Lecture Notes in Artificial Intelligence 2627

Subseries of Lecture Notes in Computer Science
Edited by J. G. Carbonell and J. Siekmann

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Barry O'Sullivan (Ed.)

Recent Advances in Constraints

Joint ERCIM/CologNet International Workshop on
Constraint Solving and Constraint Logic Programming
Cork, Ireland, June 19-21, 2002
Selected Papers

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editor

Barry O'Sullivan
Department of Computer Science
University College Cork, Ireland
E-mail: b.osullivan@cs.ucc.ie

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie; detailed bibliographic data is available in the Internet at <http://dnd.ddb.de>.

CR Subject Classification (1998): I.2.3-4, D.1, D.3.2-3, F.3.2, F.4.1, I.2.8, F.2.2

ISSN 0302-9743
ISBN 3-540-00986-8 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York,
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin GmbH
Printed on acid-free paper SPIN: 10873031 06/3142 5 4 3 2 1 0

Preface

This volume contains a selection of papers from the Joint ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming, held at the Cork Constraint Computation Centre from the 19th to the 21st of June 2002. The workshop co-located two events: the seventh meeting of the ERCIM Working Group on Constraints, co-ordinated by Krzysztof Apt, and the first annual workshop of the CologNet Area for Constraint and Logic Programming, co-ordinated by Francesca Rossi.

The aim of this workshop was to provide a forum where researchers in constraint processing could meet in an informal setting and discuss their most recent work. The Cork Constraint Computation Centre was chosen as the venue for the workshop because it is a new research centre, supported by Science Foundation Ireland and led by Eugene Freuder, which is entirely devoted to studying constraint processing. Thus, the workshop participants had an opportunity to see the centre, meet its members and investigate the potential for future collaboration.

Amongst the topics addressed by the papers in this volume are: verification and debugging of constraint logic programs; modelling and solving CSPs; explanation generation; inference and consistency processing; SAT and 0/1 encodings of CSPs; soft constraints and constraint relaxation; real-world applications; and distributed constraint solving.

I would like to acknowledge the reviewers for their assistance in evaluating submissions for inclusion in this volume. I wish to thank the ERCIM Working Group on Constraints, the CologNet Area for Constraint and Logic Programming, and the Cork Constraint Computation Centre for sponsoring this event by covering all the local expenses and supporting the participation of several Ph.D. students.

January 2003 Barry O'Sullivan

Organizing Committee

Krzysztof R. Apt, CWI, The Netherlands
François Fages, INRIA, France
Eugene C. Freuder, University College Cork, Ireland
Barry O'Sullivan, University College Cork, Ireland
Francesca Rossi, University of Padova, Italy
Toby Walsh, University College Cork, Ireland

Additional Referees

Stefano Bistarelli
Lucas Bordeaux
James Bowen
Sebastian Brand
Ken Brown
Inês Lynce
João Marques-Silva
Ian Miguel
Eric Monfroy
Steve Prestwich
Patrick Prosser
Pearl Pu
Christian Schulte
Richard J. Wallace
Peter Zoeteweij

Sponsors

ERCIM Working Group on Constraints
CologNet Area for Constraint and Logic Programming
Cork Constraint Computation Centre

Table of Contents

Abstract Verification and Debugging of Constraint Logic Programs

Manuel Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García

Department of Computer Science
Technical University of Madrid (UPM)
{herme,german,bueno,pedro}@fi.upm.es

(Extended Abstract)

Keywords: Global Analysis, Debugging, Verification, Constraint Logic Programming, Optimization, Parallelization, Abstract Interpretation.

1 Background

The technique of Abstract Interpretation [13] has allowed the development of sophisticated program analyses which are provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to *optimization* during program compilation. However, recently, novel and promising applications of semantic approximations have been proposed in the more general context of program *verification* and *debugging* [3,10,7].

In the case of Constraint Logic Programs (CLP), a comparatively large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis have been developed to a powerful and mature level (see, e.g., [28,9,21,6,22,24] and their references). These systems can approximate at compile-time a wide range of properties, from directional types to variable independence, determinacy or termination, always safely, and with a significant degree of precision.

Our proposed approach takes advantage, within the context of program verification and debugging, of these significant advances in static program analysis techniques and the resulting concrete tools, which have been shown useful for other purposes such as optimization, and are thus likely to be present in compilers. This is in contrast to using traditional proof-based methods (e.g., for the case of CLP, [1,2,15,19,34]), developing new tools and procedures (such as specific concrete [4,17,18] or abstract [10,11] diagnosers and declarative debuggers), or limiting error detection to run-time checking (e.g., [34]).

2 An Approach Based on Semantic Approximations

We now briefly describe the basis of our approach [7,25,31]. We consider the important class of semantics referred to as *fixpoint semantics*. In this setting, a

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 1–14, 2003.

Table 1. Set theoretic formulation of verification problems

Property	Definition
P is partially correct w.r.t. $\mathcal{I}$	$[\![P]\!] \subseteq \mathcal{I}$
P is complete w.r.t. $\mathcal{I}$	$\mathcal{I} \subseteq [\![P]\!]$
P is incorrect w.r.t. $\mathcal{I}$	$[\![P]\!] \not\subseteq \mathcal{I}$
P is incomplete w.r.t. $\mathcal{I}$	$\mathcal{I} \not\subseteq [\![P]\!]$

(monotonic) semantic operator (which we refer to as S_P) is associated with each program P. This S_P function operates on a semantic domain which is generally assumed to be a complete lattice or, more generally, a chain complete partial order. The meaning of the program (which we refer to as $[\![P]\!]$) is defined as the least fixpoint of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixpoint is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

Both program verification and debugging compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program, which we denote by $\mathcal{I}$. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In Table 1 we define classical verification problems in a set-theoretic formulation as simple relations between $[\![P]\!]$ and $\mathcal{I}$.

Using