(v_k)$ for all $1 \leq k < i$, and $\alpha(v_i) := \beta(v_j)$. Then, due to the static variable and value orderings, α is a node that has been fully explored before β, and α dominates β, which is clear by mapping v_i to v_j. Thus, β is also pruned by DSSB.

Now assume a lex-ordering constraint on the value signatures is violated. Denote the interchangeable values by d_i and d_j, with $1 \leq i < j$. Since β was chosen minimally, when we denote the variable component that shows that $\mathrm{sig}_\beta(d_i) < \mathrm{sig}_\beta(d_j)$ by V_k, we know that $\mathrm{sig}_\beta(d_i)[\ell] = \mathrm{sig}_\beta(d_j)[\ell]$ for all $\ell < k$ and $\mathrm{sig}_\beta(d_i)[k] + 1 = \mathrm{sig}_\beta(d_j)[k]$. With $s := \max\{p \mid p < t \;\; \& \;\; \beta(v_p) = d_i\}$, we set $\alpha : \{v_1, \dots, v_{s+1}\} \rightarrow D$ with $\alpha(v_r) := \beta(v_r)$ for all $r \leq s$ and $\alpha(v_{s+1} := d_i)$. Again, due to the static variable and value orderings, α is a node that has been fully explored before β, and α dominates β, which is clear simply by mapping d_i to d_j. Hence, β is also pruned by DSSB. □

We conclude that dynamic symmetry breaking draws its strength from its ability to accommodate dynamic variable and value orderings, but causes an unnecessary overhead when these orderings are fixed. In this case, static symmetry breaking offers a much more light-weight method that achieves exactly the same symmetry-breaking effectiveness for piecewise symmetric CSPs. Can we find general conditions under which a static symmetry-breaking method leads to symmetry-free search trees?

Acknowledgements. The authors are partly supported by grant IG2001-67 of STINT, the Swedish Foundation for Int'l Cooperation in Research and Higher Education.

References

1. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In T. Walsh, editor, *Proceedings of CP'01*, volume 2293 of *LNCS*, pages 93–107. Springer-Verlag, 2001.
2. F. Focacci and M. Milano. Global cut framework for removing symmetries. In T. Walsh, editor, *Proceedings of CP'01*, volume 2239 of *LNCS*, pages 77–92. Springer-Verlag, 2001.
3. I. P. Gent and B. M. Smith. Symmetry breaking during search in constraint programming. In *Proceedings of ECAI'00*, pages 599–603, 2000.
4. J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of ISMIS'93*, volume 689 of *LNAI*, pages 350–361. Springer-Verlag, 1993.
5. J.-F. Puget. An efficient way of breaking value symmetries. In *Proceedings of AAAI'06*. AAAI Press, 2006.
6. J.-C. Régin. Generalized arc-consistency for global cardinality constraint. In *Proceedings of AAAI'96*, pages 209–215. AAAI Press, 1996.
7. C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In *Proceedings of ECAI'04*, pages 211–215. IOS Press, 2004.
8. M. Sellmann and P. Van Hentenryck. Structural symmetry breaking. In *Proceedings of IJCAI'05*. Morgan Kaufmann, 2005.
9. P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In *Proceedings of IJCAI'03*, Morgan Kaufmann, 2003.

The Modelling Language Zinc

Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace

Clayton School of IT, Monash University, Australia
{mbanda, marriott, reza.rafeh, wallace}@mail.csse.monash.edu.au

Abstract. We describe the Zinc modelling language. Zinc provides set constraints, user defined types, constrained types, and polymorphic predicates and functions. The last allows Zinc to be readily extended to different application domains by user-defined libraries. Zinc is designed to support a modelling methodology in which the same conceptual model can be automatically mapped into different design models, thus allowing modellers to easily "plug and play" with different solving techniques and so choose the most appropriate for that problem.

1 Introduction

Solving combinatorial problems is a remarkably difficult task which requires the problem to be precisely formulated and efficiently solved. Even formulating the problem precisely is surprisingly difficult and typically requires many cycles of formulation and solving. Efficiently solving it often requires development of tailored algorithms which exploit the structure of the problem, and extensive experimentation to determine which technique or combination of techniques is most appropriate for a particular problem. Reflecting this discussion, modern approaches to solving combinatorial problems divide the task into two (hopefully simpler) steps. The first step is to develop the *conceptual model* of the problem which specifies the problem without consideration as to how to actually solve it. The second step is to *solve* the problem by mapping the conceptual model into an executable program called the *design model*. Ideally, the same conceptual model can be transformed into different design models, thus allowing modellers to easily "plug and play" with different solving techniques [4].

Here we introduce a new modelling language, Zinc, specifically designed to support this methodology. There has been a considerable body of research into problem modelling which has resulted in a progression of modelling languages including AMPL [2], Localizer [6], OPL [7], and specification languages including ESRA [1] and ESSENCE [3]. We gladly acknowledge the strong influence that OPL has had on our design. Our reasons to develop yet another modelling language are threefold.

First, we want the modelling language to be solver and technique independent, allowing the same conceptual model to be mapped to different solving techniques and solvers, i.e., be mapped to design models that use the most appropriate technique, be it local search, mathematical modelling (MIP), Constraint Programming (CP), or a combination of the above. To date the implemented languages have been tied to specific underlying platforms or solving technologies. For example, AMPL is designed to interface to MIP packages such as Cplex and Xpress-MP, Localizer was designed to map down to a local search engine, and ESSENCE and ESRA are designed for CP techniques. Of the above, only OPL was designed to combine the strengths of both MIP and CP, and now the most recent version of OPL only supports MIP.

Second, we want to provide high-level modelling features but still ensure that the Zinc models can be refined into practical design models. Zinc offers structured types, sets, and user defined predicates and functions which allow a Zinc model to be encapsulated as a predicate. It also allows users to define "constrained objects" i.e., to associate constraints to a particular type thus specifying the common characteristics that a class of items are expected to have [5]. It supports polymorphism, overloading and type coercion which make the language comfortable and natural to use. However, sets must be finite, and recursion is restricted to iteration so as to ensure that execution of Zinc programs is guaranteed to terminate. Zinc is more programming language like than the specification based approaches of ESSENCE and ESRA. These provide a more abstract kind of modeling based on first-order relations. Currently, they do not support variables with continuous domains or, as far as we can tell, functions or predicates. Furthermore, only limited user-defined types are provided.

And third, we want Zinc to have a simple, concise core but allow it to be extended to different application areas. This is achieved by allowing Zinc users to define their own application specific library predicates, functions and types. This contrasts with, say, OPL which provides seemingly ad-hoc built-in types and predicates for resource allocation and cannot be extended to model new application areas without redefining OPL itself since it does not allow user-defined predicates and functions.

2 Zinc

Zinc is a first-order functional language with simple, declarative semantics. It provides: mathematical notation-like syntax; expressive constraints (finite domain and integer, set and linear arithmetic); separation of data from model; high-level data structures and data encapsulation including constrained types; user defined functions and constraints.

As an example of Zinc, consider the model in Figure 1 for the perfect squares problem [8]. This consists of a base square of size `sizeBase` (6 in the figure) and a list of squares of various sizes `squares` (three of size 3, one of size 2 and five of size 1 in the figure). The aim is to place all squares into the base without overlapping each other.

The model defines a constrained type `PosInt` as a positive integer and declares the parameter `sizeBase` to be of this type. A record type `Square` is used to model each of the squares. It has three fields `x`, `y` and `size` where (x, y) is the (unknown) position of the lower left corner of the square and `size` is the size of its sides. The first constraint in the model ensures each square is inside the base (note

```
type PosInt = (int:x where x>0);
PosInt: sizeBase;
record Square=(var 1..sizeBase: x, y; PosInt: size);
list of Square:squares;

constraint  forall(s in squares)
               s.x + s.size =< sizeBase+1 /\
               s.y + s.size =< sizeBase+1;
predicate nonOverlap(Square: s,t) =
               s.x+s.size =< t.x  \/  t.x+t.size =< s.x \/
               s.y+s.size =< t.y  \/  t.y+t.size =< s.y;
constraint forall(i,j in 1..length(squares) where i<j)
               nonOverlap(squares[i], squares[j]);
predicate onRow(Square:s, int: r) =
               s.x =< r /\ r < s.x + s.size;
predicate onCol(Square:s, int: c) =
               s.y =< c /\ c < s.y + s.size;
assert sum(s in squares) (s.size^2) == sizeBase^2;
constraint  forall(p in 1..sizeBase)
               sum(s in Squares) (s.size*holds(onRow(s,p))) == sizeBase /\
               sum(s in Squares) (s.size*holds(onCol(s,p))) == sizeBase;

output(squares);
```

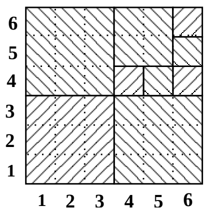


Fig. 1. Perfect Squares model

that \/ and /\ denote disjunction and conjunction, respectively). The model contains three user-defined predicates: `nonOverlap` which ensures two squares do not overlap, while `onRow` and `onCol` ensure the square is, respectively, on a particular row or column in the base.

The squares provided as input data are assumed to be such that they fit in the base exactly. To check this assumption, the model includes an assertion that equates their total areas.

The last constraint in the model is redundant since it is derived from the assumption that the squares exactly fill the base: the constraint simply enforces each row and column in the base to be completely full.

Data for the model can be given in a separate data file as, for example:

```
sizeBase=6;
squares = [ (x:_,y:_,size:s) | s in [3,3,3,2,1,1,1,1,1]];
```

Let us now look at the more interesting features of Zinc.

Types: Zinc provides a rich variety of types: integers, floats, strings (for output), Booleans, arrays, sets, lists, tuples, records, discriminated union (i.e. variant records) and enumerated types. All types have a total order on their elements, thus facilitating the specification of symmetry breaking and of polymorphic predicates and functions. In the case of compound types this total order is the natural lexicographic ordering based on their component types.

A useful feature of Zinc is that arrays are not restricted to integer indexes, they are actually maps from virtually any type to any other type. Similarly, Sets can be of any data type as long as they are finite.

In order to allow natural mathematical-like notation, Zinc provides automatic coercion from integers to floats, from sets to lists, from lists to arrays and from tuples to records with the same field types. A set is coerced to a list with its elements in increasing order, and a list of length n is coerced to an array with index set $1..n$.

One of the novel features of Zinc is that types, such as `PosInt` in the perfect squares model in Figure 1, can have an associated constraint on elements of that type. Effectively, whenever a variable is declared to be of constrained type, the constraint is implicitly placed in the model.

Variables: All variables must be declared with a type except for local variables occurring in an array, list or set comprehension. The reason for requiring explicit typing is that automatic coercion and separate datafiles precludes complete type inference.

Variables have an associated *instantiation* which indicates whether they are *parameters* of the model whose value is known before performing any solving of the model, or *decision variables* whose value is known only after. Variables are, by default, assumed to be parameters, and are declared to be decision variables by adding the `var` keyword before their type definition. This keyword can only be applied to integers, floats, enumerated types, Booleans, and sets. Lists of variable length are not allowed.

Expressions: Zinc provides all the standard mathematical functions and operators, many of which are overloaded to accept floats and integers.

List, set and array comprehensions provide the standard iteration constructs in Zinc. Examples of their use are shown in the preceding example. Other iterations such as `forall`, `exists`, `sum` and `product` are defined as Zinc library functions using $foldl(Fun, List, Init)$, which applies the binary function Fun to each element in $List$ (working left-to-right) with initial accumulator value set to $Init$. For instance,

```
constraint  forall(list of bool: L) = foldl('/\',L,true);
function int: sum(list of int: L) = foldl('+',L,0);
function float: sum(list of float: L) = foldl('+',L,0);
```

$foldl$ and $foldr$ are the only higher-order functions provided by Zinc. User-defined higher-order functions and predicates are not allowed in Zinc.

Constraints: Zinc supports the usual constraints over integers, floats, Booleans, sets and user defined enumerated type constants. All constraints, including user-defined constraints, are regarded as Boolean functions and can be combined using the standard Boolean operators. Higher-order constraints can also be readily defined, which is useful, for example, to define functions such as the built-in function *holds* which returns 1 if the constraint holds, and 0 otherwise.

The standard global constraints, such as `alldifferent`, are provided. Note that thanks to the existence of a total order on the elements of any type, the `alldifferent` global constraint works for lists of any type, including records.

User-defined Predicates and Functions: One of the most powerful features of Zinc is the ability for users to define their own predicates and functions, such as `nonOverlap`, `onRow` and `onCol`, in the perfect square model in Figure 1. Zinc supports polymorphic types and context-free overloading. Although the average modeller may not use these facilities, it allows standard modelling functions to be defined in Zinc itself. We have previously seen an example of how the library function `sum` is overloaded to take either a list of integers or a list of floats, and how the library function `alldifferent` is polymorphically defined for lists of any type. As another example of polymorphism, consider the polymorphic predicate `between` (with polymorphic types being indicated by `$T`):

```
predicate between($T: x,y,z) =
        (x =< y /\ y =< z) \/ (z=<y /\ y=<x);
```

which applies to numeric and non-numeric types, lists, tuples, records and sets!

User-defined functions and predicates are instantiation-overloaded in the sense that a definition can take both parameters and decision variables.

3 Conclusion and Future Work

We have presented a new modelling language Zinc designed to allow natural, high-level specification of a conceptual model. Unlike most other modelling languages, Zinc provides set constraints, constrained types, user defined types, and polymorphic predicates and functions. The last allows Zinc to be readily extended to different application domains by user-defined libraries.

One of the main aims of developing Zinc is that a Zinc model can be mapped into design models that utilize different solving techniques such as local search or tree-search with propagation based solvers. Currently, we are implementing three mapping modules to map the Zinc models into design models in ECLiPSe for three different solving techniques: constraint programming, local search and mathematical methods.

References

1. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR*, pages 214–232, 2003.
2. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2002.
3. A.M. Frisch, M. Grum, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The essence of ESSENCE: A constraint language for specifying combinatorial problems. In *Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 73–88, 2005.

4. C. Gervet. *Large scale combinatorial optimization: A methodological viewpoint*, volume 57 of *Discrete Mathematics and Theoretical Computer Science*, pages 151–175. DIMACS, 2001.
5. B. Jayaraman and P. Tambay. Modeling engineering structures with constrained objects. In *PADL*, pages 28–46, 2002.
6. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Proc. Principles and Practice of Constraint Programming - CP97*, pages 237–251, 1997.
7. P. Van Hentenryck, I. Lustig, L.A. Michel, and J.-F. Puget. *The OPL Optimization Programming Language.* MIT Press, 1999.
8. E. W. Weisstein. Perfect square dissection. From MathWorld –A Wolfram Web Resource, http://mathworld.wolfram.com/PerfectSquareDissection.html, 1999.

A Filter for the Circuit Constraint

Latife Genç Kaya and J.N. Hooker

Tepper School of Business, Carnegie Mellon University, Pittsburgh, USA
lgenc@andrew.cmu.edu, john@hooker.tepper.cmu.edu

Abstract. We present an incomplete filtering algorithm for the circuit constraint. The filter removes redundant values by eliminating nonhamiltonian edges from the associated graph. We identify nonhamiltonian edges by analyzing a smaller graph with labeled edges that is defined on a separator of the original graph. The complexity of the procedure for each separator S is approximately $O(|S|^5)$. We found that it identified all infeasible instances and eliminated about one-third of the redundant domain elements in feasible instances.

The circuit constraint can be written

$$\text{circuit}(x_1, \ldots, x_n)$$

where the domain of each x_i is $D_i \subset \{1, \ldots, n\}$. The constraint requires that $y_1, \ldots, y_n$ be a cyclic permutation of $1, \ldots, n$, where

$$\begin{aligned} & y_{i+1} = x_{y_i}, \quad i = 1, \ldots, n-1 \\ & y_1 = x_{y_n} \end{aligned}$$

Let directed graph G contain an edge (i, j) if and only if j belongs to the domain of x_i. If edge (i, j) is selected when $x_i = j$, the circuit constraint requires that the selected edges form a hamiltonian circuit or *tour* of G.

One approach to filtering the circuit constraint is to make use of necessary conditions for hamiltonicity of G. Chvátal [2, 3] analyzes several conditions, one of which (*1-toughness*) is a very restricted case of a condition we develop for filtering nonhamiltonian edges.

Some elementary techniques for filtering domains after a partial tour has been constructed are described by Shufelt and Berliner [4], whose analysis relies on the special structure of a chessboard problem, and Caseau and Laburthe [1], who solve small traveling salesman problems. Neither approach is intended for the general filtering problem in which arbitrary variable domains are given.

1 Separator Graph

Given a graph $G = (V, E)$, a set of vertices $S \subset V$ is a (vertex) *separator* of G if $V \setminus S$ induces a subgraph $\bar{G}_S$ of G with at least two connected components $C_1, \ldots, C_p$. The *separator graph* G_S for a separator S of G consists of a directed

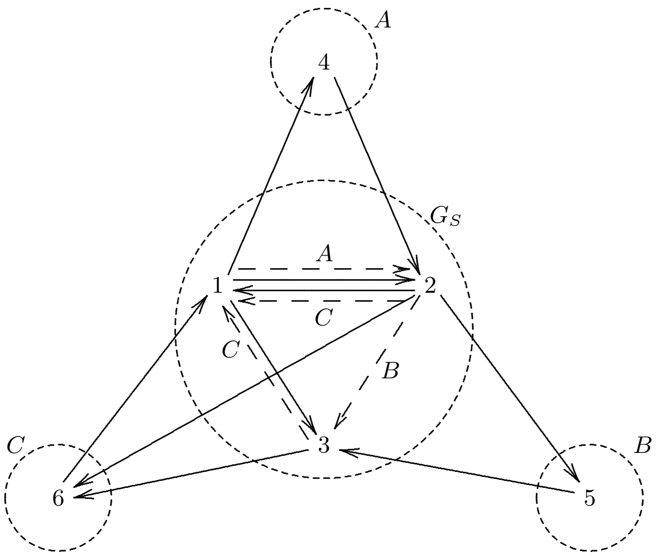


Fig. 1. Graph G on vertices $\{1, \ldots, 6\}$ contains the solid edges, and the separator graph G_S on $S = \{1, 2, 3\}$ contains the solid (unlabeled) edges and dashed (labeled) edges within the larger circle. The small circles surround connected components of the separated graph.

graph with vertex set S and edge set E_S, along with a set L_S of *labels* corresponding to the connected components of $\bar{G}_S$. E_S contains (a) an *unlabeled* edge (i, j) for each $(i, j) \in E$, as well as (b) a *labeled* edge $(i, j)^C$ whenever $C \in L_S$ and $(i, c_1), (c_2, j) \in E$ for some pair of vertices c_1, c_2 in connected component C (possibly $c_1 = c_2$ and c_1 and c_2 need not be connected by an edge).

Consider for example the graph G of Fig. 1. Vertex set $S = \{1, 2, 3\}$ separates G into three connected components that may be labeled A, B and C, each of which contains only one vertex. Thus $L_S = \{A, B, C\}$, and the separator graph G_S contains the three edges that connect its vertices in G plus four labeled edges. For example, there is an edge $(1, 2)$ labeled A, which can be denoted $(1, 2)^A$, because there is a directed path from some vertex in component A through $(1, 2)$ and back to a vertex of component A.

A hamiltonian cycle of G_S is *permissible* if it contains at least one edge bearing each label in L_S. An edge of G_S is *permissible* if it is part of some permissible hamiltonian cycle of G_S. Thus the edges $(1, 2)^A$, $(2, 3)^B$ and $(3, 1)^C$ form a permissible hamiltonian cycle in Fig. 1, and they are the only permissible edges.

Theorem 1. *If S is a separator of directed graph G, then G is hamiltonian only if G_S contains a permissible hamiltonian cycle. Furthermore, an edge of G connecting vertices in S is hamiltonian only if it is a permissible edge of G_S.*

Proof. Consider an arbitrary hamiltonian cycle H of G. We can construct a permissible hamiltonian cycle H_S for G_S as follows. Consider the sequence of vertices in H and remove those that are not in S; let $i_1, \ldots, i_m, i_1$ be the remaining sequence of vertices. H_S can be constructed on these vertices as follows. For any pair i_k, i_{k+1} (where i_{m+1} is identified with i_1), if they are adjacent in H then (i_k, i_{k+1}) is an unlabeled edge of G_S and connects i_k and i_{k+1} in H_S. If i_k, i_{k+1} are not adjacent in H then all vertices in H between i_k and i_{k+1} lie in the same connected component C of the subgraph of G induced by $V \setminus S$. This means (i_k, i_{k+1}) is an edge of G_S with label C, and $(i_k, i_{k+1})^C$ connects i_k and i_{k+1} in H_S. Since H passes through all connected components, every label must occur on some edge of H_S, and H_S is permissible.

We now show that if (i, j) with $i, j \in S$ is an edge of a hamiltonian cycle H of G, then (i, j) is an edge of a permissible hamiltonian cycle of G_S. But in this case (i, j) is an unlabeled edge of G_S, and by the above construction (i, j) is part of H_S.

Corollary 1. *If $|L_S| > |S|$ for some separator S, then G is nonhamiltonian.*

Proof. The separator graph G_S has $|S|$ vertices and therefore cannot have a hamiltonian cycle with more than $|S|$ edges.

If $|L_S| \leq |S|$ for all separators S, G is *1-tough*, adapting Chvátal's term [3] to directed graphs.

Corollary 2. *If $|L_S| = |S|$ for some separator S, then no edge connecting vertices of S is hamiltonian.*

Proof. An edge e that connects vertices in S is unlabeled in G_S. If e is hamiltonian, some hamiltonian cycle in G_S that contains e must have at least $|S|$ labeled edges. But since the cycle must have exactly $|S|$ edges, all the edges must be labeled and none can be identical to e.

2 Finding Separators

We use a straightforward breadth-first-search heuristic to find separators of G. We arrange the vertices of G in levels as follows. Arbitrarily select a vertex i of G as a *seed* and let level 0 contain i alone. Let level 1 contain all neighbors of i in G. Let level k (for $k \geq 2$) contain all vertices j of G such that (a) j is a neighbor of some vertex on level $k-1$, and (b) j does not occur in levels 0 through $k-1$. If maximum level $m \geq 2$, the vertices on any given level k $(0 < k < m)$ form a separator of G. Thus the heuristic yields $m-1$ separators.

The heuristic can be run several times as desired, each time beginning with a different vertex on level 0.

3 Cardinality Filter and Vertex Degree Filtering

The next step of the algorithm is to identify nonpermissible edges of G_S for each separator S by a relaxation of the permissibility condition.

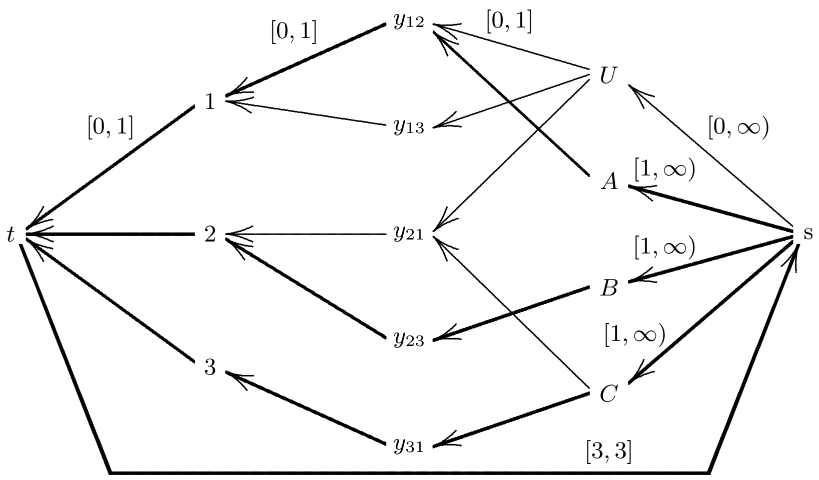


Fig. 2. Flow model for simultaneous gcc and out-degree filtering of nonhamiltonian edges. Heavy lines show the only feasible flow.

Let Y contain a variable y_{ij} for each *ordered* pair of vertices i, j in G_S. The domain of y_{ij} contains label C for each edge $(i,j)^C$ in G_S and the element U if unlabeled edge (i,j) is in G_S. A permissible hamiltonian cycle must satisfy the constraint

$$\text{gcc}(Y, (C_1, \ldots, C_p, U), (1, \ldots, 1, 0), (\infty, \ldots, \infty)) \tag{1}$$

Vertex degree filtering is based on the fact that the in-degree and out-degree of every vertex in a hamiltonian cycle is one. Constraint (1) can be combined with out-degree filtering by constructing a capacitated flow graph G_S^{out} with the following vertices

source s and sink t
U and $C_1, \ldots, C_p$
a vertex for each $y_{ij} \in Y$
a vertex for every vertex of G_S

and the following directed edges

(s, C_i) with capacity range $[1, \infty)$ for $i = 1, \ldots, p$
(s, U) with capacity range $[0, \infty)$
(C, y_{ij}) with capacity range $[0, 1]$ for every edge $(i,j)^C \in E_S$
(U, y_{ij}) with capacity range $[0, 1]$ for every unlabeled edge $(i,j) \in E_S$
(y_{ij}, i) with capacity range $[0, 1]$ for every ordered pair (i,j) such that
    (i,j) or $(i,j)^C$ belongs to E_S
(i, t) with capapcity range $[0, 1]$ for every vertex i of G_S
return edge (t, s) with capacity range $[|S|, |S|]$

The gcc and out-degree constraints are simultaneously satisfiable if and only if G_S^{out} has a feasible flow. The same is true of the graph G_S^{in} constructed in an analogous way to enforce in-degree constraints. Thus

```
Let $G$ be the directed graph associated with circuit$(x_1, \ldots, x_n)$.
Let $D_i$ be the current domain of $x_i$ for $i = 1, \ldots, n$.
Let $s$ be a limit on the size of separators considered.
For one or more vertices $i$ of $G$:
    Use the breadth-first-search heuristic to create a collection $\mathcal{S}$ of separators,
        with $i$ as the seed.
    For each $S \in \mathcal{S}$ with $|S| \leq s$:
        For $G'_S = G^{out}_S, G^{in}_S$:
            If $G'_S$ has a feasible flow $f$ then:
                For each edge $(U, y_{ij})$ of $G'_S$ on which $f$ places zero flow:
                    If there is no augmenting path from $y_{ij}$ to $U$ then delete $j$ from $D_i$.
            Else stop; circuit$(x_1, \ldots, x_n)$ is infeasible.
```

Fig. 3. Filtering algorithm for the circuit constraint

Theorem 2. *An edge (i, j) of G is nonhamiltonian if there is a separator S of G for which the maximum flow on arc (U, y_{ij}) of either G^{out}_S or G^{in}_S is zero.*

This can be checked by first computing a feasible flow f on G^{out}_S and on G^{in}_S. The maximum flow on (U, y_{ij}) is zero if (a) f places zero flow on (U, y_{ij}), and (b) there is no augmenting path from y_{ij} to U.

For example, the network G^{out}_S for the graph G and separator S of Fig. 1 is shown in Fig. 2. Since the flow of zero on edges (U, y_{12}), (U, y_{13}) and (U, y_{21}) is maximum in each case, the three edges $(1, 2)$, $(1, 3)$, and $(2, 1)$ are nonhamiltonian.

4 The Algorithm and Computational Results

The filtering algorithm (Fig. 3) has complexity of approximately $O(|S|^5)$ for each seoparator S. In preliminary computational tests on several thousand random graphs with up to 15 vertices, we detected all nonhamiltonian graphs and eliminated about 1/3 of nonhamiltonian edges in hamiltonian graphs.

References

1. Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In L. Naish, editor, *Proceedings, Fourteenth International Conference on Logic Programming (ICLP 1997)*, volume 2833, pages 316–330. The MIT Press, 1997.
2. V. Chvátal. Edmonds polytopes and weakly hamiltonian graphs. *Mathematical Programming*, 5:29–40, 1973.
3. V. Chvátal. Tough graphs and hamiltonian circuits. *Discrete Mathematics*, 5:215–228, 1973.
4. J. Shufelt and H. Berliner. Generating hamiltonian circuits without backtracking from errors. *Theoretical Computer Science*, 132:347–375, 1994.

A New Algorithm for Sampling CSP Solutions Uniformly at Random

Vibhav Gogate and Rina Dechter

Donald Bren School of Information and Computer Science
University of California, Irvine, CA 92697
{vgogate, dechter}@ics.uci.edu

Abstract. The paper presents a method for generating solutions of a constraint satisfaction problem (CSP) uniformly at random. Our method relies on expressing the constraint network as a uniform probability distribution over its solutions and then sampling from the distribution using state-of-the-art probabilistic sampling schemes. To speed up the rate at which random solutions are generated, we augment our sampling schemes with pruning techniques used successfully in constraint satisfaction search algorithms such as conflict-directed back-jumping and no-good learning.

1 Introduction

The paper presents a method for generating solutions to a constraint network uniformly at random. The idea is to express the uniform distribution over the set of solutions as a probability distribution and then generate samples from this distribution using monte-carlo sampling. We develop monte-carlo sampling algorithms that extend our previous work on monte-carlo sampling algorithms for probabilistic networks [4] in which the output of generalized belief propagation is used for sampling.

Our experiments reveal that pure sampling schemes, even if quite advanced [4], may fail to output even a single solution for constraint networks that have few solutions. So we propose to enhance sampling with search techniques that aim at finding a consistent solution fast, such as conflict directed back-jumping and no-good learning.

We demonstrate empirically the performance of our search+sampling schemes by comparing them with two previous schemes: (a) the WALKSAT algorithm [6] and (b) the mini-bucket approximation [2]. Our work is motivated by a real-world application of generating test programs in the field of functional verification (see [2] for details).

2 Preliminaries

Definition 1 (constraint network). *A constraint network (CN) is defined by a 3-tuple,* $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$*, where* $\mathbf{X}$ *is a set of variables* $\mathbf{X} = \{X_1, \ldots, X_n\}$*, associated with a set of discrete-valued domains,* $\mathbf{D} = \{\mathbf{D_1}, \ldots, \mathbf{D_n}\}$*, and a set of constraints* $\mathbf{C} = \{C_1, \ldots, C_r\}$*. Each constraint* C_i *is a pair* $(\mathbf{S_i}, \mathbf{R_i})$*, where* $\mathbf{R_i}$ *is a relation* $\mathbf{R_i} \subseteq \mathbf{D_{S_i}}$ *defined on a subset of variables* $\mathbf{S_i} \subseteq \mathbf{X}$*.* $\mathbf{R_i}$ *contains the allowed tuples of* C_i*. A solution is an assignment of values to variables* $\mathbf{x} = (X_1 = x_1, \ldots, X_n = x_n)$*,* $X_i \in \mathbf{D}_i$*, such that* $\forall\, C_i \in \mathbf{C}$*,* $\mathbf{x_{S_i}} \in \mathbf{R_i}$*.*

Definition 2 (Random Solution Generation Task). *Let* **sol** *be the set of solutions to a constraint network* $\mathcal{R} = (\mathbf{X}, \mathbf{D}, \mathbf{C})$. *We define a uniform probability distribution* $P_u(\mathbf{x})$ *relative to* $\mathcal{R}$ *such that for every assignment* $\mathbf{x} = (X_1 = x_1, \ldots, X_n = x_n)$ *to all the variables that is a solution, we have* $P_u(\mathbf{x} \in \mathbf{sol}) = \frac{1}{|\mathbf{sol}|}$ *while for non-solutions we have* $P_u(\mathbf{x} \notin \mathbf{sol}) = 0$. *The task of random solution generation is to generate positive tuples from this distribution uniformly at random.*

3 Generating Solutions Uniformly at Random

In this section, we describe how to generate random solutions using monte-carlo (MC) sampling. We first express the constraint network $\mathcal{R}(\mathbf{X}, \mathbf{D}, \mathbf{C})$ as a uniform probability distribution $\mathcal{P}$ over the space of solutions: $\mathcal{P}(\mathbf{X}) = \alpha \prod_i C_i(\mathbf{S_i} = \mathbf{s_i})$. Here, $C_i(\mathbf{s_i}) = 1$ if $\mathbf{s_i} \in R_i$ and 0 otherwise. $\alpha = 1/\sum \prod_i f_i(\mathbf{S_i})$ is the normalization constant. Clearly, any algorithm that samples tuples from $\mathcal{P}$ accomplishes the solution generation task. This allows us to use the following monte-carlo (MC) sampler to sample from $\mathcal{P}$.

```
Algorithm Monte-Carlo Sampling
Input: A factored distribution $\mathcal{P}$ and a time-bound ,Output: A collection of samples from $\mathcal{P}$.
Repeat until the time-bound expires

 1. FOR j = 1 to n
     (a) Sample $X_j = x_j$ from $P(X_j|X_1 = x_1, \ldots, X_{j-1} = x_{j-1})$
 2.  End FOR
 3.  If $x_1, \ldots, x_n$ is a solution output it.
```

Hence forth, we will use P to denote the conditional distribution $P(X_j|X_1, \ldots, X_{j-1})$ (which is derived from $\mathcal{P}$). In [2], a method is presented to compute the conditional distributions P from $\mathcal{P}$ in time exponential in tree-width. But the tree-width is usually large for real-world networks and so we have to use approximations.

4 Approximating *P* Using Iterative Join Graph Propagation

Because exact methods for computing the conditional probabilities P are impractical when the tree-width is large, we consider a generalized belief propagation algorithm called Iterative Join Graph Propagation (IJGP) [3] to compute an approximation to P. IJGP is a belief-propagation algorithm that takes a factored probability distribution $\mathcal{P}$ and a partial assignment $\mathbf{E} = \mathbf{e}$ as input. It then performs message passing on a special structure called the join-graph. The output of IJGP is a collection of functions which can be used to approximate $\mathcal{P}(X_j|\mathbf{e})$ for each variable X_j of $\mathcal{P}$. If the number of variables in each cluster is bounded by i (called the i-bound), we refer to IJGP as IJGP(i). The time and space complexity of one iteration of IJGP(i) is bounded exponentially by i.

IJGP(i) can be used to compute an approximation Q of P by executing it with $\mathcal{P}$ and the partial assignment $X_1 = x_1, \ldots, X_{j-1} = x_{j-1}$ as input and then using Q instead of P in step 1(a) of algorithm monte-carlo sampling. In this case, IJGP(i) should be executed n times, one for each instantiation of variable X_j to generate one full sample.

This process may be slow because the complexity of generating N samples in this way is $O(nNexp(i))$. To speed-up the sampling process, in [4] we pre-computed the approximation of P by executing IJGP(i) just once, yielding a complexity of $O(nN + exp(i))$.

Therefore, in order to be able to have a flexible control between the two extremes of using IJGP(i) just once, prior to sampling, versus using IJGP(i) at each variable instantiation, we introduce a control parameter p, measured as a percentage, which allows executing IJGP(i) every p % of the possible n variable instantiations. We call the resulting technique IJGP(i,p)-sampling.

4.1 Rejection of Samples

It is important to note that when all P's are exact in the algorithm monte-carlo sampling, all samples generated are guaranteed to be solutions to the constraint network. However, when we approximate P using IJGP such guarantees do not exist and our scheme will attempt to generate samples that are not consistent and therefore need to be rejected. Since the amount of rejection in IJGP(i,p)-sampling can be quite high, we equipped the basic IJGP(i,p)-sampling scheme with pruning algorithms common in search schemes for solving constraint problems. We describe these schemes in the next section.

5 Backjumping and No-Good Learning to Improve IJGP(i,p)-Sampling

Traditional sampling algorithms start sampling anew from the first variable in the ordering when an inconsistent assignment (sample) is generated. Instead, the algorithms can backtrack to the previous variable and sample a new value for the previous variable as is common in search algorithms. In other words, we could perform backtracking search instead of pure sampling. Before we sample a new value for the previous variable, we can update our sampling probability to reflect the discovery of the rejected sample. Also, instead of using naive backtracking we can use a more advanced approach such as conflict-directed backjumping and no-good learning. In conflict-directed backjumping, the algorithm backtracks a few levels back, to a variable that can be relevant to the current variables, instead of the recent previous variable [1]. In no-good learning each time an inconsistent assignment (sample) is discovered, the algorithm adds the assignment as a constraint (no-good) to the original constraint network so that in subsequent calls to the search procedure, the same assignment is not sampled. We learn only those no-goods which are bounded by i (the i-bound of IJGP(i)) to maintain constant space.

Once a no-good bounded by i is discovered, we check if the scope of the no-good is included in a cluster of the join-graph. If it is, then we insert the no-good in the cluster and subsequent runs of IJGP utilize this no-good; thereby potentially improving its approximation. We refer to the algorithm resulting from adding back-jumping search and no-good learning to IJGP(i,p)-sampling as IJGP(i,p)-SampleSearch.

6 Experimental Evaluation

We experimented with 5 algorithms (a) IJGP(i,p)-sampling which does not perform search, (b) MBE(i)-sampling which uses mini-bucket-elimination instead of IJGP to

Table 1. Performance of IJGP(3,p)-sampling and MBE(3)-sampling on random binary CSPs

Problems (N,K,C,T)	Time	IJGP(3,p)-SampleSearch No learning				IJGP(3,p)-SampleSearch learning				MBE(3)-SampleSearch
		p=0	p=10	p=50	p=100	p=0	p=10	p=50	p=100	p=0
		KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S
100,4,350,4	1000s	0.0346 0.0074 82290	0.0319 0.0061 42398	0.0108 0.0028 19032	0.011 0.0017 11792	0.0403 0.0086 103690	0.0172 0.0048 37923	0.013 0.0026 25631	0.0053 0.0008 9872	0.134 0.073 93823
100,4,370,4	1000s	0.0249 0.0089 18894	0.0235 0.0062 17883	0.0267 0.0084 2983	0.0156 0.0037 1092	0.0167 0.0058 28346	0.0188 0.0061 14894	0.0143 0.0049 3329	0.0106 0.0019 1981	0.107 0.0332 33895

approximate P and does not perform backjumping and no-good learning, (c) IJGP(i,p)-SampleSearch as described in section 5, (d) MBE(i)-SampleSearch which incorporates backjumping in MBE(i)-sampling as described in section 5 and (e) WALKSAT (which only works on SAT instances). We experimented with randomly generated binary constraint networks and SAT benchmarks available from satlib.org. Detailed experiments are presented in the extended version of the paper [5]. Here, we describe results on 100-variable random CSP instances, on logistics benchmarks and on verification benchmarks. For each network, we compute the fraction of solutions that each variable-value pair participates in i.e. $P_e(X_i = x_i)$. Our sampling algorithms output a set of solution samples $\mathbf{S}$ from which we compute the approximate marginal distribution: $P_a(X_i = x_i) = \frac{N_{\mathbf{S}(x_i)}}{|\mathbf{S}|}$ where $N_{\mathbf{S}(x_i)}$ is the number of solutions in the set $\mathbf{S}$ with X_i assigned the value x_i. We then compare the exact distribution with the approximate distribution using two error measures (accuracy): (a) *Mean Square error* - the square of the difference between the approximate and the exact and (b) *KL distance* - $P_e(x_i) * log(P_e(x_i)/P_a(x_i))$ averaged over all values, all variables and all problems. We also report the number of solutions generated by each sampling technique.

100-variable random CSPs: We experimented with randomly generated 100-variable CSP instances with domain size and tightness of 4. Here, we had to stay with relatively small problems in order to compute the exact marginal for comparison. The time-bound used is indicated by the column *Time* in Table 1. The results are averaged over 100 instances. We used an i-bound of 3 in all experiments. Here, pure IJGP(i,p)-sampling and pure MBE(i)-sampling [2] which do not perform search did not generate any consistent samples (solutions) and so we report results on IJGP(i,p)-SampleSearch and MBE(i)-SampleSearch in Table 1. We observe that as we increase p, the accuracy of IJGP(i,p)-SampleSearch increases while the number of solutions generated decreases. Thus, we clearly have a trade-off between accuracy and the number of solutions generated as we change p. It is clear from Table 1 that our new scheme IJGP(i,0)-SampleSearch is better than MBE(i,0)-SampleSearch both in terms of accuracy and the number of solutions generated. Also, no-good learning improves the accuracy of IJGP(i,p)-SampleSearch in most cases.

SAT benchmarks: We experimented with logistics and verification SAT benchmarks available from satlib.org. On all the these benchmarks instances, we had to reduce the number of solutions that each problem admits by adding unary clauses in order to apply

Table 2. KLD, Mean-squared Error and #Solutions for SAT benchmarks

	Logistics.a			Logistics.d			Verification1			Verification2		
	N=828,Time=1000s			N=4713,Time=1000s			N=2654,Time=10000s			N=4713,Time=10000s		
	IJGP(3,10)		WALK	IJGP(3,10)		WALK	IJGP(3,10)		WALK	IJGP(3,10)		WALK
	No Learn	Learn		No Learn	Learn		No Learn	Learn		No Learn	Learn	
KL	0.00978	0.00193	0.01233	0.0009	0.0003	0.0008	0.0044	0.0037	0.003	0.0199	0.0154	0.01
MSE	0.001167	0.00033	0.00622	0.00073	0.00041	0.0002	0.0035	0.0021	0.0012	0.009	0.0088	0.0073
#S	23763	32893	882	10949	19203	28440	1394	945	11342	1893	1038	8390

our exact algorithms. Here, we only experimented with our best performing algorithm IJGP(i,p)-SampleSearch with i=3 and p=10. From Table 2 we can see that on the logistics benchmarks, IJGP(3,10)-SampleSearch is slightly better than WALKSAT in terms of accuracy while on the verification benchmarks WALKSAT is slightly better than IJGP(3,10)-SampleSearch. WALKSAT however dominates IJGP(3,10)-SampleSearch in terms of the number of solutions generated (except Logistics.a).

7 Summary and Conclusion

The paper presents a new algorithm for generating random, uniformly distributed solutions for constraint satisfaction problems. This algorithm falls under the class of monte-carlo sampling algorithms that sample from the output of generalized belief propagation and extend our previous work [4]. We show how to improve upon conventional monte-carlo sampling methods by integrating sampling with back-jumping search and no-good learning. This has the potential of improving the performance of monte-carlo sampling methods used in the belief network literature [4], especially on networks having large number of zero probabilities. Our best-performing schemes are competitive with the state-of-the-art SAT solution samplers [6] in terms of accuracy and thus present a monte-carlo style alternative to random walk solution samplers like WALKSAT [6].

Acknowledgements

This work was supported by the NSF grants IIS-0331707 and IIS-0412854.

References

1. R. Dechter: Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence, 41:273312, 1990.
2. Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In AAAI, 2002.
3. Rina Dechter, Kalev Kask, and Robert Mateescu. Iterative join graph propagation. In UAI 02, pages 128136. Morgan Kaufmann, August 2002.
4. Vibhav Gogate and Rina Dechter. Approximate inference algorithms for hybrid bayesian networks with discrete constraints. UAI-2005, 2005.
5. Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. Technical report, University of California, Irvine, 2006.
6. Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In AAAI, 2004.

Sports League Scheduling: Enumerative Search for Prob026 from CSPLib

Jean-Philippe Hamiez and Jin-Kao Hao

LERIA, Université d'Angers, UFR Sciences
2, boulevard Lavoisier, 49045 Angers CEDEX 01, France
{Jean-Philippe.Hamiez, Jin-Kao.Hao}@univ-angers.fr

Abstract. This paper presents an enumerative approach for a sports league scheduling problem. This simple method can solve some instances involving a number T of teams up to 70 while the best known constraint programing algorithm is limited to $T \leq 40$. The proposed approach relies on interesting properties which are used to constraint the search process.

1 Introduction

This paper deals with "Prob026" from CSPLib [1], also known as the "balanced tournament design" problem in combinatorial design theory [2, pages 238–241]. It seems to be first introduced in [3].

- There are $T = 2n$ teams (i.e. T even). The season lasts $W = T - 1$ weeks. Weeks are partitioned into $P = T/2$ slots (periods);
- $c_{\mathcal{H}}$ constraint: All teams play each other exactly once ($\mathcal{H}$alf competition);
- $c_{\mathcal{W}}$ constraint: All teams play in each $\mathcal{W}$eek;
- $c_{\mathcal{P}}$ constraint: No team plays more than twice in the same $\mathcal{P}$eriod.

Various techniques were used to tackle Prob026: Integer programming [4,5] ($T \leq 12$), basic local search [4] ($T \leq 14$), local search with a many-valued propositional logic encoding [6] ($T \leq 16$), randomized deterministic complete search [7] ($T \leq 18$), local search with classical propositional logic encoding [8] ($T \leq 20$), constraint programming with powerful filtering algorithm [9] ($T \leq 24$), multiple threads [10] ($T \leq 28$), constraint programming [11] ($T \leq 30$), constraint programming with problem transformation [12] and tabu search [13] ($T \leq 40$).

Note that solutions exist for all $T \neq 4$ [14]. Furthermore, direct constructions have already been proposed when $(T - 1) \bmod 3 \neq 0$ or $T/2$ is odd [14,15,16]. This leaves open the cases where $T \bmod 12 = 4$.

In this paper, we present EnASS, an Enumerative Algorithm for Sports Scheduling for Prob026. Given T, EnASS starts building a particular conflicting schedule (called $\overline{s}$) verifying a set $\mathcal{R}$ of properties (or $\mathcal{R}$equirements). The set S of solutions is generated using $\overline{s}$ in a simple exhaustive way with backtracks and observed to identify new properties. $\mathcal{R}$ is then updated to solve Prob026 for larger T or to accelerate the resolution. Despite the exponential-time complexity of EnASS, we manage to build particular $\mathcal{R}$ sets that enable EnASS to find solutions to Prob026 for most T up to 70 in a reasonable amount of time. Note that similar ideas have been recently used for constraint reasoning [17].

2 Reducing the Complexity

Since any valid schedule can be thought of as a particular permutation of the $T(T-1)/2$ matches, the search space size is $[T(T-1)/2]!$. In other words, the search space size grows as the factorial of the square of $T/2$.

Patterned one-factorization [2, page 662, example 4.33] can be used to verify $c_{\mathcal{H}}$ and $c_{\mathcal{W}}$, the goal of `EnASS` being then to satisfy the last constraint $c_{\mathcal{P}}$. Form a regular polygon with the first $T-1$ teams. Draw W sets of $P-1$ parallels connecting vertices in pairs starting with each w side. Each set, augmented with the pair of missing teams, corresponds to the matches to place in week w [18]. Let $\overline{s}$ be the tournament obtained (in linear-time complexity) with this technique, where $\overline{s}\langle p,w\rangle$ is the match scheduled in period p and week w in $\overline{s}$. See [15] for a full detailed description and the formal model used to build $\overline{s}$.

Prob026 has symmetries that can be combined [19]: renumbering of the teams, permutation of weeks or / and periods. They can be avoided using patterned one-factorization and fixing the first week.

Prob026 solutions verify this property: In each p period, two different "$\mathcal{D}$eficient" [14] teams (a 2-set $\mathcal{D}_p$) appear exactly once. Furthermore, if one considers any $p' \neq p$ period, then $\forall t \in \mathcal{D}_p$, t appears twice in period p'. More formally, if $c_{\mathcal{D}}$ refers to this implicit constraint, then: $c_{\mathcal{D}}(p) \Leftrightarrow \forall p' \neq p, \mathcal{D}_{p'} \cap \mathcal{D}_p = \emptyset$.

3 Prob026: A Constraint Satisfaction Problem

Let $x = \langle p,w\rangle$ be any assignment of a match in period p and week w. Values of this variable type are of (t,t') pattern, meaning that team t meets team t' in period p and week w, noted $x \mapsto (t,t')$. So, the set X of variables is $X = \{x = \langle p,w\rangle, 1 \leq p \leq P, 1 \leq w \leq W\}$. Domains are defined according to the comments from the previous section: $\forall x = \langle p,1\rangle \in X, d_x = \{\overline{s}\langle p,1\rangle\}$ and $\forall x = \langle p,w\rangle \in X(w>1), d_x = \{\overline{s}\langle \overline{p},w\rangle, 1 \leq \overline{p} \leq P\}$. Since $\overline{s}$ verifies already $c_{\mathcal{W}}$ and $c_{\mathcal{H}}$, the set of constraints is only composed of the implicit $c_{\mathcal{D}}$ constraint (see Sect. 2) and $c_{\mathcal{P}}$: For each team t and each period p, we impose the constraint $c_{\mathcal{P}}(t,p) \Leftrightarrow |\{x = \langle p,w\rangle \mapsto (t,t'), 1 \leq w \leq W, t' \neq t\}| \leq 2$.

4 `EnASS`: Overall Procedure

Let $w_f = 2$ and $w_l = W$ be the first (respectively last) week that `EnASS` considers when filling any period and $\mathcal{R} = \mathcal{R}_0 = \{c_{\mathcal{P}},\, c_{\mathcal{D}}\}$.

`EnASS` requires three parameters: p and w identify the current variable, $\overline{p}$ specifies the value assignment tried. The function returns TRUE if a solution is found, FALSE otherwise. `EnASS` is called first, after building $\overline{s}$, with $(p,w,\overline{p}) = (1,2,1)$ meaning that it tries to fill period 1 of week 2 with the $\overline{s}\langle 1,2\rangle$ match. Note that we only give here the pseudo-code of `EnASS` for finding a first solution since it can easily be modified to return the entire set of all-different solutions.

`EnASS`$(p, w, \overline{p})$:

1. If $p = P + 1$ then return TRUE: A solution is obtained since all periods are filled and valid according to $\mathcal{R}$;
2. If $w = w_l + 1$ then return `EnASS`$(p + 1, w_f, 1)$: Period p is filled and valid according to $\mathcal{R}$, try to fill next period;
3. If $\overline{p} = P + 1$ then return FALSE: Backtrack since no value remains for $\langle p, w\rangle$;
4. If $\exists\, 1 \leq p' < p / \langle p', w\rangle = \overline{s}\langle \overline{p}, w\rangle$ then return `EnASS`$(p, w, \overline{p}+1)$: Value already assigned to a variable, try next value;
5. $\langle p, w\rangle \leftarrow \overline{s}\langle \overline{p}, w\rangle$: Try to assign a value to the current variable;
6. If $\mathcal{R}$ is locally verified and `EnASS`$(p, w + 1, 1) =$ TRUE then return TRUE: The assignment leads to a solution;
7. Undo step 5 and return `EnASS`$(p, w, \overline{p} + 1)$: $\mathcal{R}$ is locally violated or next calls lead to a failure, backtrack and try next value.

We will refer to this complete `EnASS` function with `EnASS`$_0$. All `EnASS` functions were coded in `C` (`cc` compiler) and ran on an Intel PIV processor (2 Ghz) Linux station. A time limit of 3 hours was imposed.

`EnASS`$_0$ solved Prob026 for all $T \leq 32$ in less than three minutes except for $T = 24$. This clearly outperforms [4,5,6,7,8,9,10,11] and competes well with [12,13].

5 Invariants in Prob026

We describe here exact `EnASS` variants that are no more complete since they work on a subset of the `EnASS`$_0$ solutions space.

Some solutions to Prob026 verify the following $r_{\Rightarrow}$ property: assume that $\langle p, w\rangle$ has been fixed to a match x with $w_f \leq w \leq P$, then x and the $\langle p, T - w + 1\rangle$ match appear in the same period in $\overline{s}$. More formally, $\forall\, w_f \leq w \leq P, r_{\Rightarrow}(p, w) \Leftrightarrow \langle p, w\rangle = \overline{s}\langle \overline{p}, w\rangle \Rightarrow \langle p, T - w + 1\rangle = \overline{s}\langle \overline{p}, T - w + 1\rangle$.

This leads to `EnASS`$_1$ which comes from `EnASS`$_0$ by setting $w_l = P$ and adding the $r_{\Rightarrow}$ requirement to $\mathcal{R}_0$: $\mathcal{R}_1 = \{c_{\mathcal{P}},\, c_{\mathcal{D}},\, r_{\Rightarrow}\}$.

Columns 2–4 in Table 1 give results obtained with `EnASS`$_1$: Number $|S_1|$ of solutions ("$\geq n$" indicates that `EnASS`$_1$ found n solutions when reaching the time limit), time (including the $\overline{s}$ construction) and number of backtracks to reach a first solution. "-" marks mean that `EnASS`$_1$ found no solution within the time limit or |BT| is larger than the maximal value authorized by the system.

`EnASS`$_1$ clearly outperforms `EnASS`$_0$ and [12,13]. However, other invariants are needed to tackle larger instances within the time limit. For this purpose, we reinforce the set $\mathcal{R}$ of requirements by adding the following two properties:

1. r_I: Inverse weeks w_f and W. More formally, $\forall\, w \in \{w_f, W\}, r_I(w) \Leftrightarrow \forall\, 1 \leq p \leq P, \langle p, w\rangle = \overline{s}\langle P - p + 1, w\rangle$;
2. r_V: Matches (t, T) form a "V" like pattern. More formally, $\forall\, 1 \leq p < P, r_V(p) \Leftrightarrow \langle p, p + 1\rangle = \overline{s}\langle P, p + 1\rangle$ and $\langle p, T - p\rangle = \overline{s}\langle P, T - p\rangle$.

This leads to EnASS$_2$ with $\mathcal{R}_2 = \{c_{\mathcal{P}}, c_{\mathcal{D}}, r_{\Rightarrow}, r_I, r_V\}$. Naturally, an additional step must be added in EnASS (between steps 1 and 2) due to r_V and w_f has to be set to 3.

Columns 5–7 in Table 1 give results obtained with EnASS$_2$. Note that no result is reported for $T \bmod 4 = 0$ or $T > 70$ since EnASS$_2$ failed in these cases within the time limit.

Table 1. Computational results (times in seconds)

T	EnASS$_1$			EnASS$_2$		
	$\lvert S_1\rvert$	Time	$\lvert$BT$\rvert$	$\lvert S_2\rvert$	Time	$\lvert$BT$\rvert$
32	$\geq$ 3 657 013	<1	332 306	-	-	-
34	$\geq$ 2 173 500	<1	1 342 216	≥ 1	<1	130 149
36	$\geq$ 1 122 145	<1	2 160 102	-	-	-
38	$\geq$ 692 284	5.34	13 469 359	≥ 1	<1	2 829 421
40	$\geq$ 523 804	6.25	16 393 039	-	-	-
42	$\geq$ 339 383	107.69	256 686 929	≥ 1	2.11	7 836 823
44	$\geq$ 236 614	876.91	1 944 525 360	-	-	-
46	$\geq$ 119 383	1 573.31	3 565 703 651	≥ 1	<1	1 323 929
48	$\geq$ 90 009	542.79	1 231 902 706	-	-	-
50	$\geq$ 19 717	6 418.52	-	≥ 1	13.75	47 370 701
54	-	-	-	≥ 1	10.59	29 767 940
58	-	-	-	≥ 1	269.88	827 655 311
62	-	-	-	≥ 1	279.38	494 071 117
66	-	-	-	≥ 1	7 508.51	1 614 038 658
70	-	-	-	≥ 1	8 985.05	-

6 Conclusion

We presented EnASS, an Enumerative Algorithm for Sports Scheduling, for Prob-026 from CSPLib. Based on this basic procedure, we derived two effective exact algorithms to constraint the search process by integrating solutions properties.

Computational results showed that these algorithms clearly outperform [4,5,6,7,8,9,10,11,13] and the best known constraint programming approach [12] which is limited to $T \leq 40$: EnASS solved Prob026 in a reasonable amount of time for all $T \leq 50$ and, for $50 < T \leq 70$, solutions have been generated for some T values.

EnASS is a simple enumerative algorithm with backtrack. One possible way to solve Prob026 for larger T or to speed up EnASS could be to use more elaborated backtracking techniques.

Acknowledgements

We would like to thank the reviewers of the paper for their useful comments. Special thanks go to Professor J.H. Dinitz from the University of Vermont for having suggested to us the origins of Prob026 and some important previous works.

References

1. Gent, I., Walsh, T.: CSPLib: A benchmark library for constraints. Volume 1520 of LNCS. Springer-Verlag (1998) 480–481 `http://www.csplib.org/`.
2. Colbourn, C., Dinitz, J., eds.: The CRC Handbook of Combinatorial Designs. Volume 4 of Discrete Mathematics and Its Applications. CRC Press (1996)
3. Gelling, E., Odeh, R.: On 1-factorizations of the complete graph and the relationship to round-robin schedules. Congressus Numerantium **9** (1974) 213–221
4. McAloon, K., Tretkoff, C., Wetzel, G.: Sports league scheduling. In: Proceedings of the Third ILOG Optimization Suite International Users' Conference. (1997)
5. Gomes, C., Selman, B., McAloon, K., Tretkoff, C.: Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In: Proceedings AIPS'98, AAAI Press (1998) 208–213
6. Béjar, R., Manyà, F.: Solving combinatorial problems with regular local search algorithms. Volume 1705 of LNAI. Springer-Verlag (1999) 33–43
7. Gomes, C., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings AAAI/IAAI'98, AAAI Press/MIT Press (1998) 431–437
8. Béjar, R., Manyà, F.: Solving the round robin problem using propositional logic. In: Proceedings AAAI/IAAI'00, AAAI Press/MIT Press (2000) 262–266
9. Régin, J.C.: Modeling and solving sports league scheduling with constraint programming. In: Proceedings INFORMS'98. (1998)
10. Wetzel, G., Zabatta, F. Technical report, City University of New York, USA (1998)
11. Van Hentenryck, P., Michel, L., Perron, L., Régin, J.C.: Constraint programming in OPL. Volume 1702 of LNCS. Springer-Verlag (1999) 98–116
12. Régin, J.C.: Constraint programming and sports scheduling problems. In: Proceedings INFORMS'99. (1999)
13. Hamiez, J.P., Hao, J.K.: Solving the sports league scheduling problem with tabu search. Volume 2148 of LNAI. Springer-Verlag (2001) 24–36
14. Schellenberg, P., van Rees, G., Vanstone, S.: The existence of balanced tournament designs. Ars Combinatoria **3** (1977) 303–318
15. Hamiez, J.P., Hao, J.K.: A linear-time algorithm to solve the Sports League Scheduling Problem (prob026 of CSPLib). Discrete Applied Mathematics **143** (2004) 252–265
16. Haselgrove, J., Leech, J.: A tournament design problem. American Mathematical Monthly **84** (1977) 198–201
17. Gomes, C., Sellmann, M.: Streamlined constraint reasoning. Volume 3258 of LNCS. Springer-Verlag (2004) 274–289
18. Lockwood, E.: American tournaments. The Mathematical Gazette **20** (1936) 333
19. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. Volume 2470 of LNCS. Springer-Verlag (2002) 462–476

Dynamic Symmetry Breaking Restarted

Daniel S. Heller and Meinolf Sellmann

Brown University
Department of Computer Science
115 Waterman Street, P.O. Box 1910
Providence, RI 02912
{dheller, sello}@cs.brown.edu

Symmetry breaking by dominance detection (SBDD) [4,6,1,12], has proven to excel on problems that contain large symmetry groups. The core task of SBDD is the dominance detection algorithm. The first automated dominance detection algorithms were based on group theory [7], while the first provably polynomial-time dominance checkers for specific types of value symmetry were devised in [15]. This work was later extended to tackle any kind of value symmetry in polynomial time [13]. Based on these results, for specific "piecewise" symmetric problems, [14] showed that breaking variable and value symmetry can be broken simultaneously in polynomial time. The method was named structural symmetry breaking (SSB) and is based on the structural abstraction of a given partial assignment of values to variables.

Compared with other symmetry breaking techniques, the big advantage of dynamic symmetry breaking is that it can accommodate dynamic variable and value orderings. Dynamic orderings have been shown to be vastly superior to static orderings in many different types of constraint satisfaction problems. However, robust heuristics for the selection of variables and values are hard to come by. For the task of variable selection, a bias towards variables with smaller domains often works comparably well, but there always remains a fair probability that we hit instances on which a solver gets trapped in extremely long runs. Particularly, heavy-tailed runtime distributions have been reported [9]. One way to circumvent this problematic situation is to randomize the solver and to restart the search when a run takes too long [10]. We show how symmetry no-goods can be used in restarted methods and introduce practical enhancements of SSB for its application in restarted solvers.

1 Symmetry No-Goods and Restarts

Unfortunately, due to space restrictions, we cannot review SSB here. For definitions and a detailed description of the method, we must therefore refer the reader to [14]. SSB (as a special form of SBDD) stores the most general previously fully expanded search nodes as a list of no-goods. In contrast to ordinary no-goods, an SBDD no-good implicitly represents a whole set of no-goods (namely the set of all its symmetric variants), and it is the algorithmic task of the dominance checker to see whether this set contains a no-good that is relevant with respect to the current search node.

What is interesting to note is that SBDD no-goods also keep a record of those parts of the search space that have already been searched through. In that regard, it is of interest to store them (or at least the most powerful ones) between restarts. There is a trade-off, however: No-goods will only be beneficial if the method that prevents us from exploring the same part of the search space more than once does not impose a greater computational cost than what the exploration would cost anyway. One simple thing that we can do is to remove those no-goods from the list that have very little impact anyway because they only represent a small part of the search space. This is an idea that is commonly used in SAT, too. However, for symmetry-nogoods we can do more.

2 Delayed Ancestor-Based Filtering

We introduce delayed symmetry filtering. The core idea here is to apply an inexpensive inference mechanism that quickly identifies which no-goods cannot possibly cause effective symmetry-based filtering at a given search node. Like this, we hope to save many of the expensive calls to SSB-based domain filtering. Note that no-goods are only used for ancestor-based filtering, which is why this idea will only be applied for this type of symmetry filtering. We will discuss special methods to improve the performance of sibling-based filtering in Section 3.

2.1 A Simple Pretest

We start by introducing a very simple pretest before a full-fledged ancestor-based filtering call is being made. What we need to identify are simple conditions under which a previously expanded node α (as usual, α is identified with the partial assignment that leads to the node) cannot "almost dominate" the current search node β. To make this more precise, with "almost" we mean to say that one more assignment to β could result to a successful dominance relation with α, which is a necessary condition for SSB filtering to have any effect.

First, we observe that β must contain at least as many variable assignments as α minus 1. This is a trivial condition which is always true in a one-shot tree search as all no-goods stored by SBDD were taken from search nodes at the same or lower depth as that of the current node. However, for no-goods stored in earlier restarts, this test can quickly reveal that ancestor-based filtering will not be effective.

Only if the above condition holds, we perform one more test before applying the full-fledged filtering call: we can look a little bit closer at the two assignments α and β and see whether α is close to dominating β. Before looking at each value individually, determining all their signatures and which ones dominates which, we can do the same on the level of value classes: For each value partition Q_l, we determine how many variables in each value partition are taking a value in it under assignment γ, thus computing a signature for each partition of mutually symmetric values: $sign_\gamma(Q_l) := (|\{X \in P_k \mid \gamma(v) \in Q_l\}|)_{k \leq r}$ for all $1 \leq l \leq s$.

Lemma 1. *Given assignments α and β such that α dominates β, we have that, for all $1 \leq l \leq s$, it holds that $sign_\alpha(Q_l) \leq sign_\beta(Q_l)$ (whereby with $\leq$ we denote the component-wise comparison of the two tuples).*

Proof. Let $l \in \{1, \dots, s\}$. Since α dominates β, we have that, for all $v \in Q_l$, it holds $sign_\alpha(v) \leq sign_\beta(w)$ for some value $w \in Q_l$ that is the unique matching partner of v. Consequently, it holds that $sign_\alpha(Q_l) = \sum_{v \in Q_l} sign_\alpha(v) \leq \sum_{w \in Q_l} sign_\beta(w) = sign_\beta(Q_l)$. □

Thus, SSB filtering can only be effective if the inequality holds for all but at most one value partition l, and if for that partition we have that $sign_\alpha(Q_l) \leq sign_\beta(Q_l) + e_k$, where e_k is the unit vector with a 1 in position $1 \leq k \leq r$. Only if this condition holds, we finally apply ancestor-based filtering.

Note that our simple pretest can be conducted much faster than a full-fledged filtering call: it runs in time linear in the size of the given assignments (which is in $O(n)$) whereas ancestor-based filtering wrt each ancestor requires time $O(m^{2.5} + mn)$ for a CSP with n variables and m values.

2.2 Deterministic Lower Bounds

Assume that a call to the ancestor-based filtering procedure reveals that we are at least p edges short of finding a perfect matching in the value dominance graph that was set-up for assignments α and β.[1] Clearly, as was already noted in [14], this means that at least another $p - 1$ variable assignments need to be added to β before filtering can become effective. By adding this information to no-good α, and by keeping track of the depth of the current search node β, we can avoid many useless filtering calls. What is interesting is that we cannot only propagate this information when diving deeper into the tree, but also upon backtracking.

Consider the following situation: For no-good α, the check against the current search node in depth d results in a maximum matching with 4 edges missing to be perfect. Then, at depth $d + 4 - 1 = d + 3$, we call for ancestor filtering wrt α again and find that there are still 4 edges missing. Clearly, this means that none of the last 3 branching decisions has brought us any closer to a successful dominance relation with α, and this information can be used even when backtracking up from the current position. At depth $d + 2$, for example, we know, even without conducting the filtering call, that the maximum matching must have 4 edges missing. Which implies that, when diving deeper into the tree from depth $d + 2$, ancestor-based filtering cannot be effective until we reach depth $d + 2 + 4 - 1 = d + 5$.

More generally, if at depth $d + p - 1$ we find a maximum matching with $q \leq p$ edges missing to perfection, when backtracking up to depth $d + r - 1$ for some $r < p$, we are sure that there are at least $\max\{r + q, p\} - 1$ more variable assignments necessary before filtering wrt ancestor α can be effective. Consequently, we will not call for ancestor-based filtering wrt α until we reach depth $\max\{d + r + q - 1, d + p - 1\}$ in the search tree.

Note that the above procedure also works if we never get to perform the full filtering procedure at depth $d+p-1$ because our pretest fails: If the first condition fails, instead of using the number of missing perfect matching edges, we can simply count the number of variable assignments still needed before at least as many variables are assigned in the current search node as are in α. And in the second case, we can count how many more assignments are necessary before each value class signature in α can have become lower or equal to the signatures in the current partial assignment.

3 Incremental Sibling-Based Filtering

Sibling-based filtering requires that we compute the sets of mutually symmetric values that have the same signature under the current partial assignment. Rather than recomputing the signatures of all values and regrouping the values after a branching step has

[1] Note that, due to the special way that SBDD unifies no-goods, $p > 1$ is only possible for no-goods generated in earlier runs.

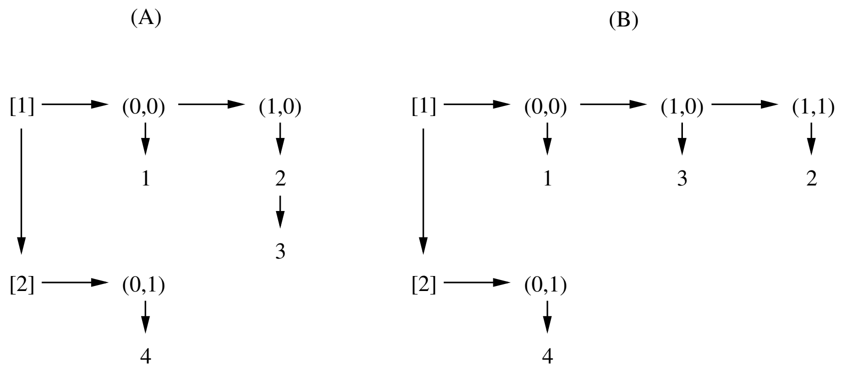


Fig. 1. An efficient data structure supporting sibling-based filtering incrementally. The leftmost column depicts value partitions $[1] = \{1, 2, 3\}$ and $[2] = \{4\}$. For both partitions, horizontally it follows a sorted list of signatures (over two variable partitions in this example) that are each associated with all the values underneath them in the current partial assignment. Even though signatures are actually stored in sparse format, we show them explicitly to improve the readability. Value 2 in the left assignment for instance has signature (1,0) and shares it with value 3.

added another variable assignment, we use an incremental data structure for this purpose so as to conduct this type of symmetry related inference as efficiently as possible.

First, let us describe the idea of sparse signatures that are needed to guarantee the worst-case complexity as given in [14]: Instead of writing down entire signatures, for each value we maintain a sparse list that only contains the non-zero entries of a signature, together with the information to which variable partition an entry in the sparse list belongs. To set up this sparse representation from a new partial assignment, we first order the variable instantiations in a given partial assignment according to the partition that the corresponding variable belongs to. This can be done in time linear in the number of variable partitions. In this order, we then scan through the partial assignment and set up the sparse signatures simply by adding one to the last entry if the current variable belongs to the same partition as the last, and by introducing a new non-zero entry if the variable belongs to a new partition.

For the current search node, we group values in the same value partition and with the same signature in the following data structure. It consists of an array of lists, one for each value partition. Each such list contains, in lexicographic order, the different signatures within the respective value partition. Associated with each signature is yet another list of values in the partition that have the signature, whereby each value holds a pointer to the signature. Note that this data structure allows us to perform sibling based-filtering extremely efficiently. Given a variable, the different values that we need to consider when branching are only the first values in each list of each signature in each value partition.

Now, when branching by assigning a value to some variable, we update the data structure incrementally. Note that the value assigned is the first in its list of values with the same signature. Moreover, the value holds a pointer to its signature. Therefore, we can compute its new signature incrementally, and since the signatures within the value partition are ordered lexicographically, we can also find out quickly to which signature the value needs to be added, whereby we create a new list of values if the value's new signature is not yet in our list. Finally, we remove the value from the list of values for its old signature and add it to the list of values for its new signature, while updating the value's pointer to its own signature.

We illustrate the data structure in Figure 1 on the following example. Assume we are given four variables $X_1, \ldots, X_4$, whereby the first two and the last two are symmetric. Assume further that the variables can take four values $1, \ldots, 4$, whereby the first three are symmetric. Figure 1 (A) shows our incremental data structure for the partial assignment $\{(X_1, 2), (X_2, 3), (X_3, 4)\}$. We see that we can easily pick non-symmetric values simply by choosing the first representative for each signature. In our example, those are the values $1, 2$, and 4. Figure 1 (B) shows the data structure after another variable has been instantiated by adding $(X_4, 2)$ to our assignment. We see that the data structure can easily be adapted by updating the signature of value 2.

4 Conclusions

We introduced two practical algorithmic enhancements of structural symmetry breaking: delayed ancestor-based filtering and incremental sibling-based filtering. Especially the first is designed for the application in restarted solvers where it reduces the computational efforts when using symmetry no-goods from previous restarts.

References

1. N. Barnier and P. Brisset. Solving the Kirkman's schoolgirl problem in a few seconds. *Proceedings of CP'02*, 477–491, 2002.
2. C. Brown, L. Finkelstein, P. Purdom Jr. Backtrack searching in the presence of symmetry. *Proceedings of AAECC-6*, 99–110, 1988.
3. J. Crawford, M. Ginsberg, E. Luks, A. Roy. Symmetry-breaking predicates for search problems. *Proceedings of KR'96*, 149–159, 1996.
4. T. Fahle, S. Schamberger, M. Sellmann. Symmetry Breaking. *Proceedings of CP'01*, 93–107, 2001.
5. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh. Breaking row and column symmetries in matrix models. *Proceedings of CP'02*, 462–476, 2002.
6. F. Focacci and M. Milano. Global cut framework for removing symmetries. *Proceedings of CP'01*, 77–92, 2001.
7. I. Gent, W. Harvey, T. Kelsey, S. Linton. Generic SBDD using computational group theory. *Proceedings of CP'03*, 333–347, 2003.
8. I. Gent and B. Smith. Symmetry breaking in constraint programming. *Proceedings of ECAI'00*, 599–603, 2000.
9. C.P. Gomes, B. Selman, N. Crato. Heavy-Tailed Distributions in Combinatorial Search. *Proceedings of CP'97*, 121–135, 1997.
10. Dynamic Restart Policies. H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, B. Selman. *Proceedings of AAAI'02*, 674–682, 2002.
11. S. Prestwich and A. Roli. Symmetry Breaking and Local Search Spaces. *Proceedings of CPAIOR'05*, 273–287, 2005.
12. J.-F. Puget. Symmetry breaking revisited. *Proceedings of CP'02*, 446–461, 2002.
13. C. Roney-Dougal, I. Gent, T. Kelsey, S. Linton. Tractable symmetry breaking using restricted search trees. *Proceedings of ECAI'04*, 211–215, 2004.
14. M. Sellmann and P. Van Hentenryck. Structural Symmetry Breaking. *Proceedings of IJCAI'05*, 298–303, 2005.
15. P. Van Hentenryck, P. Flener, J. Pearson, M. Agren. Tractable symmetry breaking for CSPs with interchangeable values. *Proceedings of IJCAI'03*, 277–282, 2003.

The Effect of Constraint Representation on Structural Tractability

Chris Houghton, David Cohen, and Martin J. Green

Department of Computer Science,
Royal Holloway, University of London, UK

Abstract. Tractability results for structural subproblems have generally been considered for explicit relations listing the allowed assignments. In this paper we define a representation which allows us to express constraint relations as either an explicit set of allowed labelings, or an explicit set of disallowed labelings, whichever is smaller. We demonstrate a new structural width parameter, which we call the interaction width, that when bounded allows us to carry over well known structural decompositions to this more concise representation. Our results naturally derive new structurally tractable classes for SAT.

1 Introduction

An instance of the constraint satisfaction problem is a collection of variables to be assigned, a universe of possible values and a collection of constraints. Each constraint has a relation which restricts the allowed simultaneous assignments to a set of these variables. This set of variables is called the scope of the constraint.

The constraint satisfaction problem is, in general, NP-hard. As such, it is an important area of research to identify subproblems which are tractable.

The structure of a constraint satisfaction problem instance (CSP) is defined to be the hypergraph whose vertices are the variables of the instance and whose hyperedges are the constraint scopes.

When we are given a CSP to solve, the constraint relations will be expressed in some encoding, that is, by some sequence of symbols. We classify these encodings based on the expression method used. The set of encodings that we allow a class of CSPs to be expressed by is called a representation. Practitioners generally use the most concise representation they can to express a problem.

Most research on tractability has been concerned with explicitly allowed encodings [2], which we shall refer to as the positive representation. In this paper we wish to investigate how well our current theory translates into a more natural representation.

We define a notion of structural width for a hypergraph which we call the interaction width. We are able to show that certain structural classes with bounded interaction width are tractable.

We describe SAT in terms of our complement representation and show why this is a natural approach for discussing the structural tractability of classes of SAT instances.

2 CSPs and Representations

Definition 1. *A constraint satisfaction problem instance (CSP) is a triple $\langle V, D, C\rangle$ where; V is a set of variables, D is a finite set which we call the universe of the problem, and C is a set of constraints.*

Each constraint *$c \in C$ is a pair $\langle \sigma, \rho\rangle$, where σ (called the constraint scope) is a subset of V and ρ (called the constraint relation) is a set of labelings of σ. Each* labeling *is a function from σ into the universe D.*

A solution *to a CSP, $P = \langle V, D, C\rangle$ is a mapping $s : V \rightarrow D$ such that for every $\langle \sigma, \rho\rangle \in C$ we have that s restricted to σ is an element of ρ.*

A hypergraph *is a pair, $\langle V, E\rangle$, where V is a set of* vertices *and E is a set of subsets of V, called the* hyperedges. *For any CSP $P = \langle V, D, C\rangle$, the* structure *of P, denoted $\sigma(P)$, is the hypergraph $\langle V, \{\sigma \mid \langle \sigma, \rho\rangle \in C\}\rangle$.*

The class of CSPs with *acyclic* structure is tractable [1].

Example 1. Let $\mathcal{A}$ be the class of CSPs generated by taking an instance of graph 3-coloring and adding a universal constraint which allows all labelings. This does not alter solution, but forces instances of $\mathcal{A}$ to have acyclic structure.

This anomaly relies on the universal constraint being expressed by listing every possible assignment to all variables in the CSP. An encoding is the way in which a constraint relation is expressed. It is usual for the labelings in a constraint relation to be encoded as the allowed assignments to the variables in the scope. We shall refer to this encoding as the *positive encoding*. The labelings may also be encoded as the disallowed assignments (*complement encoding*).

Definition 2. *A* representation, *R, is a set of possible encodings. A CSP, $P = \langle V, D, C\rangle$, is said to be expressed in a representation, R, if for every constraint, $\langle \sigma, \rho\rangle \in C$, ρ is expressed by the encoding in R which has the smallest size for ρ. (This is similar to the concept of Minimum Description Length [3].)*

The representation which allows only the positive encoding is called the positive representation *(*Pos*) and the representation which allows only the complement encoding is called the* complement representation *(*Comp*). The representation which allows both encodings is called the* mixed representation *(*Mixed*).*

3 Tractability with Respect to Representation

We show that the tractable classes of each of these representations is distinct by demonstrating classes which distinguish them.

Definition 3. *A class of CSPs is called* tractable *if there is a polynomial time algorithm to decide membership and to solve the instances of the class.*

Define by $\mathrm{T}(\mathcal{R})$ *the tractable classes of representation $\mathcal{R}$.*

Proposition 1. *Consider two representations, $\mathcal{R}$ and $\mathcal{Q}$, such that $\mathcal{Q} \subseteq \mathcal{R}$. Assuming that there is a polynomial conversion from relations expressed w.r.t. $\mathcal{Q}$ to relations expressed w.r.t. $\mathcal{R}$, then for a set, S, of CSPs, we have that if $S \in \mathrm{T}(\mathcal{R})$ then $S \in \mathrm{T}(\mathcal{Q})$.*

Proof. Let P be any CSP in S expressed w.r.t. $\mathcal{Q}$. We use the polynomial time conversion to change the expression of P from $\mathcal{Q}$ to $\mathcal{R}$ and then solve using the algorithm for S w.r.t. $\mathcal{R}$.

Corollary 1. *Let S be a set of CSPs such that $S \in \mathrm{T}\,(\mathrm{Mixed})$. We have that $S \in \mathrm{T}\,(\mathrm{Pos})$ and $S \in \mathrm{T}\,(\mathrm{Comp})$.*

Proof. Any relation expressed in the larger of the two encodings must list more than half the possible number of assignments. The universal constraint on this scope is at most twice as big, so we are able to generate the other encoding.

Class $\mathcal{A}$ from Ex. 1 is in $\mathrm{T}\,(\mathrm{Pos})$ but not $\mathrm{T}\,(\mathrm{Comp})$. The universal constraint has no size when expressed in Comp so is directly equivalent to graph 3-coloring.

Example 2. Let $\mathcal{B}$ be the class of CSPs with $2n$ variables generated by taking an instance of graph 3-coloring over n of the variables and adding a single constraint, c_1, over the remaining n variables which allows only a single labeling.

The class of instances, $\mathcal{B}$, in example 2 is not tractable when expressed in Pos. The added constraint, c_1, is small when expressed in the positive encoding, and so the problem is directly equivalent to graph coloring. When expressed in Comp, the size of this constraint dominates the size of the graph coloring component and so a simple polynomial time algorithm is to test all possible assignments to the graph coloring component.

So, $\mathrm{T}\,(\mathrm{Pos})$ is incomparable to $\mathrm{T}\,(\mathrm{Comp})$ as neither is contained in the other. However, anything in $\mathrm{T}\,(\mathrm{Mixed})$ must also be in both $\mathrm{T}\,(\mathrm{Pos})$ and $\mathrm{T}\,(\mathrm{Comp})$, $\mathrm{T}\,(\mathrm{Mixed})$ must be a proper subset of both $\mathrm{T}\,(\mathrm{Pos})$ and $\mathrm{T}\,(\mathrm{Comp})$. This poses the question; does $\mathrm{T}\,(\mathrm{Mixed})$ contain any interesting (non-trivial) classes?

4 Converting Mixed to Pos

We shall show that by bounding some new notion of structural size, called the *interaction width*, we can convert certain CSPs from Mixed to Pos. If there are subclasses of CSP classes in $\mathrm{T}\,(\mathrm{Pos})$ for which there is a polynomial conversion from Mixed to Pos, then such subclasses are tractable w.r.t. Mixed.

If we represent a hypergraph as a Venn Diagram, where the hyperedges are the sets, an interaction region of the hypergraph is a region of the Venn Diagram. Interaction width is the maximal number of regions over any of the hyperedges.

Definition 4. *Let $H = \langle V, E \rangle$ be a hypergraph. We define the* interaction *on vertex $x \in V$, denoted $\tau\,(x)$, to be the set of edges containing x so that $\tau\,(x) = \{e \in E \mid x \in e\}$.*

We define I to be the set of interactions for all vertices, $I = \{\tau\,(x) \mid x \in V\}$. We define $I\,(e)$ to be the set of interactions for the vertices which are in the edge e so that $I\,(e) = \{X \in I \mid e \in X\}$. The interaction region*, $V\,(X)$, associated with the interaction $X \in I$ is the set of vertices which are in the same interaction as X that is, $V\,(X) = \{x \in V \mid \tau\,(x) = X\}$. The* interaction width*, denoted $I_w\,(H)$, of H is the largest number of non-singleton interactions associated with any of its edges; $I_w\,(H) = \max\{|I\,(e) - \{\{e\}\}| \mid e \in E\}$.*

There are two types of interaction region in which we may not have enough information to do the conversion in polynomial time. We call these 'isolated regions' and 'trivial complement regions'.

For a hypergraph, $H = \langle V, E\rangle$, we define the removal of a set of vertices, $V' \subseteq V$, to be the hypergraph $H' = \langle V', E'\rangle$ where $E' = \{e \cap V' \mid e \in E\}$.

It is straightforward to show that removing a set of vertices does not increase the structural decomposition width of a hypergraph. Structural decomposition of the original structure may therefore be used to solve the converted instance.

Our conversion requires that we project out certain interaction regions.

Definition 5. *The projection of a constraint, $\langle \sigma, \rho\rangle$ onto a subset, X, of its scope is the constraint $\langle X, \{f_{|X} \mid f \in \rho\}\rangle$.*

Method 1 performs projection in polynomial time for constraints expressed w.r.t. Comp and the resulting constraint is also expressed w.r.t. Comp.

Method 1. *Given an encoding of a constraint, $\langle \sigma, \rho\rangle$, where ρ is a set of disallowed assignments, and a subset of the variables in the scope, $\sigma' \subseteq \sigma$ we can project ρ onto σ' in the following way;*

Restrict the assignments of ρ so that they are only over the variables of σ' to give ρ'. For every l in ρ', if every possible extension to l exists in ρ then keep l in ρ', else discard l.

After performing projection on a constraint represented w.r.t. Pos, it may then allow more than half the possible assignments and need to be converted to Comp (which can be done in polynomial time for the same reason).

Definition 6. *Given a hypergraph, $H = \langle V, E\rangle$, for each $e \in E$ the region associated with the interaction $\{e\}$ is called an* isolated region.

If we project out the isolated regions of a CSP, we can solve the reduced instance and extend any solution to the isolated regions of the original instance.

Definition 7. *Let $P = \langle V, D, C\rangle$ be a CSP encoded w.r.t.* Mixed *or* Comp *and X be an interaction of $\sigma(P)$. Let $C_X \subseteq C$ be the constraints with scopes in X. X is a* complement interaction *if all constraints in C_X are encoded w.r.t.* Comp.

Let ρ be the set of all assignments from the given (complement) encoding of the constraints in C_X restricted to the region $V(X)$. If ρ does not contain all possible assignments over $V(X)$, $|\rho| < |D|^{|V(X)|}$, then we call $V(X)$ a trivial complement region.

We can remove trivial complement regions as not all disallowed assignments exist in ρ so any missing assignment must be allowed by all extensions for every constraint in C_X. We can see that ρ can be generated in polynomial time. By assuming an order on assignments we can easily check if one is missing from ρ. We can stop after finding a single missing assignment and remember it for the purpose of extending solutions later.

Let $\mathcal{H}$ be a set of hypergraphs with interaction width i. We can now provide an algorithm for converting any CSP represented w.r.t. Mixed and whose structure is in $\mathcal{H}$ to a solution preserved CSP represented w.r.t. Pos.

Method 2. *INPUT: CSP $P = \langle V, D, C\rangle$ and the hypergraph of P, $H = \langle V, E\rangle$.*

1. *Find and project out all isolated regions.*
2. *Find and project out trivial complement regions, remembering extensions. (Let H' be the reduced structure and $P' = \langle V', D, C'\rangle$ be the reduced instance after projecting out isolated regions and trivial complement regions.)*
3. *Convert the reduced instance to the positive representation.*
   - *Create a mapping, L, which maps from interactions I of the hypergraph H' to sets of assignments on the respective interaction regions such that*
     - *For each interaction, $X \in I$, if there exists a constraint, $\langle\sigma', \rho'\rangle \in C'$ whose relation is expressed w.r.t.* Pos *and whose scope contains $V(X)$, the region of the interaction X, then set $L(X)$ to be $\rho_{|_{V(X)}}$. Else, set $L(X)$ to be the set of possible assignments to $V(X)$ over D.*
   - *Create a new CSP, $\bar{P} = \langle V', D, \bar{C}\rangle$ with structure H' such that*
     - *For each hyperedge σ' of H', create the new constraint in $\langle\sigma', \bar{\rho}\rangle$ in $\bar{P}$ such that $\bar{\rho}$ is the product over $X \in I(\sigma')$ of $(L(X))$.*
     - *For each constraint $\langle\sigma', \bar{\rho}\rangle \in \bar{C}$, then for every constraint $\langle\sigma', \rho'\rangle \in C'$, if ρ' is represented w.r.t.* Pos*, then $\bar{\rho} := \bar{\rho} \cap \rho'$. Else, $\bar{\rho} := \bar{\rho} - \rho'$.*

It is straightforward to show that this algorithm runs in polynomial time for bounded interaction width. The bound is required for generating the new constraints based on the products of the assignments over the regions.

Once we have solved the reduced CSP we can then extend the found solution to the trivial complement regions and the isolated regions of the original CSP.

For any tractably identifiable structural decomposition, such as bounded width hypertrees [2], we generate a new tractable class w.r.t. Mixed.

4.1 Structurally Tractable Classes of SAT

Each clause in a SAT instance only disallows a single assignment. There can be no polynomial time conversion from clauses to Pos unless the arity is bound.

However, SAT may be naturally represented in Mixed. As such, structural tractability results (with bounded interaction width) naturally extend to SAT.

Szeider [4] has also developed a structural tractability result for SAT which is based on the *treewidth* of the so called *incidence graph*. We can show that even just for SAT, these two structurally tractable classes are incomparable, so there are two distinct structural tractability results for SAT. However, ours has a natural extension to domains of larger size, so we hope may be applicable to other practical problems.

References

1. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30:479–513, 1983.
2. Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural csp decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.
3. J. Rissanen. Modeling by shortest data description. In *Automatica, vol. 14*, 1978.
4. Stefan Szeider. On fixed-parameter tractable parameterizations of sat. In *SAT*, pages 188–202, 2003.

Failure Analysis in Backtrack Search for Constraint Satisfaction*

Tudor Hulubei and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
`tudor@hulubei.net, b.osullivan@cs.ucc.ie`

1 Introduction

Search effort is typically measured in terms of the number of backtracks, constraint checks, or nodes in the search tree, but measures such as the number of incorrect decisions have also been proposed. Comparisons based on mean and median effort are common. However, other researchers focus on studying runtime distributions, where one can observe a (non-)heavy-tailed distribution under certain conditions [2, 3].

In this paper we augment our traditional statistics-based approach to studying systematic search with a visualisation method that uses heatmaps, which can be more informative and present different views of the relationship between the depth at which a mistake occurred and the size of the refutation associated with it. We compare search algorithms on the basis of the number of, and effort required to recover from, *individual mistakes*. We highlight interesting differences between random and real-world problems, contradicting conventional wisdom that states that mistakes at the top of the search tree are much more expensive to refute than those made deeper in the tree.

We also observe some interesting patterns in terms of where most of the search effort is consumed *over a large population of problem instances* and show that it is not always the case that extremely large mistakes account for most of the effort. Finally, we show that variable ordering heuristics alone can avoid making mistakes, but that their performance cannot be attributed exclusively to either fail-firstness or promise.

2 Experiments

We study the failure characteristics of backtrack search methods in constraint satisfaction problems. Our analysis is based on counting the number of times an assignment was made during search that took us off the path to a solution. We refer to such decisions as *mistakes* [5]. The set of nodes visited by the algorithm in order to recover from a mistake is the *refutation tree* of that mistake. The number of nodes in that tree is the *refutation size*, which is our measure of effort.

Our empirical analysis includes configurations of uniform Model B random binary problems and quasigroup completion problems, encoded using binary constraints. With

* This work was supported by Science Foundation Ireland (Grant 00/PI.1/C075). We thank the Boole Centre For Research in Informatics for providing access to their Beowulf cluster.

the exception of the random 17×8 problems (i.e. 17 variables with uniform domain size 8), where we used backtracking, all other experiments used MAC. Our data sets of random problems contain approximately 10,000 instances for each algorithm used. The QWH-10 data set includes a total of over 1,000,000 instances. Specifically, our data sets comprise of the following problems:

1. Dense random 30×10, density 0.86, tightness 0.15, at the phase transition.
2. Sparse random 30×10, density 0.3, tightness 0.35, in the easy region.
3. Sparse random 150×10, density 0.03356, tightness 0.52, in the easy region.
4. QWH-10 with 90% random balanced holes.
5. Random 17×8, density 0.84, tightness 0.09375 (easy but heavy-tailed when using random orderings); and 0.25, near the phase transition and non-heavy-tailed.

Rather than using the overall effort required to solve each instance, which we refer to as *instance-based effort*, as the basis of our analysis, we also considered the effort required to refute each mistake separately, which we refer to as *mistake-based effort*. We established that they were highly correlated, and so observations made on the latter can be used to draw conclusions about the former. Studying search algorithms at the mistake-level allows us to perform a more detailed analysis of the interactions between variable and value ordering heuristics over a large population of instances. For the remainder of the paper we will base our comparison of search heuristics on an analysis of mistake-level effort through the use of heatmaps (best viewed in colour at `http://hulubei.net/tudor/papers/fabscs`).

2.1 Distribution of Search Effort

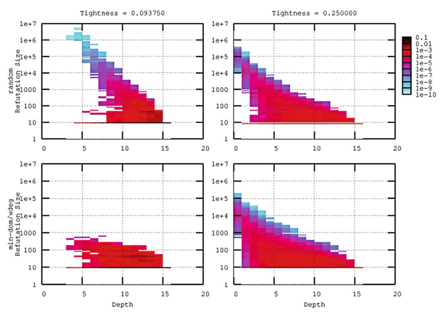
For random problems 17×8, as well as for QWH-10 with 90% holes, Figure 1 shows heatmaps of the probability of encountering mistakes of a certain size at a certain depth (Figures 1(a) and 1(c)), as well as the proportion of effort spent at a certain depth in refutations of a certain size (Figures 1(b) and 1(d)). Colours represent a log-scale.

Random 17×8 instances, using a random variable ordering with backtrack search exhibit heavy tails in the easy region, but not in the hard region. While a survival function-based analysis [2] can demonstrate the presence or absence of such large mistakes, heatmap visualisations also show precisely the depth where these mistakes occur.

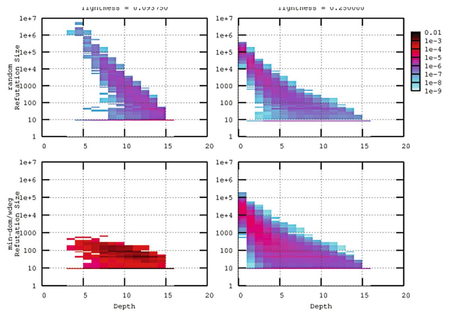
It is also interesting to consider how the size of mistakes varies with depth. The conventional wisdom is that mistakes made at, or near, the top of the tree are exponentially larger than those made deeper in the tree. However, the QWH-10 plots in Figure 1 contradict that assumption – the largest mistakes occur at *intermediate* depths.

The heatmap in Figure 1(d) depicts, *for a population of instances*, the proportion of effort required to refute, for QWH-10 with 90% holes, mistakes of various sizes at each depth. The darkest spots in the heatmap represent those mistake sizes for which the cumulative effort over all the mistakes in our data set was proportionally the largest at that depth. It is clear that over a population of instances, for all the algorithms we used, the bulk of the effort is spent in refuting the extremely large number of small mistakes (4 to 100 nodes) that occur deep down in the search tree (Figure 1(c)).

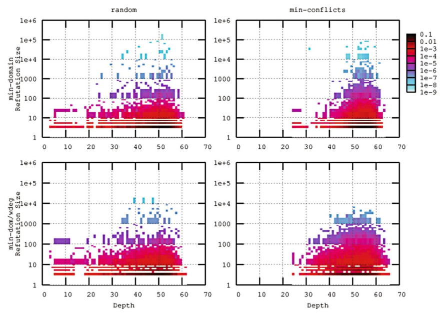
Random binary problems do exhibit exponential decay of the refutation size with depth (Figures 1(a), 1(b) and 2(d), similar results for random 150×10 and 30×10).



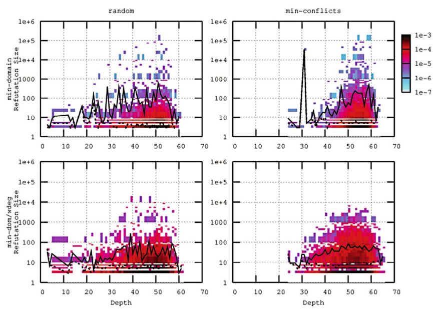
(a) Random 17 × 8 (left: easy, right: hard): probability of a refutation of a certain size (y-axis) occurring at a given depth (x-axis).



(b) Random 17 × 8 (left: easy, right: hard): proportion of effort spent in refutations of a certain size (y-axis) at a given depth (x-axis).



(c) QWH-10 problems with 90% holes: probability of a refutation of a certain size (y-axis) occurring at a given depth (x-axis).



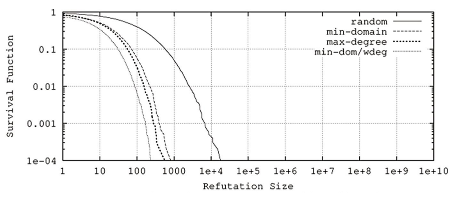
(d) QWH-10 problems with 90% holes: proportion of effort spent in refutations of a certain size (y-axis) at a given depth (x-axis). Lines: dotted=medians, continuous=means.

Fig. 1. Heatmaps for random problems 17 × 8 and QWH-10 with 90% holes

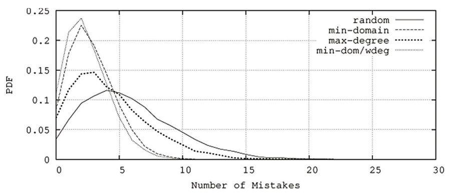
Moreover, as random problems approach the phase transition, mistakes start occurring close to the root of the tree, and over a population of instances, the bulk of the effort, which corresponds to the dark colour in the heatmaps, shifts towards the top of the tree.

Clearly, the mean and median refutation sizes in Figure 1(d) fail to provide any information with respect to the wide range and distribution of refutation sizes encountered here. The large number of small-to-medium size refutations not only keeps the median below 10, but also prevents the extremely large mistakes from contributing to the mean. Furthermore, the medians and means in Figure 1(d) do not indicate where the effort is in terms of depth. While *for a population of instances* of QWH-10 the effort seems dominated by the disproportionately large number of small mistakes, *for any particular difficult instance*, a small number of large refutations dominate the search effort.

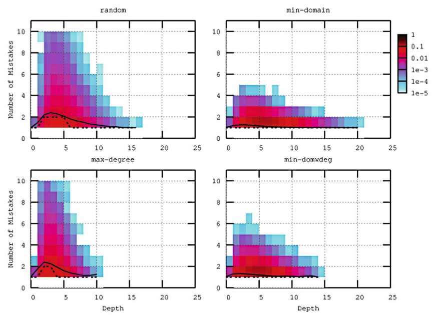
MAC with min-conflicts and min-dom/wdeg is the only algorithm in our arsenal that can eliminate heavy tails for QWH-10 [5]. The bottom-right plot in Figure 1(d) shows a far smaller *variation* in the size of the refutations than any of the other 3 plots. This translates into a significant reduction in the variation in instance-based effort and is



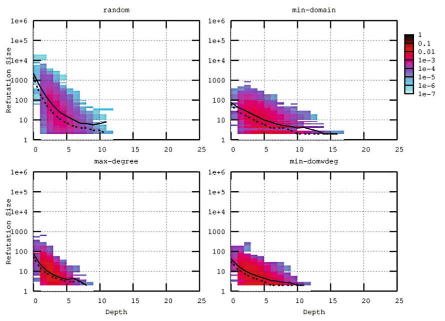
(a) 1-CDF for the runtime distribution.



(b) PDF of the number of mistakes per instance.



(c) Probability of a certain number of mistakes (y-axis) at a given depth (x-axis) in an instance. Lines: dotted=medians, continuous=means.



(d) Proportion of effort spent in refutations of a certain size (y-axis) at a given depth (x-axis). Lines: dotted=medians, continuous=means.

Fig. 2. Sparse random 30×10: Promise vs fail-firstness; 4 variable orderings, random values

consistent with the non-heavy-tailed nature of that data set. Similar, but more complex behaviour can be observed in Figure 1(b). The upper-left heatmap is the only one corresponding to a heavy-tailed distribution. There are two characteristics of the other three heatmaps that help in visually determining the absence of heavy-tails: in the lower-left plot there is insufficient variation in the refutation sizes; in the plots on the right, the greatest proportion of our effort is associated with the large refutations.

2.2 Promise Versus Fail-Firstness

Good variable ordering heuristics reduce the effort required to refute insoluble subtrees using a property called fail-firstness [4]. Variable ordering heuristics have also been shown to exhibit promise, i.e. they can contribute to a search algorithm's ability to avoid making mistakes [1]. Promise was measured previously based on the probability that search remains on the path to a solution. Here we measure it very differently and present an alternative analysis of its complex interaction with fail-firstness.

The remainder of our experiments are based on sparse random problems with 30 variables and uniform domain size 10. Figure 2(a) clearly shows that min-dom/wdeg and random variable orderings are the best and worst heuristics, respectively, and that max-degree performs better than min-domain. Further supporting the usefulness of heatmaps, we can see how Figure 2(b), as well as the means and medians in Figure 2(c), portray min-domain and min-dom/wdeg as being very similar. The heatmaps in

Figure 2(c), however, clearly show the mistakes made by the two heuristics are distributed differently across depths, with min-dom/wdeg having a higher probability of making more mistakes per instance over almost the entire range of depths it covers.

Figures 2(c) and 2(d) present a measure of promise and fail-firstness, respectively, for each variable ordering heuristic, arranged left to right and top to bottom in increasing order of performance, as per Figure 2(a). A comparison of the various heuristics depicted there may seem contradictory at first: max-degree seemingly outperforms min-domain due to its better fail-firstness and despite its worse promise (more mistakes), while min-dom/wdeg performs better than max-degree due to its better promise, and despite its slightly worse fail-firstness. Smith and Grant [6] showed that trying harder to fail first does not always improve performance, and our experiments support a similar conclusion for promise. From the heuristics studied here, min-dom/wdeg performs best not because it makes fewer mistakes, or because it refutes them with less effort, but because it strikes a good balance between these two properties.

3 Conclusions

Our novel use of heatmaps nicely complements the use of survival functions and allows a more granular view of the complex interaction between a heuristic's ability to avoid mistakes and its ability to recover from them. Heatmaps have helped show very clearly that the effort required to recover from mistakes is not always correlated with the depth where they occur, and better search heuristics do not necessarily make fewer mistakes, or have the ability to recover from them quickly.

References

1. J.C. Beck, P. Prosser, and R.J. Wallace. Variable ordering heuristics show promise. In *Proceedings of CP-2004*, LNCS 3258, pages 711–715, 2004.
2. C.P. Gomes, C. Fernández, B. Selman, and C. Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10(4):317–337, 2005.
3. C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Automated Reasoning*, 24(1/2):67–100, 2000.
4. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
5. T. Hulubei and B. O'Sullivan. The impact of search heuristics on heavy-tailed behaviour. *Constraints*, 11(2–3):157–176, 2006.
6. B.M. Smith and S.A. Grant. Trying harder to fail first. In *Proceedings of ECAI-1998*, volume 14, pages 249–253, 1998.

Heavy-Tailed Runtime Distributions: Heuristics, Models and Optimal Refutations[⋆]

Tudor Hulubei and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
tudor@hulubei.net, b.osullivan@cs.ucc.ie

1 Introduction

We perform an in-depth empirical study of runtime distributions associated with a continuum of problem formulations for QWH-10 with 90% holes[1] defined between the points where a formulation is entirely specified in terms of binary inequality constraints to models specified using an increasing number of ALLDIFFERENT global constraints [4]. For each model we study a variety of variable and value ordering heuristics. We compare their runtime distributions against runtime distributions where any mistakes made in search are refuted optimally [2], and make the following observations:

1. For the problems considered, *variations in the heuristics used have a far more significant effect on hybrid models* (i.e. models using both binary and global constraints) than they do on purely binary models.
2. While algorithms tend to perform better on hybrid models, a straight line can still be observed in a log-log plot of their runtime distributions, even when mistakes are refuted optimally. In other words, *runtime distributions of hybrid models can remain inherently heavy-tailed* [3].
3. *Models using global constraints are not always better than purely binary models.* We encountered configurations where increasing the number of global constraints used to enforce distinct values on rows and columns (and removing the corresponding sets of binary constraints) does not lead to a monotonic decrease in search effort. *The discrepancy all but disappeared when we looked at the corresponding (quasi-)optimal refutations* for the exact same configuration.
4. With the exception of a few unusual cases, using global constraints did improve search performance and, when that occured, the *refutations encountered for hybrid models were much closer to their corresponding optimal* than for the binary model.
5. For the problems considered, the variables used in refuting a mistake usually represent only a small fraction of the variables still uninstantiated at the time the mistake was made, *yet this small subset of variables can most of the time be re-ordered to refute the mistake optimally.*

[⋆] This work was supported by Science Foundation Ireland (Grant 00/PI.1/C075). We thank the Boole Centre For Research in Informatics for providing access to their Beowulf cluster.

[1] We choose under-constrained problems in order to study heavy-tailed runtime distributions [1].

2 Experiments

Our experiments were performed on satisfiable QWH-10 problem instances with 90% random balanced holes, and included 4 variable ordering heuristics: min-domain, min-dom/ddeg[2], brelaz and min-dom/wdeg, and 3 value ordering heuristics: random, min-conflicts and max-conflicts, totalling 12 algorithms (we broke ties randomly). Most binary instances were too difficult to solve using random variable orderings or variable ordering anti-heuristics, which is why these heuristics have not been included.

We began our study of various ways to model a CSP by encoding these problems using only binary constraints (propagated using MAC) and then gradually replacing the binary constraints used to enforce distinct cells on rows and columns with equivalent n-ary ALLDIFFERENT constraints (which propagate generalised arc-consistency). In addition to the binary model, we represented QWH-10 using 3 different hybrid models by randomly selecting 2, 4, and 8 (out of the 20 possible) rows and/or columns and replacing their corresponding binary constraints with a single n-ary ALLDIFFERENT constraint. We will use $hybrid = X$ to denote a certain model, with X being 0 for the binary case and 2, 4, or 8 for the others (16 and 20 are too easy).

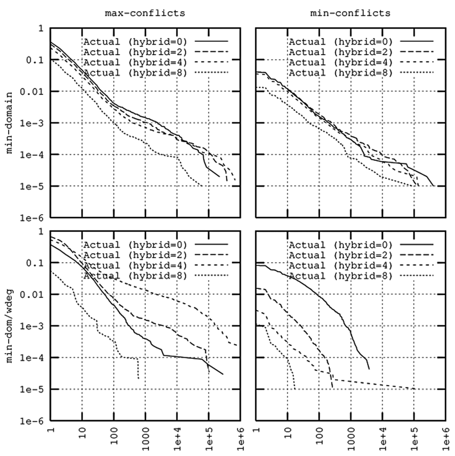
We refer to a *mistake point* [2] as an assignment that cannot lead to a solution even though one existed before that assignment was made. An *actual refutation* is the search tree corresponding to a mistake, as obtained by some algorithm, with the *optimal refutation* for that mistake corresponding to a search tree of minimum size. Finally, the *quasi-optimal refutation* is the smallest refutation whose height does not exceed that of the actual refutation.

3 Results

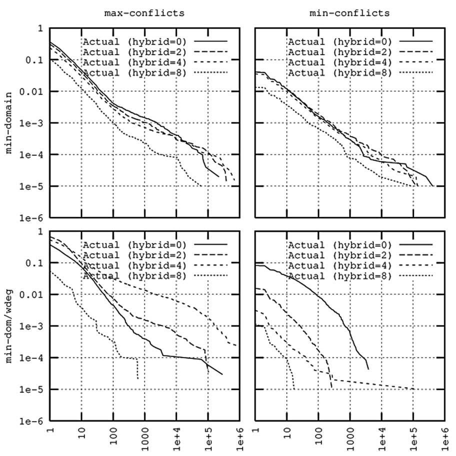
The plots in Figures 1(a) and 1(b) show the actual and (quasi-)optimal runtime distributions of our 12 algorithms on the binary and hybrid models. We use the term *shorter* to refer to refutations that are either optimal, quasi-optimal, or simply the shortest improved refutations we could find that were smaller than the corresponding actual refutations, and the term *restricted shorter* to denote the smallest refutations we could find when the search for optimal refutations was restricted to the variables involved in the actual refutation. We use the term *cumulative effort* to refer to the effort required to refute all the mistakes encountered in a given instance.

Figure 1(a) shows the runtime distributions of our algorithms on the binary and hybrid models. MAC+min-conflicts+min-dom/wdeg is the only algorithm that succeeds in eliminating heavy tails for the binary model [3] and keeps doing so as we add more global constraints, while all other algorithms remain heavy-tailed for all hybrid models (hybrid=X with $X \leq 8$). Moreover, Figure 1(b) shows that, for some algorithms, heavy tails do not disappear even when the mistakes encountered are refuted optimally. In other words, for almost all hybrid models studied, even if we were able to use an oracle to refute insoluble subtrees optimally, for some combinations of heuristics we would still see heavy tails (*inherent heavy-tailedness*).

[2] We abbreviate dynamic-degree as ‘ddeg’ and weighted-degree as ‘wdeg’.

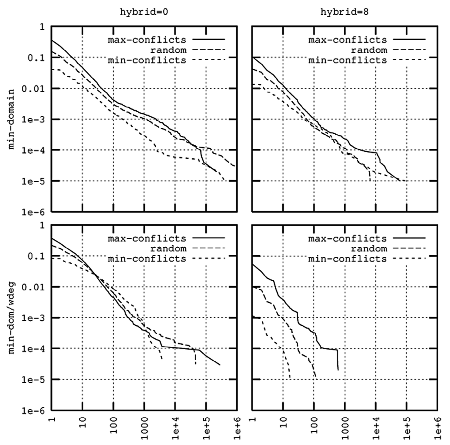


(a) *Actual* refutation effort.

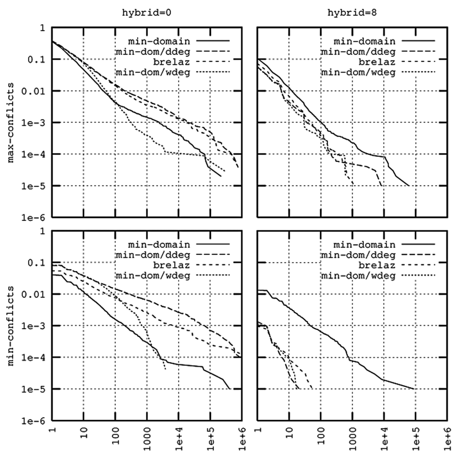


(b) *Shorter* refutation effort.

Fig. 1. Complement of the CDF (y-axis) of the actual (left) and shorter (right) effort (x-axis). We vary the value ordering across columns and variable ordering across rows.



(a) Binary and hybrid=8 models using a poor (min-domain) and a good (min-dom/wdeg) variable ordering (various value orderings).



(b) Binary and hybrid=8 models using a poor (max-conflicts) and a good (min-conflicts) value ordering (various variable orderings).

Fig. 2. Complement of the CDF (y-axis) of the cumulative *actual* effort (x-axis)

Figures 1(a) and 1(b) also show that as our models become more sophisticated through the addition of global constraints, the slopes of the runtime distributions decrease to the point they can no longer be considered heavy-tailed. However, more sophisticated models do not always lead to better performance. Combining max-conflicts with min-dom/wdeg performs worse for the hybrid=4 model than for the binary model

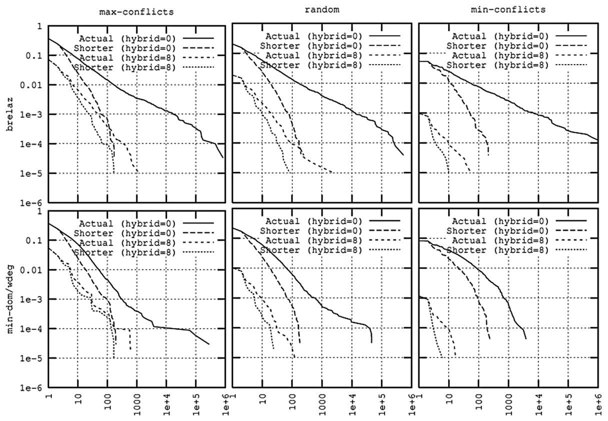
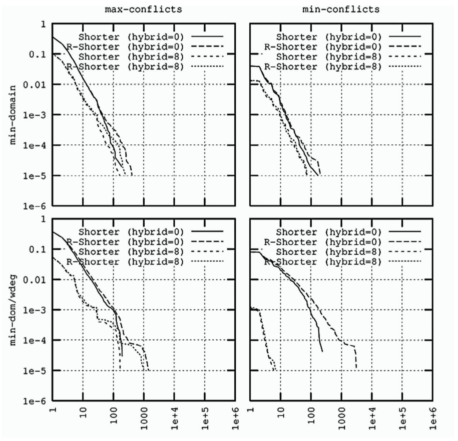
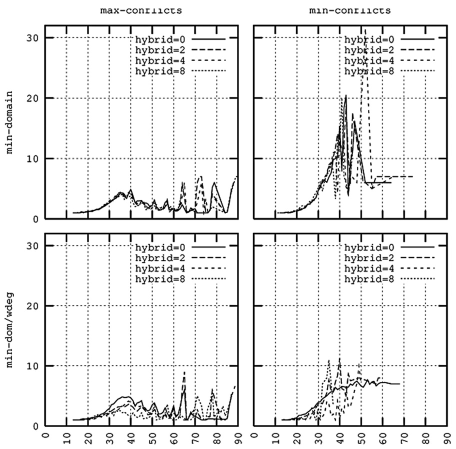


Fig. 3. Comparison of the complement of the CDF (y-axis) for the cumulative *actual* and *shorter* effort (x-axis) for the binary and hybrid=8 models



(a) Comparison of the complement of the CDF (y-axis) for the cumulative *shorter* and *restricted shorter (R-Shorter in the plots)* (x-axis) refutations for the binary and hybrid=8 models.



(b) Average number of variables involved in the *actual* refutations (y-axis) as a function of the number of variables still uninstantiated at the time a mistake was made (x-axis).

Fig. 4. Statistics on restricted refutations

(hybrid=0), as can be seen in Figure 1(a). In the corresponding optimal refutations, while the binary model still outperforms the hybrid=4 model, it only does so by a very small margin. Also, it seems that more sophisticated models can benefit more from good ordering heuristics than the equivalent binary models.

Figure 2(a) shows that the performance improvements due to better value orderings heuristics become significant when a good variable ordering (min-dom/wdeg) is applied to the hybrid=8 model. Similarly, Figure 2(b) shows that the performance improvements due to better variable ordering heuristics become significant when a good value ordering heuristic (min-conflicts) is applied to the hybrid=8 model. These improvements are closely correlated with the runtime distribution of the (quasi-)optimal refutations (not shown for lack of space). Hardly any difference can be observed for the binary models. This analysis suggests that heuristics can infer more from more sophisticated models.

Figure 3 shows another advantage of using global constraints: compared with binary models, actual refutations are closer to optimality. Finally, Figure 4(a) shows that for the binary and hybrid=8 models, the (quasi-)optimal and restricted (quasi-)optimal refutations have almost identical runtime distributions[3]. Interestingly, the average number of variables involved in the actual refutations is only a small fraction of the total number of variables still uninstantiated when the mistake was made (Figure 4(b)). These observations suggest that all the variable ordering heuristics studied here select a very small subset of the remaining uninstantiated variables that could be re-ordered to obtain an optimal refutation. What differentiates a good heuristic from a poor one is the ability to select those variables in an order that minimises the size of the refutation.

4 Conclusions

We have shown empirically that for QWH-10, variations in heuristics have a greater effect on formulations involving a mix of binary and global constraints than on purely binary models. Models using global constraints are not always better than purely binary models. We have also shown that the small subset of variables used by a heuristic to refute a mistake can be *re-ordered* to obtain an almost optimal refutation. This raises the question of why heuristics select the right variables, but fail to find better refutations.

References

1. C.P. Gomes, C. Fernández, B. Selman, and C. Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10(4):317–337, 2005.
2. T. Hulubei and B. O'Sullivan. Optimal refutations for constraint satisfaction problems. In *Proceedings of IJCAI-2005*, pages 163–168, 2005.
3. T. Hulubei and B. O'Sullivan. The impact of search heuristics on heavy-tailed behaviour. *Constraints*, 11(2–3):157–176, 2006.
4. J. C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI*, pages 362–367, 1994.

[3] Due to time and technical constraints, we ran the *quasi-optimal* experiment and *restricted* experiments in parallel, generating potentially different instances, which explains why occasionally the restricted effort appears to be less than the quasi-optimal.

An Extension of Complexity Bounds and Dynamic Heuristics for Tree-Decompositions of CSP

Philippe Jégou, Samba Ndojh Ndiaye, and Cyril Terrioux

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 France
{philippe.jegou, samba-ndojh.ndiaye, cyril.terrioux}@univ-cezanne.fr

Abstract. This paper deals with methods exploiting tree-decomposition approaches for solving constraint networks. We consider here the practical efficiency of these approaches by defining five classes of variable orders more and more dynamic which guarantee time complexity bounds from $O(exp(w+1))$ to $O(exp(2(w+k)))$, with w the "tree-width" of a CSP and k a constant. Finally, we assess practically their relevance.

1 Introduction

The CSP formalism (Constraint Satisfaction Problem) offers a powerful framework for representing and solving efficiently many problems. An instance of CSP is defined by a tuple (X, D, C) where X is a set of n variables, taking their values in finite domains from D, and being subject to constraints from C. Given an instance, the CSP problem consists in determining if there is an assignment of each variable which satisfies each constraint. This problem is NP-complete. In this paper, without loss of generality, we only consider binary constraints (i.e. constraints which involve two variables). So, the structure of a CSP can be represented by the *constraint graph* $G = (X, C)$. The usual approach for solving CSP (Backtracking), has an exponential theoretical time complexity in $O(exp(n))$. To improve this bound, structural methods like *Tree-Clustering* [1] were proposed (see [2] for a survey and a theoretical comparison of these methods). They are based on particular features of the instance like the "tree-width" of the constraint graph (denoted w). The *tree-width* w of G is the minimal width over all the tree-decompositions of G [3]. A *tree-decomposition* of G is a pair (E, T) where $T = (I, F)$ is a tree with nodes I and edges F and $E = \{E_i : i \in I\}$ a family of subsets of X, such that each subset (called cluster) E_i is a node of T and verifies: (i) $\cup_{i \in I} E_i = X$, (ii) for each edge $\{x, y\} \in C$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$, and (iii) for all $i, j, k \in I$, if k is in a path from i to j in T, then $E_i \cap E_j \subseteq E_k$. The width of a tree-decomposition (E, T) is equal to $max_{i \in I} |E_i| - 1$. Recent studies (e.g. [4]) integrate as quality parameter for a decomposition, its efficiency for solving the considered CSP. This paper deals with the question of an efficient use of the considered decompositions. We focus on the BTD method (Backtracking on tree-decomposition [5]) which seems to be the most effective method proposed until now within the framework of these structural methods. Indeed, most of works based on this approach only present theoretical results, except [6,5]. BTD proceeds by an enumerative search guided by a static pre-established partial order induced by a tree-decomposition of the constraint graph. This permits to bound its time complexity by $O(exp(w+1))$, while its space complexity is $O(n.s.d^s)$ with s the size of the largest minimal separators of the graph. Since the efficiency of dynamic variable orders is known, we propose five classes of orders which exploit dynamically the tree-decomposition and guarantee time complexity bounds. Then we define several heuristics for each class.

In section 2, we define the classes and heuristics to compute their orders. Section 3 is devoted to experimental results and conclusions.

2 Classes of Orders and Heuristics

Even though, the basic version of BTD uses a compatible static variable ordering, we prove here by defining the following classes that it is possible to consider more dynamic orders without loosing the complexity bounds. These orders are in fact provided by the cluster order and the variable ordering inside each cluster. Firstly, we give the definition of a generalized tree-decomposition [7]. The set of directed k-covering tree-decompositions of a tree-decomposition (E, T) of G with E_1 its root cluster and k a non nil positive integer, is defined by the set of tree-decompositions (E', T') of G that verify: (i) $E_1 \subset E'_1$, E'_1 the root cluster of (E', T'), (ii) $\forall E'_i \in E'$, $E'_i \subset E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_K}$, with $E_{i_1} \ldots E_{i_K}$ a path in T, and (iii) $|E'_i| \leq w^+ + k$, where $w^+ = max_{E_i \in E}|E_i|$. Given a CSP and a tree-decomposition of its constraint graph, we define:

– **Class 1.** Enumerative static compatible order.
– **Class 2.** Static compatible cluster order and dynamic variable order in the clusters.
– **Class 3.** Dynamic compatible cluster order and dynamic variable order in the clusters.
– **Class 4.** *Class 3* order on a directed k-covering tree-decomposition of the tree-decomposition.
– **Class 5.** *Class 3* order on a set of directed k-covering tree-decompositions of the tree-decomposition.
– **Class ++.** Enumerative dynamic order.

The defined classes form a hierarchy since we have: *Class 1* $\subset$ *Class 2* $\subset$ *Class 3* $\subset$ *Class 4* $\subset$ *Class 5* $\subset$ *Class ++*. The *Class ++* gives a complete freedom, but it does not guarantee time complexity bounds, contrary to the *Class 3*.

Theorem 1. *Let the enumerative order be in the Class 3, the time complexity of BTD is* $O(exp(w+1))$.

The properties of the *Class 3* offer the possibility to choose any cluster to visit next since the variables on the path from the root cluster to that cluster are already assigned. And in each cluster, the variable ordering is totally free. The definitions of the *Class 4* and *Class 5* enforce the order of one assignment to be in the *Class 3*. So we derive natural theorems:

Theorem 2. *Let the enumerative order be in the Class 4 with constant k, the time complexity of BTD is* $O(exp(w^+ + k))$.

Theorem 3. *[7] Let the enumerative order be in the Class 5, the time complexity of BTD is* $O(exp(2(w^+ + k)))$.

We define many heuristics to compute orders in the Classes proposed here and, by lack of place, we only present the more efficient ones:

- $minexp(A)$: this heuristic is based on the expected number of partial solutions of clusters [8] and on their size. It chooses as root cluster one which minimizes the ratio between the expected number of solutions and the size of the cluster. It allows to start the exploration with a large cluster having few solutions.
- $size(B)$: we have here a local criteria: we choose the cluster of maximum size as root cluster
- $minexp_s(C)$: this heuristic is similar to *minexp* and orders the son clusters according to the increasing value of their ratio.
- $minsep_s(D)$: we order the son clusters according to the increasing size of their separator with their parent.
- $nv(E)$: we visit first the son cluster where appears the next variable in the variable order among the variables of the unvisited son clusters.
- $minexp_{sdyn}(F)$: the next cluster to visit minimizes the ratio between the current expected number
- $nv_{sdyn}(G)$: We visit first the son cluster where appears the next variable in the variable order among the variables of the unvisited son clusters.

Inside a cluster, we use min domain/degree heuristic for choosing the next variable (static version mdd_s for class 1 and dynamic mdd_{dyn} for the other classes).

3 Experimental Study and Discussion

Applying a structural method on an instance generally assumes that this instance presents some particular topological features. So, our study is performed on random partial structured CSPs described in [7]. All these experimentations are performed on a Linux-based PC with a Pentium IV 3.2GHz and 1GB of memory. For each class, the presented results are the average on instances solved over 50. We limit the runtime to 30 minutes. Above, the solver is stopped and the involved instance is considered as unsolved. In the table, the letter M means that at least one instance cannot be solved because it requires more than 1GB of memory. We use MCS [9] to compute tree-decompositions because it obtains the best results

Table 1. Parameters w^+ and s of the tree-decomposition and runtime (in s) on random partial structured CSPs with *mdd* for class 1 and mdd_{dyn} for classes 2, 3 and 4

CSP	w^+	s	Class 1		Class 2		Class 3		Class 4			
			B	A	B	A	A	B	B	A	A	B
(n, d, w, t, s, n_c, p)			D	C	D	C	F	G	D	C	F	G
$(150, 25, 15, 215, 5, 15, 10)$	13.0	12.2	9.31	28.12	3.41	2.52	2.45	5.34	2.75	2.17	2.08	2.65
$(150, 25, 15, 237, 5, 15, 20)$	12.5	11.9	9.99	5.27	5.10	2.47	1.99	5.47	2.58	1.76	1.63	2.97
$(150, 25, 15, 257, 5, 15, 30)$	12.1	11.4	13.36	27.82	3.38	5.06	4.97	3.55	1.41	1.05	1.13	1.30
$(150, 25, 15, 285, 5, 15, 40)$	11.5	10.6	3.07	8.77	1.13	0.87	1.27	1.17	1.67	0.39	0.63	1.75
$(250, 20, 20, 107, 5, 20, 10)$	17.8	16.9	54.59	57.75	15.92	12.39	12.14	14.93	10.18	7.75	7.34	10.26
$(250, 20, 20, 117, 5, 20, 20)$	17.2	16.5	55.39	79.80	23.38	14.26	13.25	24.14	10.05	8.81	8.39	10.34
$(250, 20, 20, 129, 5, 20, 30)$	16.8	15.8	26.21	21.14	7.23	5.51	6.19	7.84	33.93	4.61	4.41	34.20
$(250, 20, 20, 146, 5, 20, 40)$	15.9	15.2	44.60	30.17	26.24	3.91	4.51	17.99	11.38	3.17	3.17	10.63
$(250, 25, 15, 211, 5, 25, 10)$	13.0	12.3	28.86	38.75	15.33	11.67	13.37	18.12	5.86	7.71	6.65	6.44
$(250, 25, 15, 230, 5, 25, 20)$	12.8	11.9	20.21	34.47	8.60	7.12	14.84	19.47	4.19	3.94	3.36	6.81
$(250, 25, 15, 253, 5, 25, 30)$	12.3	11.8	11.36	16.91	5.18	11.13	5.14	5.26	2.80	3.71	3.52	3.06
$(250, 25, 15, 280, 5, 25, 40)$	11.8	11.1	7.56	32.74	3.67	16.32	17.49	4.91	4.03	1.40	1.26	3.55
$(250, 20, 20, 99, 10, 25, 10)$	17.9	17.0	M	M	M	M	M	M	66.94	63.15	62.99	66.33
$(500, 20, 15, 123, 5, 50, 10)$	13.0	12.5	12.60	13.63	7.01	8.08	7.31	7.54	5.48	4.50	4.41	5.86
$(500, 20, 15, 136, 5, 50, 20)$	12.9	12.1	47.16	19.22	25.54	23.49	27.01	15.11	4.86	4.92	3.94	5.24

in the study performed in [4] on triangulation algorithms to compute a good tree-decomposition w.r.t. CSP solving. FC and MAC are often unable to solve several instances of each class within 30 minutes.

Table 1 shows the runtime of BTD with several heuristics of Classes 1, 2, 3 and 4. For Class 5, we cannot get good results and then, the results are not presented. Also it presents the width of the computed tree-decompositions and the maximum size of the separators. Clearly, we observe that Class 1 orders obtain poor results. This behaviour is not surprising since static variable orders are well known to be inefficient compared to dynamic ones. A dynamic strategy allows to make good choices by taking in account the modifications of the problem during search. That explains the good results of Classes 2 and 3 orders. The results show as well the crucial importance of the root cluster choice since each heuristic of the Classes 2 and 3 has a dramatic runtime on an average of 4 instances over all instances of all classes because of a bad choice of root cluster. The memory problems marked by M can be solved by using a *Class 4* order with the *sep* heuristic for grouping variables (we merge cluster whose intersection is greater than a value s_{max}). Table 1 gives the runtime of BTD for this class with $s_{max} = 5$. When we analyze the value of the parameter k, we observe that in general, that its value is limited (between 1 to 6). Yet, for the CSPs $(250, 20, 20, 99, 10, 25, 10)$, the value of k is near 40, but this high value allows to solve them.

The heuristics improve very significantly their results obtained for the Classes 2 and 3. The impact of the dynamicity is obvious. *minexp* and *nv* heuristics solve all the instances except one due to a bad root cluster choice, *size* solves all the instances. Except this unsolved instance, *minexp* obtains very promising results. The son cluster ordering has a limited effect because the instances considered have a few son clusters reducing the possible choices and so their impact. The best results are obtained by $minexp + minexp_{sdyn}$, but $size + minsep_s$ obtains often similar results and succeed in solving all instances in the *Class*

4. The calculus of the expected number of solution assumes that the problem constraints are independent, what is the case for the problems considered here. Thus, $size + minsep$ may outperform $minexp + minexp_{sdyn}$ on real-world problems which have dependent constraints.

These experiments highlight the importance of dynamic orders and make us conclude that the Class 4 gives the best variable orders w.r.t CSP solving with a good value of k. Merging clusters with k less than 5 decreases the maximal size of separator and leads to an important reduction of the runtime.

To summarize, we aim to improve the practical interest of the CSP solving methods based on tree-decompositions. This study takes now on importance for solving hard instances with suitable structural properties since they are seldom solved by enumerative methods like FC or MAC. We defined classes of variable orders which guarantee good theoretical time complexity bounds. A comparison of these classes with relevant heuristics w.r.t. CSP solving, points up the importance of a dynamic variable ordering. Indeed the best results are obtained by *Class 4* orders because they give more freedom to the variable ordering heuristic while their time complexity is $O(exp(w+k))$ where k is a constant to parameterize. Note that for the most dynamic class (the Class 5), we get a time complexity in $O(exp(2(w+k)))$ which should be too large to expect a practical improvement. Then, for *Class 4*, we aim to exploit better the problem features to improve the computing of k. This study will be pursued on the optimization problem.

References

1. R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.
2. G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124:343–282, 2000.
3. N. Robertson and P.D. Seymour. Graph minors II: Algorithmic aspects of treewidth. *Algorithms*, 7:309–322, 1986.
4. P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proceedings of CP*, pages 777–781, 2005.
5. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
6. G. Gottlob, M. Hutle, and F. Wotawa. Combining hypertree, bicomp and hinge decomposition. In *Proceedings of ECAI*, pages 161–165, 2002.
7. P. Jégou, S. N. Ndiaye, and C. Terrioux. Strategies and heuristics for exploiting tree-decompositions of constraint networks. In *Proceedings of WIGSK*, 2006.
8. B. Smith. The Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In *Proceedings of ECAI*, pages 100–104, 1994.
9. R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13 (3):566–579, 1984.

Clique Inference Process for Solving Max-CSP[⋆]

Mohand Ou Idir Khemmoudj and Hachemi Bennaceur

LIPN-CNRS UMR 7030, 99 Av J-B. Clément, 93430 Villetaneuse France
{MohandOuIdir.Khemmoudj, Hachemi.Bennaceur}@lipn.univ-paris13.fr

Abstract. In this paper we show that the clique concept can be exploited in order to solve Max-CSP. We present a clique inference process which leads to construct linear systems useful for computing new lower bounds. The clique inference process is introduced in the PFC-MPRDAC [5] algorithm and the obtained algorithm is called PFC-MPRDAC+CBB (CBB for Clique Based Bound). The carried out experiments have shown that PFC-MPRDAC+CBB leads to obtain very encouraging results.

1 Introduction

A binary Constraint Satisfaction Problem (CSP) is defined by a triplet (X, D, C) where $X = \{X_1, X_2, ..., X_n\}$ is a set of n variables, $D = \{D_1, D_2, ..., D_n\}$ is a set of n domains where D_i is a set of d_i possible values for X_i, C is a set of e constraints in which a constraint C_{ij} involves variables X_i and X_j and assigns costs to assignments to variables X_i and X_j (namely, $C_{ij} : D_i \times D_j \rightarrow \{0, 1\}$). A pair (v_k, v_l) satisfies C_{ij} if $C_{ij}(v_k, v_l) = 0$. A solution of the CSP is a total assignment satisfying each of the constraints. In some cases the CSP may be over-constrained and it may be of interest to find a total assignment violating the minimum number of constraints. This problem is usually referred as Max-CSP and can be represented by a graph G whose vertices represent the values of the variables and the edges represent the not permitted pairs of values. A union of subsets, $E_{i_1} \subseteq D_{i_1}, E_{i_2} \subseteq D_{i_2}, ..., E_{i_m} \subseteq D_{i_m}$, forms a clique of G if and only if for each couple $(j_1, j_2) \in \{i_1, i_2, ..., i_m\}^2$ such that $E_{j_1} \neq \emptyset$ and $E_{j_2} \neq \emptyset$ we have $C_{j_1 j_2} \in C$ and $C_{j_1 j_2}(v_k, v_l) = 1 \ \forall (v_k, v_l) \in E_{j_1} \times E_{j_2}$. We will say that a clique is binary if it is a union of two subsets of two domains of the CSP.

In this work, we propose a linear formulation for any binary clique and show how it can be used to propose a new linear models useful for solving Max-CSP. We then propose a clique inference process which leads to construct interesting linear systems useful for computing good lower bounds.

2 Clique Based Linear Models for Max-CSP

In this section we propose a linear formulation for binary clique, and show how it can be exploited to construct linear systems useful for computing new lower

[⋆] This work is supported in part by the French Electricity Board (EDF).

bounds. We begin the process modeling by introducing for each variable X_i, d_i binary variables x_i^k ($k \in [1, d_i]$). We also introduce for each constraint C_{ij} a binary variable η_{ij}. We will denote by x the array of components x_i^k, by η the array of components η_{ij} and by S the set of all solutions of the Max-CSP : $x \in S \Leftrightarrow \sum_{v_k \in D_i} x_i^k = 1 \forall i \in [1, n]$. A binary clique $\Gamma_{ij} = E_i \cup E_j$ $(E_i \subseteq D_i, E_j \subseteq D_j)$ can be formulated as follows : $\psi(E_i, E_j) = \sum_{v_k \in E_i} x_i^k + \sum_{v_l \in E_j} x_j^l \leq 1 + \eta_{ij}$. This linear constraint expresses the fact that if the variables X_i and X_j are respectively assigned to values $v_k \in E_i$ and $v_l \in E_j$ then the binary constraint C_{ij} is violated (η_{ij} must be equal to 1). Thus, for each set of m binary cliques, $\Gamma = \{\Gamma_{ij}^t = E_i^t \cup E_j^t : t = 1...m,\ 1 \leq i < j \leq n\}$ we can define the following optimization problem :

$$IP(\Gamma) \begin{cases} \min \eta.\mathbb{1}_e & \\ s.t: & \psi(E_i^t, E_j^t) \leq 1 + \eta_{ij}\ t = 1...m \\ & x \in S \end{cases}$$

where $\mathbb{1}_e$ is a vector of e 1, e is the constraint number of the Max-CSP.

Each lower bound of the linear system $IP(\Gamma)$ is a lower bound of the Max-CSP. Moreover, if Γ contains, for each constraint $C_{ij} \in C$ and for each pair $(v_k, v_l) \in D_i \times D_j : C_{ij}(v_k, v_l) = 1$, at least one clique Γ_{ij}^t involving v_k and v_l then the linear system $IP(\Gamma)$ is equivalent to the Max-CSP [3].

The clique inference schema which we use to define sets of cliques associates for a constraint $C_{ij} \in C$ and a subset E_i of D_i the maximal binary clique $\Gamma_{ij}(E_i) = \phi_{ji}(\phi_{ij}(E_i)) \cup \phi_{ij}(E_i)$, where $\phi_{ij}(E_i) = \{v_l \in D_j : C_{ij}(v_k, v_l) = 1\ \forall v_k \in E_i\}$ and $\phi_{ji}(\phi_{ij}(E_i)) = \{v_k \in D_i : C_{ij}(v_k, v_l) = 1\ \forall v_l \in \phi_{ij}(E_i)\}$. A particular set of cliques which we consider is $\Gamma^0 = \{\Gamma_{ij}(D_i), \Gamma_{ji}(D_j),\ C_{ij} \in C\}$.

Theorem 1. *To Γ^0 correspond the linear system :*

$$IP(\Gamma^0) \begin{cases} min \sum_{C_{ij} \in C} \eta_{ij} & \\ st: \psi(D_i, \phi_{ij}(D_i)) \leq 1 + \eta_{ij} & \forall C_{ij} \in C \\ \quad\ \ \psi(\phi_{ji}(D_j), D_j) \leq 1 + \eta_{ij} & \forall C_{ij} \in C \\ \quad\ \ x \in S & \end{cases}$$

The value $V(LP(\Gamma^0))$ of the continuous relaxation of $IP(\Gamma^0)$ is greater or equal to the lower bounds based on the Directed Arc Consistency [8], Reversible Directed Arc Consistency [4] and Weighted Arc Consistency [1] counts[1].

3 A Clique Inference Process

This section presents a local search process constructing p clique sets $\Gamma^1, \Gamma^2, ...,$ Γ^p in such way that $LR(\mathbb{1}, \Gamma^1) < LR(\mathbb{1}, \Gamma^2) < ... < LR(\mathbb{1}, \Gamma^{p-1}) = LR(\mathbb{1}, \Gamma^p)$, where $LR(\mathbb{1}, \Gamma^q)$ $(1 \leq q \leq p)$ is the value of the Lagrangean relaxation which

[1] A proof of the theorem is given in [3].

we obtain if we dualize the clique constraints of $IP(\Gamma^q)$ by using a vector of Lagrangean multipliers with all components equal to 1.

Let us denote by $LR(\lambda, \Gamma^0)$ the value of the Lagrangean relaxation which we obtain if we dualize the clique constraints of $IP(\Gamma^0)$ by using the vector of Lagrangean multipliers $\lambda = (\lambda_{ij}, \lambda_{ji})_{c_{ij} \in C} : \lambda_{ij} \geq 0$. Multipliers $\lambda_{ij}, \lambda_{ji}$ are respectively associated to the constraints $\psi(D_i, \phi_{ij}(D_i)) \leq 1$ and $\psi(\phi_{ji}(D_j), D_j) \leq 1$.

First, the dual Lagrangean problem $\max\limits_{\lambda \geq 0} LR(\lambda, \Gamma^0)$ is solved in an approximated way by using the greedy local search algorithm **GreedyOpt** [4]. Then,

GreedyOpt

```
1. STOP ← $false$, $\forall C_{ij} \in C, \lambda_{ij} = 1, \lambda_{ji} = 0$
2. WHILE(! STOP)
  a. STOP ← $true$
  b. $\forall C_{ij} \in C$
    i. $\lambda' \leftarrow \lambda, \lambda'_{ij} \leftarrow \lambda_{ji}, \lambda'_{ji} \leftarrow \lambda_{ij}$
    ii. if $(LR(\lambda', \Gamma^0) > LR(\lambda, \Gamma^0))$ then $\{\lambda \leftarrow \lambda'$, STOP ← $false\}$
```

the process initializes Γ^1 by assigning to each constraint $C_{ij} \in C$ one of the cliques $\Gamma_{ij}(D_i)$ or $\Gamma_{ji}(D_j)$, according to whether $(\lambda_{ij}, \lambda_{ji})$ is given equal to (1,0) or (0,1) by **GreedyOpt**. It maintains two counters for each value v_k of each variable X_i : the clique count cc_i^k and the reduced cost $rc_i^k = cc_i^k - \min\limits_{v_{k'} \in D_i} cc_i^{k'}$. The clique count cc_i^k is initialized by the number of different cliques in Γ^1 involving the value v_k of X_i. The current lower bound is $LR(\mathbb{1}, \Gamma^1) = \sum\limits_{i=1}^{n} \min\limits_{v_k \in D_i} cc_i^k - e$. The process continues in order to improve this bound. Consider the following notations :

– $\Gamma^q = \{\Gamma_{ij}^q, C_{ij} \in C\}$: the clique set generated at the q-th process step;
– $\Gamma_{ij}^q = E_{ij}^q \cup E_{ji}^q$: the associated clique to the constraint C_{ij} ;
– $MIN(E_{ij}^q) = \{v_k \in E_{ij}^q : rc_i^k = 0\}$: values in E_{ij}^q with a null reduced cost;
– $MIN(E_{ji}^q) = \{v_l \in E_{ji}^q : rc_j^l = 0\}$: values in E_{ji}^q with a null reduced cost;

ClimbOpt

```
1. $\Gamma^q \leftarrow \Gamma^{q-1}$
2. $\forall C_{ij} \in C$
   if $(\lambda_{ij} = 1)$ then $\Gamma^q \leftarrow \Gamma^q \setminus \Gamma_{ij}^q \cup \Gamma_{ij}(MIN(E_{ij}^{q-1}))$
   else $\Gamma^q \leftarrow \Gamma^q \setminus \Gamma_{ij}^q \cup \Gamma_{ji}(MIN(E_{ji}^{q-1}))$
3. $NewLB = V(LR(\mathbb{1}, \Gamma^q))$
```

The sets $\Gamma^q, 2 \leq q \leq p$ are obtained by the **ClimbOpt** procedure which replaces the cliques of Γ^{q-1} (step 2) : each clique Γ_{ij}^{q-1} ($\forall C_{ij} \in C$) is replaced by one of the cliques $\Gamma_{ij}(MIN(E_{ij}^{q-1}))$ or $\Gamma_{ji}(MIN(E_{ji}^{q-1}))$ according to whether $(\lambda_{ij}, \lambda_{ji})$ is equal to (1,0) or (0,1). The procedure **ClimbOpt** is executed while it leads to increase the lower bound.

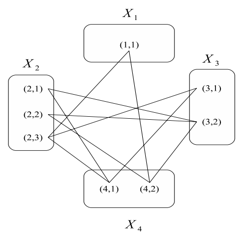


Fig. 1. Example of Max-CSP

Example 1. *Fig. 1 shows an example of Max-CSP with 4 variables* ($X = \{X_1, X_2, X_3, X_4\}$) *and 5 constraints* ($C = \{C_{12}, C_{14}, C_{23}, C_{24}, C_{34}\}$)*. The edges on graph represent the incompatible pairs. When we solve the dual Lagrangean problem* $\max_{\lambda} LR(\lambda, x)$ *corresponding to* Γ^0 *by the* ***GreedyOpt*** *procedure we obtain the solution vector* $\lambda = (1,0,1,0,1,0,1,0,1,0)$*. The set* Γ^1 *is then initialized as follow:* $\Gamma^1 = \{\Gamma_{12}(D_1), \Gamma_{14}(D_1), \Gamma_{23}(D_2), \Gamma_{24}(D_2), \Gamma_{34}(D_3)\} = \{\{(1,1),(2,3)\}, \{(1,1),(4, \quad 2)\}, \{(2,1),(2,2),(2,3)\}, \{(2,1),(2,2),(2,3)\}, \{(3,1),(3,2)\}\}$. *The initial values of the clique counts are given by the two first tables on Fig.2. The corresponding lower bound is* $LR(\mathbb{1}, \Gamma^1) = \sum_{i=1}^{n} \min_{v_k \in D_i} cc_i^k - e = 2 + 2 + 1 - 5 = 0$. *The process continues by executing* ***ClimbOpt****. This leads to construct* Γ^2*. Only the cliques associated to the constraints* C_{23} *and* C_{34} *are replaced by new ones. The clique* $\Gamma_{23}(D_2)$ *is replaced by* $\Gamma_{23}(MIN(E_{23}^1)) = \Gamma_{23}(\{(2,1),(2, \ 2)\}) = \{(2,1),(2,2),(3,2)\}$ *and the clique* $\Gamma_{34}(D_3)$ *is replaced by* $\Gamma_{34}(MIN(E_{34}^1)) = \Gamma_{34}(\{(3,1)\}) = \{(3,1),(4,1)\}$*. These replacements modify the clique counts of some values of variables. The values of the clique counts are now as given by the two last tables on Fig.2 and the new lower bound is* $LR(\mathbb{1}, \Gamma^2) = \sum_{i=1}^{n} \min_{v_k \in D_i} cc_i^k - e = 2 + 2 + 1 + 1 - 5 = 1$.

i	cc_i^1	cc_i^2	cc_i^3	min
1	2	∞	∞	2
2	2	2	3	2

i	cc_i^1	cc_i^2	cc_i^3	min
3	1	1	∞	1
4	0	1	∞	0

i	cc_i^1	cc_i^2	cc_i^3	min
1	2	∞	∞	2
2	2	2	2	2

i	cc_i^1	cc_i^2	cc_i^3	min
3	1	1	∞	1
4	1	1	∞	1

Fig. 2. Initial and final computed clique counts

The corresponding linear system to Γ^2 *is :*

$$IP(\Gamma^2) \begin{cases} min \ \eta_{12} + \eta_{14} + \eta_{23} + \eta_{24} + \eta_{34} & \\ st: \ x_1^1 + x_2^3 & \leq 1 + \eta_{12} \\ \quad\quad x_1^1 + x_4^2 & \leq 1 + \eta_{14} \\ \quad\quad x_2^1 + x_2^2 + x_3^2 & \leq 1 + \eta_{23} \\ \quad\quad x_2^1 + x_2^2 + x_2^3 & \leq 1 + \eta_{24} \\ \quad\quad x_3^1 + x_4^1 & \leq 1 + \eta_{34} \\ \quad\quad x \in S & \end{cases}$$

Since we have $LR(\mathbb{1}, \Gamma^2) > LR(\mathbb{1}, \Gamma^1)$ *the process executes* ***ClimbOpt*** *in order to construct* Γ^3*. This does not lead to improve the current lower bound and the process stops. The computed lower bound is equal to 1.*

4 Conclusion

In this paper, we have shown that the concept of clique can be exploited in order to solve Max-CSP. We have proposed a linear formulation for binary cliques and shown that this formulation can be used to construct linear models useful for solving Max-CSP. We have proposed a clique inference process which leads to construct partial linear systems in order to compute lower bounds. The clique inference process is introduced in the PFC-MPRDAC algorithm and the obtained algorithm is denoted PFC-MPRDAC+CBB (CBB for Clique Based Bound). Our first experimental results [3] have shown that PFC-MPRDAC+CBB outperforms PFC-MPRDAC on the tested random problems.

The sophistication of our algorithm and the study of bonds between clique inference and the stronger local consistency techniques AC^* [7], $FDAC^*$ [6] and $EDAC^*$ [2] remain as future work.

References

1. Affane, M.S., Bennaceur, H.: A Weighted Arc Consistency Technique for MAX-CSP. *ECAI* (1998) 209-213.
2. de Givry, S., Zytnicki, M., Heras, F., Larrosa, J.: Existential arc consistency: Getting closer to full arc consistency in weighted csps. *IJCAI* (2005) 84-89.
3. Khemmoudj, M.I., Bennaceur, H.: Clique Based Lower Bounds for Max-CSP. *Technical report 2006-02*, LIPN (2006).
4. Larrosa, J., Meseguer, P., Schiex, T.: Maintening Reversible DAC for Max-CSP. *Artificial Intelligence* 107(1), (1999) 149-163.
5. Larrosa, J., Meseguer, P.: Partition-Based Lower Bound for Max-CSP. *CP* (1999) 303-315.
6. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted CSP. *IJCAI* (2003) 239-244.
7. Larrosa, J., Schiex, T.: Solving Weighted CSP by Maintaining arc consistency. *Artificial Intelligence* 159(1-2), (2004) 1-26.
8. Wallace, R.: Directed arc consistency preprocessing. Dans Meyer,M., ed., *Selected papers from the ECAI Workshop on Constraint Processing,* 923 dans LNCS. Berlin: Springer. (1994) 121-137.

Global Grammar Constraints

Claude-Guy Quimper[1] and Toby Walsh[2]

[1] School of Computer Science, University of Waterloo, Canada
cquimper@math.uwaterloo.ca
[2] NICTA and UNSW, Sydney, Australia
tw@cse.unsw.edu.au

1 Introduction

Global constraints are an important tool in the constraint toolkit. Unfortunately, whilst it is usually easy to specify when a global constraint holds, it is often difficult to build a good propagator. One promising direction is to specify global constraints via grammars or automata. For example, the Regular constraint [1] permits us to specify a wide range of global constraints by means of a DFA accepting a regular language, and to propagate this constraint specification efficiently and effectively. More precisely, the Regular constraint ensures that the values taken by a sequence of variables form a string accepted by the DFA. In this paper, we consider how to propagate such grammar constraints and a number of extensions.

2 Regular Constraint

Pesant has given a domain consistency algorithm for the Regular constraint based on dynamic programming that runs in $O(ndQ)$ time and space where d is the domain size, Q is the number of states in the DFA, and n is the number of variables. The Regular constraint can be encoded using a simple sequence of ternary constraints. Enforcing GAC on this decomposition achieves GAC on the original Regular constraint. and takes just $O(ndQ)$ time and space. We introduce a second sequence of variables, Q_0 to Q_n to represent the state of the automaton. We then post the sequence of transition constraints $C(X_{i+1}, Q_i, Q_{i+1})$ for $0 \leq i < n$ which hold iff $Q_{i+1} = T(X_{i+1}, Q_i)$ where T is the transition function of the DFA. In addition, we post the unary constraints $Q_0 = q_0$ and $Q_n \in F$. To enforce GAC on such ternary constraints, we can use the table constraint available in many solvers, or primitives like the implication.

One advantage of this encoding is that we have explicit access to the states of the automaton. These can be used, for example, to model the objective function. The states of the automaton also need not be finite integer domain variables but can, for example, be set variables. We can model the open stacks problem in this way. This encoding also works with a non-deterministic finite automaton. Whist NFA only recognize regular languages, they can do so with exponentially fewer states than the smallest DFA. Enforcing GAC on the encoding takes $O(nT)$

time, where T is the number of transitions in the automaton. This number can be exponentially smaller for a NFA compared to a DFA. We can also encode the soft form of the REGULAR constraint [2] in a similar way by introducing a variable whose value is the state of the automaton and the distance from it.

For repeating sequences, we introduce two cyclic forms of the REGULAR constraint. REGULAR$_+(\mathcal{A}, [X_1, \ldots, X_n])$ holds iff the string defined by $X_1 \ldots X_n X_1$ is part of the regular language recognized by $\mathcal{A}$. To enforce GAC on REGULAR$_+$, we can create a new automaton in which states also contain the first value used. We can therefore use this new automaton and a normal REGULAR constraint to enforce GAC on REGULAR$_+$ in $O(nd^2Q)$ time. The REGULAR$_\circ$ constraint ensures that any rotation of the sequence gives a string in the regular language. More precisely, REGULAR$_\circ(\mathcal{A}, [X_1, \ldots, X_n])$ holds iff the strings defined by $X_i \ldots X_{1+(i+n-1 \bmod n)}$ for $1 \leq i \leq n$ are part of the regular language recognized by $\mathcal{A}$. Unfortunately, enforcing GAC on a REGULAR$_\circ$ constraint is NP-hard (reduction from Hamiltonian cycle).

3 CFG Constraint

Another generalization is to context-free languages. We introduce the global grammar constraint CFG(G, $[X_1, \ldots, X_n]$) which ensures that X_1 to X_n form a string accepted by the context-free grammar $\mathcal{G}$. Such a constraint might be useful in a number of applications like bioinformatics or natural language processing.

To achieve GAC on a CFG constraint, we give a propagator based on the CYK parser which requires the context-free grammar to be in Chomsky normal form. The propagator given in Algorithm 1 proceeds in two phases. In the first phase (lines 0 to 7), we use dynamic programming to construct a table $V[i, j]$ with the potential non-terminal symbols that can be parsed using values in the domains of X_i to X_{i+j-1}. $V[1, n]$ thus contains all the possible parsings of the sequence of n variables. In the second phase of the algorithm (lines 9 to 18), we backtrack in the table V and mark each triplet (i, j, A) such that there exists a valid sequence of size n in which A generates the substring of size j starting at i. When the triplet $(i, 1, A)$ is marked, we conclude there is a support for every value $a \in \text{dom}(X_i)$ such that $A \to a \in G$.

Theorem 1. `CYK-prop` *enforces GAC on* CFG$(\mathcal{G}, [X_1, \ldots, X_n])$ *in* $\Theta(|G|n^3)$ *time and* $\Theta(|G|n^2)$ *space.*

Our second propagator is based on the popular Earley chart parser which also uses dynamic programming to parse a context-free language. Whilst this propagator is more complex, it is not restricted to Chomsky normal form, and is often much more efficient than CYK as it parses strings top-down, particularly when the productions are left-recursive. The propagator again uses dynamic programming to build up possible support. Productions are annotated with a "dot" indicating position of the parser. WLOG, we assume a unique starting production $S \to U$. A successful parsing is thus $S \to U\bullet$.

Algorithm 1. CYK-prop(G, $[X_1, \ldots, X_n]$)

```
1  for $i = 1$ to $n$ do
2      $V[i, 1] \leftarrow \{A \mid A \rightarrow a \in G, a \in \mathrm{dom}(X_i)\}$
3  for $j = 2$ to $n$ do
4      for $i = 1$ to $n - j + 1$ do
5          $V[i, j] \leftarrow \emptyset$
6          for $k = 1$ to $j - 1$ do
7              $V[i, j] \leftarrow V[i, j] \cup \{A \mid A \rightarrow BC \in G, B \in V[i, k], C \in V[i + k, j - k]\}$
8  if $S \notin V[1, n]$ then return "Unsatisfiable"
9  mark $(1, n, S)$
10 for $j = n$ downto 2 do
11     for $i = 1$ to $n - j + 1$ do
12         for $A \rightarrow BC \in G$ such that $(i, j, A)$ is marked do
13             for $k = 1$ to $j - 1$ do
14                 if $B \in V[i, k]$, $C \in V[k + k, j - k]$ then
15                     mark $(i, k, B)$
16                     mark $(i + k, j - k, C)$
17 for $i = 1$ to $n$ do
18     $\mathrm{dom}(X_i) \leftarrow \{a \in \mathrm{dom}(X_i) \mid A \rightarrow a \in G, (i, 1, A)$ is marked$\}$
19 return "Satisfiable"
```

Algorithm 2 is the Earley chart parser augmented with the sets S that keep track of the supports for each value in the domains. We use a special data structure to implement these sets S. We first build the basic sets $\{X_i = v\}$ for every potential support $v \in \mathrm{dom}(X_i)$. Once a set is computed, its value is never changed. To compute the union of two sets $A \cup B$, we create a set C with a pointer on A and a pointer on B. This allows to represent the union of two sets in constant time. The data structure forms a directed graph where the sets are the nodes and the pointers are the edges. To enumerate the content of a set S, one can perform a depth-first search. The basic sets $\{X_i = v\}$ that are visited in the search are the elements of S.

Theorem 2. `Earley-prop` *enforces GAC on* CFG$(\mathcal{G}, [X_1, \ldots, X_n])$ *in* $O(|G|n^3)$ *time for an arbitrary context-free grammar, and in* $O(|G|n^3)$ *space.*

4 Related Work

For the REGULAR constraint, a propagation algorithm based on dynamic programming that enforces GAC was given in [1]. Coincidently Beldiceanu, Carlsson and Petit proposed specifying global constraints by means of deterministic finite automaton augmented with counters [3]. Propagators for such automaton are constructed automatically by means of a conjunction of signature and transition constraints. The ternary encodings used here are similar to those proposed

Algorithm 2. Earley-Prop(G, $[X_1, \ldots, X_n]$)

```
1  for $i = 0$ to $n$ do $C[i] \leftarrow \emptyset$
2  $queue \leftarrow \{(s \rightarrow \bullet u, 0, \emptyset)\}$
3  for $i = 0$ to $n + 1$ do
4      for $state \in C[i]$ do push(state,queue)
5      while $queue$ is not empty do
6          $(r, j, S) \leftarrow pop(queue)$
7          $add((r, j, S), C[i])$
8          if $r = (u \rightarrow v\bullet)$ then
9              foreach $(w \rightarrow \ldots \bullet u \ldots, k, T) \in C[j]$ do
10                 $add((w \rightarrow \ldots u\bullet \ldots, k, S \cup T), queue)$
11         else if $i \leq n$ and $r = (u \rightarrow \ldots \bullet v \ldots)$ and $v \in dom(X_i)$ then
12             $add((u \rightarrow \ldots v\bullet \ldots, j, S \cup \{X_i = v\}), C[i + 1])$
13         else if $r = (u \rightarrow \ldots \bullet v \ldots)$ and $non\_terminal(v, G)$ then
14             foreach $v \rightarrow w \in G$ such that $(v \rightarrow \bullet w, i, \emptyset) \notin C[i] \cup queue$ do
15                 $push((v \rightarrow \bullet w, i, \emptyset), queue)$
16     if $C[i] = \emptyset$ then
17         return "Unsatisfiable"
18 if $(s \rightarrow u\bullet, 0, S) \in C[n]$ then
19     for $i = 1$ to $n$ do
20         dom$(X_i) = \{a \mid X_i = a \in S\}$
21 else
22     return "Unsatisfiable"
```

Algorithm 3. add($(a, b, c), q$)

```
1  if $\exists (a, b, d) \in q$ then
2      $q \leftarrow replace((a, b, d), (a, b, c \cup d), q)$
3  else
4      $push((a, b, c), q)$
```

in [3]. However, there are a number of differences. One is that we permit non-deterministic transitions. As argued before, non-determinism can reduce the size of the automaton significantly. In addition, the counters used by Beldiceanu, Carlsson and Petit introduce complexity. For example, they need to achieve pairwise consistency to guarantee global consistency. Pesant encodes a cyclic STRETCH constraint into a REGULAR constraint in which the initial variables of the sequence are repeated at the end, and then dummy unconstrained variables are placed at the start and end [1]. Hellsten, Pesant and van Beek propose a domain consistency algorithm similar to that for the REGULAR constraint [4]. They also showed how to extend it to deal with cyclic STRETCH constraints.

5 Conclusions

We have studied a range of grammar constraints. These are global constraints over a sequence of variables which restrict the values assigned to be a string within a given language. Such constraints are useful in a wide range of scheduling, rostering and sequencing problems. For regular languages, we gave a simple encoding into ternary constraints that can be used to enforce GAC in linear time. Experiments demonstrate that such encodings are efficient and effective in practice. This ternary encoding is therefore an easy means to incorporate this global constraint into constraint toolkits. We also considered a number of extensions including regular languages specified by non-deterministic finite automata, and soft and cyclic versions of the global constraint. For context-free languages, we gave two propagators which enforce GAC based on the CYK and Earley parsers. Experiments show that the propagator based on the CYK parser is faster at the top of the search tree while the propagator based on the Earley parser is faster at the end of the search. This shows some potential for a hybrid propagator. There are many directions for future work. One promising direction is to learn grammar constraints from examples. We can leverage on results and algorithms from grammar induction. For example, it is not possible to learn a Regular constraint from just positive examples.

References

1. Pesant, G.: A regular language membership constraint for finite sequences of variables. In Wallace, M., ed.: Proceedings of 10th International Conference on Principles and Practice of Constraint Programming (CP2004), Springer (2004) 482–295
2. van Hoeve, W.J., Pesant, G., Rousseau, L.M.: On global warming : Flow-based soft global constaints. Journal of Heuristics (2006) To appear.
3. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In Wallace, M., ed.: Proceedings of 10th International Conference on Principles and Practice of Constraint Programming (CP2004), Springer (2004) 107–122
4. Hellsten, L., Pesant, G., van Beek, P.: A domain consistency algorithm for the stratch constraint. In Wallace, M., ed.: Proceedings of 10th International Conference on Principles and Practice of Constraint Programming (CP2004), Springer (2004) 290–304

Constraint Propagation for Domain Bounding in Distributed Task Scheduling

Evan A. Sultanik, Pragnesh Jay Modi, and William C. Regli

Department of Computer Science
Drexel University
3141 Chestnut Street, Philadelphia, PA, USA
{eas28, pmodi, wregli}@cs.drexel.edu

1 Introduction

The coordinated management of inter-dependent plans or schedules belonging to different agents is a complex, real-world problem arising in diverse domains such as disaster rescue, small-team reconnaissance, and security patrolling. The problem is inherently a distributed one; no single agent has a global view and must make local scheduling decisions through collaboration with other agents to ensure a high quality global schedule. A key step towards addressing this problem is to devise appropriate distributed representations. The Coordinators Task Analysis Environmental Modeling and Simulation (C_TÆMS) language is a representation that was jointly designed by several multi-agent systems researchers explicitly for multi-agent task scheduling problems [1,2,3,4]. C_TÆMS is an extremely challenging class of scheduling problem which is able to model the distributed aspects of the problem.

The Distributed Constraint Optimization Problem (DCOP) [5] was devised to model reasoning problems where agents must coordinate to ensure that solutions are globally optimal which makes it a suitable approach for representing multi-agent task scheduling. We propose an approach of problem transformation that converts a significant subclass of C_TÆMS into the DCOP representation and show this subclass of C_TÆMS problems remains computationally difficult. We demonstrate a key advantage of our constraint-based approach is the ability to apply constraint solving techniques from the constraint programming literature. In this paper, we report on experiments using the Adopt algorithm [5] which is guaranteed to find optimal solutions through asynchronous peer-to-peer message passing between agents with worst-case polynomial space requirements. We propose pre-processing techniques to perform domain pruning to significantly increase problem solving efficiency. We demonstrate a 96% reduction in state space size for a set of C_TÆMS problems. This equates to up to a 60% increase in the ability to distributedly and optimally solve C_TÆMS problems in a reasonable amount of time as a result of our constraint-based pre-processing algorithms.

2 Background

A DCOP defines a set of agents that are responsible for assigning a value to their respective variables subject to a set of soft or hard constraints. Each variable in

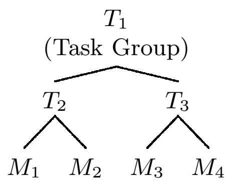


Fig. 1. An example C_TÆMS task hierarchy

the DCOP is assigned to a specific agent, whose job it is to assign the value of the variable. The objective of a DCOP is to have each agent assign values to its associated variables in order to minimize the cost over all constraints.

C_TÆMS is a derivative of the TÆMS modeling language [6] used for representing instances multi-agent scheduling problems. In this paper, we consider a restricted subclass of C_TÆMS[1] that is defined as a tuple containing a set of agents, a set of methods, a set of tasks, and a set of "Non-Local Effects" (NLEs). As shown in Figure 1, the methods and tasks are arranged in a task hierarchy in which the methods are the leaves and the root is a special task known as the "task group." Methods and tasks may have earliest start times, deadlines, and expected durations. NLEs are precedence constraints, *e.g.*, an *enables* NLE requires that method `X` cannot execute before method `Y` has finished. A "schedule" is a grammar within the C_TÆMS specification for defining the chosen start times of methods. A *feasible* schedule is one that satisfies NLEs constraints and *mutex* constraints which require that an agent not execute more than one method at once. Methods render "quality" if chosen to execute; each task has a "Quality Accumulation Function" (QAF) that defines how its children's qualities are propagated up the tree. The objective is to create a feasible schedule that maximizes the quality of the task group.

3 Technical Approach

The basis of our approach is to map a restricted class of C_TÆMS instances to an equivalent DCOP whose solution leads to an optimal schedule. The technical challenge lies in ensuring that the resulting DCOP's solution leads to an optimal schedule. The following section formalizes the approach.

3.1 Mapping C_TÆMS to a DCOP

For each agent in the C_TÆMS instance, create an associated DCOP agent. For each method, create an associated variable. The domains of the variables will contain all possible start times of the method (including an option for the method to forgo execution). In order to encode the mutex constraints, for all agents find all pairs of methods that share that agent and create a hard DCOP constraint

[1] This version differs from the full C_TÆMS by omitting probabilistic aspects of task durations, the *facilitates* and *hinders* NLEs, and non-summation QAFs.

(*i.e.* of infinite cost) for all pairs of equal domain values for the associated variables. This will ensure that an agent may not execute multiple methods at once. NLEs are encoded similarly: for all *enables* NLEs (*i.e.* method X cannot execute before method Y has finished), find all pairs of methods that are in the subtree rooted by the endpoints of the NLE and add a hard DCOP constraint for their associated variables when the NLE is violated. Finally, add one unary soft constraint for all methods' variables as follows: if a method is *not* scheduled to execute, its unary constraint will have a cost equal to the quality that the method *would have* contributed to the task group had it been executed. This mapping will produce a DCOP with $|M|$ variables and worst-case $O(|M|^2)$ constraints (where M is the set of methods in the C_TÆMS instance).

3.2 Efficiency Optimizations

In this section, we consider how to make the variable domains as small as possible while ensuring that the solution space of the DCOP still contains the optimal solution. We first consider how to ensure that domains are finite through a naive domain bounding method. Then, we leverage the hierarchical task structure of C_TÆMSto further reduce domain sizes followed by applying constraint propagation techniques such as arc consistency.

Naïve Domain Bounding. It is possible to create a finite (although not necessarily tight) upper bound on the start times of the methods. Let us consider a C_TÆMS instance in which all methods have an earliest start time of zero. Assuming all of the methods will be executed, the optimal start time of a method cannot be greater than the sum of the expected durations of all of the other methods. In the general case of heterogeneous earliest start times, we can define an upper bound on the start time of a method as the maximum finite earliest start time among all of the methods plus the duration of all other methods. This proposed method of naïve domain bounding will provably not remove all optimal solutions.

Bound Propagation. Although the nature of the distributed scheduling problem implies that a child's bounds are inherited from (and therefore cannot be looser than) its parent's, this is neither required nor enforced by C_TÆMS. Bounds can be propagated down the tree to improve upon the naïve bounding. Distributed bound propagation (requiring only local knowledge) is possible by propagating bound information down the HTN, enforcing the invariant between layers.

Constraint Propagation. A binary constraint is *arc consistent* if for each assignment of one variable there exists a feasible assignment of the other. A DCOP is said to be arc consistent if all constraints are arc consistent [7]. We use constraint propagation down the NLE chains to prune the domains, ensuring arc consistency of the DCOP. Distributed constraint propagation is achievable through continual broadcast of current start time bounds; if agents receive a bound that violates arc consistency, they increase the lower bound on their method's start time until the constraint is arc consistent and re-broadcast the new bounds. Since the lower bounds monotonically increase and are bounded above, the algorithm must terminate.

4 Results

We used the DARPA's COORDINATORs C_TÆMS scenario generator to randomly-generate a set of 100 C_TÆMS instances, each with four agents, three-to-four windows[2], one-to-three agents per window, and one-to-three NLE chains. Note that the scenario generator does not ensure that a feasible schedule exists for its resulting C_TÆMS instances. Experiments were terminated after 10,000 DCOP cycles after which a C_TÆMS instance is simply declared "insoluble."

Table 1. Efficiency of Constraint Propagation at reducing average domain size and state space size, in terms of solubility

Solubility	Avg. Domain Size Reduction	Avg. Final Domain Size	State Space Reduction	Final State Space Size
Solved	8.02%	35.29	97%	2.34×10^{71}
Unsolved	7.61%	35.24	94%	1.47×10^{77}

We used the DCOP algorithm Adopt [5] to solve the DCOPs obtained using our transformation mapping. Of the 100 random problems, none were soluble by the naïve domain bounding approach. Applying bound propagation to the naïve bounding resulted in 2% of the problems becoming soluble. Applying all of our methods resulted in 26% solubility. Using an upper one-sided paired t-test, we can say with 95% certainty that constraint propagation made an average domain size reduction of 7.62% over the domains produced from bound propagation alone. This is equatable to an average 94% decrease in state space size. Table 1 presents the state space reduction efficiency in terms of problem solubility. Since

Table 2. Solubility statistics for different complexities of C_TÆMS instances. All simulations had four agents. None of the problems bounded using the naïve method were soluble. * This is the default configuration for the C_TÆMS scenario generator.

		% Soluble		Avg. # Cycles		Avg. # Messages	
# Windows	# NLE Chains	Naïve	CP	Naïve	CP	Naïve	CP
3	0	0	40.00	-	143.87	-	2569.25
3	2	0	36.67	-	143.40	-	3114.14
3	4	0	46.67	-	220.73	-	3854.2
4	0	0	33.33	-	83.89	-	1758.5
4	2	0	11.76	-	66.18	-	2783
4	4	0	33.33	-	108.72	-	3887
5	0	0	60.00	-	122.1	-	1364.5
5	2	0	60.00	-	242.4	-	2634
5	4	0	50.00	-	248.8	-	5748.67
* 6	3	0	41.67	-	196.42	-	4025.08

[2] "Windows" are tasks whose parent is the task group (*i.e.*, they are tasks at the second layer from the root in the task hierarchy).

constraint propagation was fairly consistent in the percentage of state space reduced, this suggests that the 74% of the problems that remained insoluble were due to the large state space size inherent in their structure.

We also conducted tests over C_TÆMS problems of differing complexity; these data are presented in Table 2. Problems bounded naïvely were never soluble. Over the most complex problems with 6 windows and 3 NLEs chains, constraint propagation required an average of 144.94 messages (with a standard deviation of 16.13), which was negligible in comparison to the number required to arrive at an optimal solution.

5 Discussion

We have presented mappings from the C_TÆMS modeling language to an equivalent DCOP whose solution is guaranteed to lead to an optimal schedule. We have empirically validated our approach, using various existing techniques from the constraint processing literature, indicating that these problems are in fact soluble using our method. With the groundwork laid in solving distributed multi-agent coordination problems with distributed constraint optimization, we have many extensions in which to investigate. We hope to and are optimistic in extending our mapping to subsume a larger subset of C_TÆMS. If the resulting schedule need not be optimal (*i.e.*, feasibility is sufficient), approximation techniques for DCOPs also exist.

References

1. Boddy, M., Horling, B., Phelps, J., Golman, R., Vincent, R., Long, A., Kohout, B.: C-TAEMS language specification v. 1.06 (2005)
2. Musliner, D.J., Durfee, E.H., Wu, J., Dolgov, D.A., Goldman, R.P., Boddy, M.S.: Coordinated plan management using multiagent MDPs. In: Proc. of the AAAI Spring Symp. on Distributed Plan and Schedule Management, AAAI Press (2006)
3. Smith, S., Gallagher, A.T., Zimmerman, T.L., Barbulescu, L., Rubinstein, Z.: Multi-agent management of joint schedules. In: Proc. of the AAAI Spring Symp. on Distributed Plan and Schedule Management, AAAI Press (2006)
4. Phelps, J., Rye, J.: GPGP—a domain-independent implementation. In: Proc. of the AAAI Spring Symp. on Distributed Plan and Schedule Mgmt., AAAI Press (2006)
5. Modi, P.J., Shen, W., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence Journal (2005)
6. Decker, K.: TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In: Foundations of Distributed Artificial Intelligence, Chapter 16, G. O'Hare and N. Jennings (eds.), Wiley Inter-Science (1996) 429–448
7. Barták, R.: Theory and practice of constraint propagation. In Figwer, J., ed.: Proc. of the 3rd Wkshp. on Constraint Prog. in Decision Control, Poland (2001)

Interactive Distributed Configuration

Peter Tiedemann, Tarik Hadzic,
Thomas Stuart Henney, and Henrik Reif Andersen

Computational Logic and Algorithms Group, IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{petert, tarik, tsh, hra}@itu.dk

Abstract. *Interactive configuration* is the process of assisting a user in selecting values for parameters that respect given constraints. Inspired by the increasing demand for the real-time configuration in Supply Chain Management, we apply a compilation approach to the problem of *interactive distributed configuration* where the user options depend on constraints fragmented over a number of different locations connected through a network. We formalize the problem, suggest a solution approach based on an asynchronous compilation scheme, and perform experimental verification.

1 Introduction

An interactive configuration problem is an application of Constraint Satisfaction Problems (CSP) where a user is assisted in interactively assigning values to variables by a software tool. This *configurator* assists the user by displaying the valid choices for each unassigned variable in what are called *valid domains computations*. In many application domains user options depend on constraints fragmented over a number of different locations connected through a network. This paper attempts to extend the functionality provided by centralized configuration to distributed configuration while taking advantage of the distribution of the problem when possible. One of the current solution approaches to interactive configuration is to divide the computational effort into an offline and online phase. First compile the set of all valid solutions offline into a compact symbolic representation based on Binary Decision Diagrams (BDD). In the on-line phase we can deliver valid domains computation in time bounded polynomially in the size of the BDD representation[1,2,3].

2 Interactive Distributed Configuration

One application domain where constraint information is fragmented is in Supply Chain Management [4], where the user options depend on many interrelated businesses. The dominating relationship in a supply chain is the *supplier* producing items offered to *consumers*. We model these relationships with a *supply chain graph* (SCG), which is a directed graph $G(\mathcal{N}, A)$ where every node $N_i \in \mathcal{N}$

denotes a business entity and there is an arc $(N_i, N_j) \in A$ iff a company N_i supplies goods to N_j. We will assume that an SCG is a directed acyclic graph. Note that this will not imply in any way an acyclic constraint graph. In the SCM domain each company have independent variable namespaces in their models. For ease of exposition of the techniques discussed in this paper we will in the rest of the paper assume a global namespace between the distributed models, for details see [5]. We now introduce some basic terminology and define the distributed configuration model. An *assignment* ρ is a set of variable-value pairs (x, v), such that any one variable occurs only once. $dom(\rho)$ is the set of variables assigned by ρ. A total assignment is an assignment ρ such that $dom(\rho) = X$. An assigment ρ is valid iff there exists a total assignment ρ' such that $\rho \subseteq \rho'$ and ρ' satisfies all constraints.

Definition 1 (Distributed Configuration Model). *A* distributed configuration model $DisC(X, D, \mathcal{F}, \mathcal{N})$ *consists of a set of variables* X*, variable domains* D*, nodes* $\mathcal{N} = \{N_1, \ldots, N_k\}$ *and a set of sets of propositional formulas over subsets of* X*,* $\mathcal{F} = \{F_1, \ldots F_k\}$*. Each node* N_i *is associated a constraint* $C_i = \bigwedge_{f \in F_i} f$*. The scope of this constraint is denoted* X_i*, and the restriction of the domains to* X_i *is denoted* D_i*. A solution to the distributed configuration model is a total assignment to* X *satisfying* C_i *for all nodes* $N_i \in \mathcal{N}$*.*

We will interchangeably refer to constraints such as C_i as a constraint and as the set of solutions to this constraint over all variables X.

2.1 Overall Solution Approach

To solve the problem we take advantage of the fact that the user in any reasonable configuration problem over a supply chain is only interested in choosing values for a small subset of the variables, denoted X_u. We designate a node as the *user node* N_u, storing a constraint over variables X_u. The user performs interactive configuration by interacting with this node only. Let Sol be the set of solutions to the global problem, $Sol = \bigcap_{i=1}^{k} C_i$. At N_u we need to generate the *minimal solution space* over the user variables, that is $minSol_u = \pi_{X_u}(Sol)$. Hence all solutions in $minSol_u$ are *globally consistent.* After generating $minSol_u$ at N_u, the standard interactive configuration approach can be applied, resulting in a partial assignment $\rho \in minSol_u$[2]. Since $minSol_u$ is globally consistent we can extend ρ to $\rho' \in Sol$. Discovering this extension is called *validation* and can be performed efficiently. Due to space constraints we do not describe the validation algorithm here.

3 Distributed Compilation Algorithms

The most complex part of our solution approach is to generate a minimal solution space on a user node. The BDD representing $minSol_u$ can be calculated as: $\exists(X \backslash X_u).(C_1 \wedge C_2 \wedge C_3 ... \wedge C_k)$. Our algorithms will process a schedule placing and ordering the above conjunctions. We define a distributed conjunction schedule

```
SUBTREE-CONSISTENCY($N_i, T$)
 1  $activeTreeChildren \leftarrow$ CHILDREN($N_i, T$){CHILDREN($N_i, T$) := $\{N_j | N_j \preceq_T N_i\}$}
 2  $Sol_i \leftarrow C_i$
 3  $S \leftarrow scope_E^T(N_i) \cap scope_E^T$(PARENT($N_i, T$)){PARENT($N_i, T$) := $N_j$ s.t. $N_i \in$ CHILDREN($N_j, T$)}
 4  if $activeChildren = \emptyset$
 5    then $done \leftarrow True$
 6    else SEND(PARENT($N_i, T$), $\pi_S(C_i)$, $ACTIVE$)  { ASYN ONLY }
 7  while $\neg done$
 8  do $sol, sender, type \leftarrow$ RECEIVEMOSTRECENTMSG()
 9      Remove from message queue all older messages from sender
10      $Sol_i \leftarrow Sol_i \wedge sol$
11      if $type = FINISHED$
12        then REMOVE($activeChildren, sender$)
13              if $activeChildren = \emptyset$
14                then $done \leftarrow True$
15                else SEND(PARENT($N_i, T$), $\pi_S(Sol_i)$, $ACTIVE$)  { ASYN ONLY }
16  $send$(PARENT($N_i, T$), $\pi_S(C_i^E)$, $FINISHED$)
```

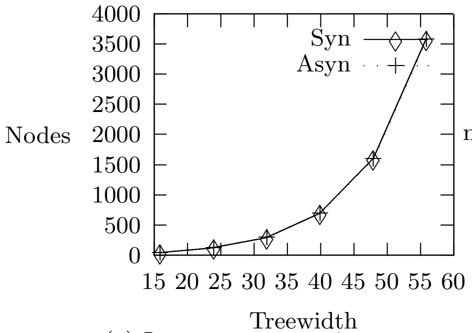
Fig. 1. Upon receiving an update(line 8) the receiving node N_i conjoins it with its current solution space Sol_i(line 10). If all children nodes have updated N_i a final update is send to the parent of N_i(line 16). ASyn in addition sends a tentative update whenever it has new information for its parent(line 6 and 15).

T as a tree with k nodes, where the root is labelled with C_u, and each remaining constraint is a label of some inner node. To each node N_i we associate a constraint Sol_i representing this nodes current view of the global solution space. Initially $Sol_i = C_i$. Given a distributed tree schedule T, if a node N_j is on a path between a root node and a node N_i, we say that N_j is an *ascendant* of N_i (or N_i is a *descendant* of N_j) and we write $N_i \preceq_T N_j$. Furthermore, given a fixed tree schedule T we define the *extended scope* $scope_E^T(N_i)$ of the node N_i as follows: $x \in scope_E^T(N_i)$ iff $x \in X_i$ or N_i has an ascendant $N_i \preceq_T N_j$ and a descendant $N_k \preceq_T N_i$ such that $x \in scope(N_j) \cap scope(N_k)$. Now we can define tree-width analogous to definitions in [6], as well as consistency notions used in the reminder of the paper.

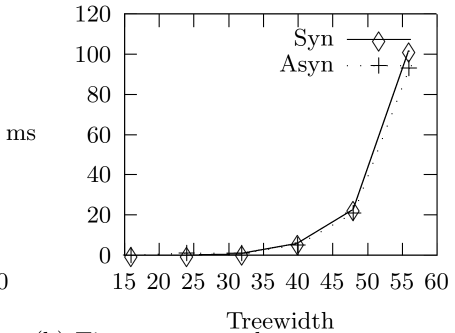
Definition 2 (Tree Width [6]). *The tree width w.r.t. a tree schedule T over a distributed model $DisC$ is denoted as $tw_T = max_{N \in \mathcal{N}} |scope_E^T(N)|$.*

Definition 3 ((Strong) Subtree Consistency). *A distributed configuration model $DisC$ is said to be* subtree consistent *w.r.t. a tree schedule T, if $\forall N_i \in \mathcal{N} : Sol_i = \pi_{X_i'}(\bigcap_{N_j \preceq_T N_i} C_j)$, where $X_i' \supseteq X_i$. If $scope_E^T(N_i) \subseteq X_i'$ $DisC$ is strongly subtree consistent.*

Subtree Consistency will enable backtrack free configuration at the user node while Strong Subtree Consistency is required for efficient validation. We describe two compilation algorithms that makes $DisC$ strongly subtree consistent. We denote them as *synchronous* (Syn) and *asynchronous* (ASyn) algorithm. Both are shown as variations of the algorithm SUBTREE-CONSISTENCY in Fig. 1. Syn obtains the simplest form of parallelism by processing seperate branches in T simultaneously. It sends one message per edge in T, that is $k-1$ messages. Asyn



(a) Largest message size



(b) Time to process largest message

N_{res}	$\sum_i c_S^i$	$\max_i\{m_S^i\}$	$\max_{i,u}\{c_{AS}^i(u)\}$	$\sum_i c_{AS}^i(l_i)$	$\max_i\{m_{AS}^i\}$	Asyn num msgs
29	1024	1800	396	972	3588	109
28+29	390	1205	380	322	3586	134
20	1504	3588	363	96	3588	126
10	2414	3588	340	96	3588	116
-	3099	3588	335	96	3588	109

(c) c_S^i is the time used to process the update to the i'th node in Syn. $c_{AS}^i(u)$ is the same for the u'th update received at node i in Asyn. $m_\cdot^i$ is the size of the i'th message send. l_i is the index of the last update received at node i. N_{res} is the index of the restricted node(s)

Fig. 2. Experimental results

allows seperate levels in T to operate in parallel. Each child sends a tentative version of its final constraint to its parent. Ascendants in T can immediately integrate this into their own constraint. When a node is updated it sends a new tentative solution space. In the worst case, each node sends one message plus one message per each node below it. The message complexity is $O(kl)$ where l is the depth of T. A hybrid of the two can be obtained by changing the message loop such that if an update that deprecates the one currently being processed arrives, the older one is abandoned. The following can be shown for all variants:

Lemma 1. *Given a compilation schedule T rooted in N_u, after the execution of SUBTREE-CONSISTENCY, DisC is strongly subtree consistent w.r.t. T.*

4 Experiments

For our experiments we utilize the GridCCC instance from [7]. The GridCCC instance is based on commonly occurring rules in supply chains involving availability of sub-components and production capacity. Its parameterized by the number of products per node, the number of components in a product, and the number of nodes. The best node ordering was achieved by assigning the user node index 1, and then assigning indices such that a supplier always had a higher index than its consumers. For GridCCC this gives a linear ordering.

To test feasibility we scale the treewidth of the instance by adjusting the number of product dependencies among nodes (Figure 2(a) and 2(b)). As expected we observe an exponential dependency on the treewidth, we also note that performance is acceptable even for quite large treewidth.

We compare the Syn and Asyn algorithms in Figure 2. The simulated Asyn algorithm differs from Figure 1 by not picking more recent messages first. Since the GridCCC instance is based on component availability constraints we expect ASyn to do well when all components are available. We restrict nodes at different positions to produce instances where child messages will have a greater impact. The algorithms were simulated on a single computer so we do not provide the total runtime for Asyn. Restricting nodes at the bottom of the schedule (high index) decreases the efficiency of the tentative updates, such that the ASyn algorithm spends a long time on the final updates. For less significant restrictions (smaller index) the time to handle the last messages as well as the largest message recieved remains small, meaning that the Asyn approach would be preferable. The hybrid approach suggested earlier would likely be preferable in practice so as to obtain good performance in all cases.

5 Conclusion

In this paper we addressed the problem of providing interactive configuration functionality in a distributed setting. We formally defined the concept of Distributed Configuration and proposed a solution approach based on techniques introduced in BDD-based compilation approaches and the area of Constraint Satisfaction. Finally we experimentally evaluated our approach and demonstrated its feasibility.

References

1. Møller, J., Andersen, H.R., Hulgaard, H.: Product configuration over the internet. In: Proceedings of the 6th INFORMS Conference on Information Systems and Technology. (2004)
2. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: PETO Conference, DTU-tryk (2004) 131–138
3. Subbarayan, S., Jensen, R.M., Hadzic, T., Andersen, H.R., Hulgaard, H., Møller, J.: Comparing two implementations of a complete and backtrack-free interactive configurator. In: CP'04 CSPIA Workshop. (2004) 97–111
4. Lancioni, R.A., Smith, M.F., Oliva, T.A.: The role of the internet in supply chain management. Industrial Marketing Management **29** (2000) 45–56
5. Hadzic, T., Henney, S., Andersen, H.R.: Interactive distributed configuration. In: CP2005 Workshop on Distributed and Speculative Constraint Processing. (2005)
6. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
7. Henney, S.: Interactive distributed configuration in supply chain management. Master's thesis, ITU (2006)

Retroactive Ordering for Dynamic Backtracking

Roie Zivan, Uri Shapen, Moshe Zazone, and Amnon Meisels

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel
{zivanr, moshezaz, shapenko, am}@cs.bgu.ac.il

Abstract. Dynamic Backtracking (DBT) is a well known algorithm for solving Constraint Satisfaction Problems. In DBT, variables are allowed to keep their assignment during backjump, if they are compatible with the set of eliminating explanations. A previous study has shown that when DBT is combined with variable ordering heuristics it performs poorly compared to standard Conflict-directed Backjumping (CBJ) [1]. The special feature of DBT, keeping valid elimination explanations during backtracking, can be used for generating a new class of ordering heuristics. In the proposed algorithm, the order of already assigned variables can be changed. Consequently, the new class of algorithms is termed *Retroactive DBT*.

In the proposed algorithm, the newly assigned variable can be moved to a position in front of assigned variables with larger domains and as a result prune the search space more effectively. The experimental results presented in this paper show an advantage of the new class of heuristics and algorithms over standard DBT and over CBJ. All algorithms tested were combined with forward-checking and used a *Min-Domain* heuristic.

1 Introduction

Conflict directed Backjumping (CBJ) is a technique which is known to improve the search of Constraint Satisfaction Problems (CSPs) by a large factor [4,7]. Its efficiency increases when it is combined with forward checking [8]. The advantage of CBJ over standard backtracking algorithms lies in the use of conflict sets in order to prune unsolvable sub search spaces [8]. The down side of CBJ is that when such a backtrack (back-jump) is performed, assignments of variables which were assigned later than the culprit assignment are discarded.

Dynamic Backtracking (DBT) [5] improves on standard CBJ by preserving assignments of non conflicting variables during back-jumps. In the original form of DBT, the culprit variable which replaces its assignment is moved to be the last among the assigned variables. In other words, the new assignment of the culprit variable must be consistent with all former assignments. Although DBT saves unnecessary assignment attempts and therefore was proposed as an improvement to CBJ, a later study by Baker [1] has revealed a major drawback of DBT. According to Baker, when no specific ordering heuristic is used, DBT performs better than CBJ. However, when ordering heuristics which are known to improve the run-time of CSP search algorithms by a large factor are used [6,2,3], DBT is slower than CBJ. This phenomenon is easy to explain.

Whenever the algorithm performs a back-jump it actually takes a variable which was placed according to the heuristic in a high position and moves it to a lower position. Thus, while in CBJ, the variables are ordered according to the specific heuristic, in DBT the order of variables becomes dependent on the algorithm's behavior [1].

In order to leave the assignments of non conflicting variables without a change on backjumps, DBT maintains a system of eliminating explanations ($Nogoods$) [5]. As a result, the DBT algorithm maintains dynamic domains for all variables and can potentially benefit from the *Min-Domain* (fail first) heuristic.

The present paper investigates a number of improvements to DBT that use radical versions of the *Min-Domain* heuristic. First, the algorithm avoids moving the culprit variable to the lowest position in the partial assignment. This alone can be enough to eliminate the phenomenon reported by Baker [1].

Second, the assigned variables which were originally ordered in a lower position than the culprit variable can be reordered according to their current domain size.

Third, a $retroactive$ ordering heuristic in which assigned variables are reordered is proposed. A *retroactive* heuristic allows an assigned variable to be moved upwards beyond assigned variables as far as the heuristic is justified.

If for example the variables are ordered according to the *Min-Domain* heuristic, the potential of each currently assigned variable to have a small domain is fully utilized. We note that although variables are chosen according to a *Min-Domain* heuristic, a newly assigned variable can have a smaller current domain than previously assigned variables. This can happen because of two reasons. First, as a result of *forward-checking* which might cause values from the current variables' domain to be eliminated due to conflicts with unassigned variables. Second, as a result of multiple backtracks to the same variable which eliminate at least one value each time. Therefore, the exploitation of the heuristic properties can be done, not only by choosing the next variable to be assigned, but by placing it in its *right* place among the assigned variables after it is assigned successfully.

The combination of the three ideas above was found to be successful in the empirical study presented in the present paper.

2 Retroactive Dynamic Backtracking

We assume in our presentation that the reader is familiar with both DBT following [1] and CBJ [8].

The first step in enhancing the desired heuristic (*Min-Domain* in our case) for DBT is to avoid the moving forward variables that the algorithm backtracks to (i.e. culprit variables). One way to do this is to try to replace the assignment of the culprit variable and *leave the variable in the same position*.

The second step is to reorder the assigned variables that have a lower order than the culprit assignment which was replaced. This step takes into consideration the possibility that the replaced assignment of a variable that lies higher in the order has the potential to change the size of the current domains of the already assigned variables that are ordered after it. The simplest way to perform this step is to reassign these variables and order them using the desired heuristic.

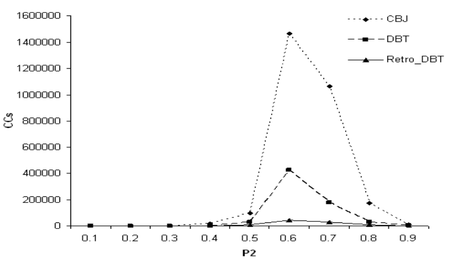
```
Retroactive FC_DBT
1.  var_list $\leftarrow$ variables;
2.  assigned_list $\leftarrow$ $\phi$;
3.  pos $\leftarrow$ 1;
4.  while (pos $<$ N)
5.     next_var $\leftarrow$ select_next_var(var_list);
6.     var_list.remove(next_var);
7.     assign(next_var);
8.  report solution;

procedure assign(var)
10. for each (value $\in$ var.current_domain)
11.    var.assignment $\leftarrow$ value;
12.    consistent $\leftarrow$ true;
13.    forall (i $\in$ var_list)
          and while consistent
14.       consistent $\leftarrow$ check_forward(var, i);
15.    if not (consistent)
16.       nogood $\leftarrow$ resolve_nogoods(i);
17.       store(var, nogood);
18.       undo_reductions(var, pos);
19.    else
20.       nogood $\leftarrow$ resolve_nogoods(pos);
21.       lastVar $\leftarrow$ $nogood.RHS\_variable$;
21.       newPos $\leftarrow$ select_new_pos(var, lastVar);
22.       assigned_list.insert(var, newPos);
23.       forall (var_1 $\in$ assigned_list) and
                (pos_var_1 $>$ newPos)
24.          check_forward(var, var_1);
25.          update_nogoods(var, var_1);
26.       forall (var_2 $\in$ var_list)
27.          update_nogoods(var, var_2);
28.       pos $\leftarrow$ pos+1;
29.       return;
30. var.assignment $\leftarrow$ $Nil$;
31. backtrack(var);

procedure backtrack($var$)
32. nogood $\leftarrow$ resolve_nogoods(var);
33. if (nogood = $\phi$)
34.    report no solution;
35.    stop;
36. culprit $\leftarrow$ $nogood.RHS\_variable$;
37. store(culprit, nogood);
38. culprit.assignment $\leftarrow$ $Nil$;
39. undo_reductions(culprit, pos_culprit);
40. forall (var_1 $\in$ assigned_list) and
          (pos_var_1 $>$ newPos)
41.    undo_reductions(var_1, pos_var_1);
42.    var_1.assignment $\leftarrow$ Nil;
43.    var_list.insert(var_1);
44.    assigned_list.remove(var_1);
45. pos $\leftarrow$ pos_culprit;

procedure update_nogoods(var_1, var_2)
46. for each (val $\in$
          {var_2.domain - var_2.current_domain})
47.    if not (check(var_2, val, var_1.assignment))
48.       nogood $\leftarrow$ remove_eliminating_nogood
             (var_1, val);
49.       if not ($\exists var\_3$ $\in$ nogood and
             pos_var_3 $<$ pos_var_1)
50.          nogood $\leftarrow$ $\langle$var_1.assignment $\rightarrow$
             var_2 $\neq$ val$\rangle$;
51.       store(var_2, nogood);
```

Fig. 1. The Retroactive FC_DBT algorithm

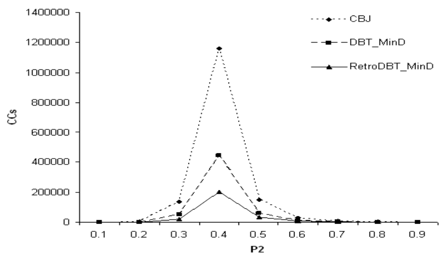
The third step derives from the observation that in many cases the size of the current domain of a newly assigned variable is smaller than the current domains of variables which were assigned before it.

Allowing a reordering of assigned variables enables the use of heuristic information which was not available while the previous assignments have been performed. This takes ordering heuristics to a new level and generates a radical new approach. Variables can be moved up in the order, in front of assigned variables of the partial solution. As long as the new assignment is placed *after* the most recent assignment which is in conflict with one of the variable's values, the size of the domain of the assigned variable is not changed.

In the best ordering heuristic proposed by the present paper, the new position of the assigned variable in the order of the $partial\_solution$ is dependent on the size of its current domain. The heuristic checks all assignments from the last up to the first assignment which is included in the union of the newly assigned variable's eliminating Nogoods. The new assignment will be placed right after the first assigned variable with a smaller current domain.

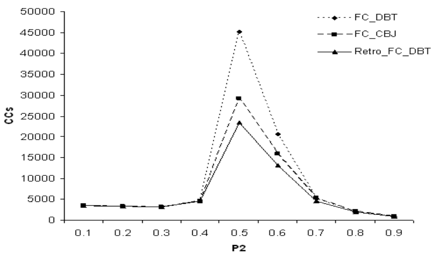


(a)

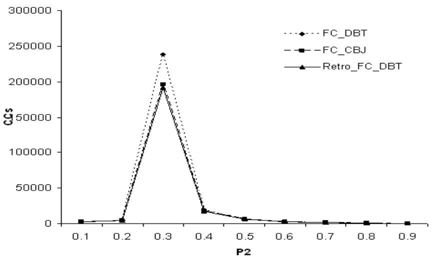


(b)

Fig. 2. CCs performed by *DBT*, *CBJ* and *Retroactive DBT* (a) $p_1 = 0.3$, (b) $p_1 = 0.7$



(a)



(b)

Fig. 3. CCs performed by *FC_DBT*, *FC_CBJ* and *FC_Retroactive DBT* (a) $p_1 = 0.3$, (b) $p_1 = 0.7$

Figures 1 presents the code of *Retroactive Forward Checking Dynamic Backtracking* ($Retro\_FC\_DBT$). For lack of space we leave out the detailed description of the algorithm and its correctness proof.

3 Experimental Evaluation

The common approach in evaluating the performance of *CSP* algorithms is to measure time in logical steps to eliminate implementation and technical parameters from affecting the results. The number of constraints checks serves as the measure in our experiments [9,7].

Experiments were conducted on random *CSPs* of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 (which are commonly used in experimental evaluations of CSP algorithms [10]). Two sets of experiments were performed. In the first set the *CSPs* included 15 variables ($n = 15$) and in the second set the *CSPs* included 20 variables ($n = 20$). In all of our experiments the number of values for each variable was 10 ($k = 10$). Two values of constraints density were used, $p_1 = 0.3$ and $p_1 = 0.7$. The tightness value p_2, was varied between 0.1 and 0.9, in order to cover all ranges of problem difficulty. For each of the pairs of fixed density and tightness (p_1, p_2), 50 different random problems were solved by each algorithm and the results presented are an average of these 50 runs.

Three algorithms were compared, Conflict Based Backjumping (CBJ), Dynamic Backtracking (DBT) and Retroactive Dynamic Backtracking (*Retro* DBT). In all of our experiments all the algorithms use a *Min-Domain* heuristic for choosing the next variable to be assigned. In the first set of experiments, the three algorithms were implemented without forward-checking.

Figure 2 (a) presents the number of constraints checks performed by the three algorithms on low density CSPs ($p_1 = 0.3$). The CBJ algorithm does not benefit from the heuristic when it is not combined with forward-checking. The advantage of both versions of DBT over CBJ is therefore large. *Retroactive DBT* improves on standard *DBT* by a large factor as well. Figure 2 (b) present the results for high density CSPs($p_1 = 0.7$). Although the results are similar, the differences between the algorithms are smaller for the case of higher density CSPs.

In our second set of experiments, each algorithm was combined with *Forward-Checking* [8]. This improvement enabled testing the algorithms on larger CSPs with 20 variables

Figure 3 (a) presents the number of constraints checks performed by each of the algorithms. It is very clear that the combination of *CBJ* with forward-checking improves the algorithm and makes it compatible with the others. This is easy to explain since the pruned domains as a result of forward-checking enable an effective use of the Min-Domain heuristic. Both *FC_CBJ* and *Retroactive FC_DBT* outperform *FC_DBT*. *Retroactive FC_DBT* performs better than *FC_CBJ*. Figures 3 (b) presents similar results for higher density CSPs. As before, the differences between the algorithms are smaller when solving CSPs with higher densities.

4 Discussion

Variable ordering heuristics such as *Min-Domain* are known to improve the performance of CSP algorithms [6,2,3]. This improvement results from a reduction in the search space explored by the algorithm. Previous studies have shown that DBT does not preserve the properties of variable ordering heuristics and as a result performs poorly compared to CBJ [1]. The *Retroactive* DBT algorithm, presented in this paper, combines the advantages of both previous algorithms by preventing the placing of variables in a position which does not support the heuristic and allowing the reordering (or reassigning) of assigned variables with lower priority than the culprit assignment after a backtrack operation.

We have used the mechanism of *Dynamic Backtracking* which by maintaining eliminating $Nogoods$, allows variables with higher priority to be reassigned while lower priority variables keep their assignment. These dynamically maintained domains enable to take the *Min-Domain* heuristic to a new level. Standard backtracking algorithms use ordering heuristics only to decide on which variable is to be assigned next. *Retroactive* DBT enables the use of heuristics which reorder assigned variables. Since the sizes of the current domains of variables are dynamic during search, the flexibility of the heuristics which are possible in *Retroactive* DBT enables a dynamic enforcement of the *Min-Domain* property over assigned and unassigned variables.

The ordering of assigned variables requires some overhead in computation when the algorithm maintains consistency by using *Forward Checking*. This overhead was found by the experiments presented in this paper to be worth the effort since the overall computation effort is reduced.

References

1. Andrew B. Baker. The hazards of fancy backtracking. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94), Volume 1*, pages 288–293, Seattle, WA, USA, July 31 - August 4 1994. AAAI Press.
2. C. Bessiere and J.C. Regin. Using bidirectionality to speed up arc-consistency processing. *Constraint Processing (LNCS 923)*, pages 157–169, 1995.
3. R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136:2:147–188, April 2002.
4. Rina Dechter. *Constraint Processing*. Morgan Kaufman, 2003.
5. M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.
6. R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
7. G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 21:365–387, 1997.
8. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
9. P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
10. B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.

Author Index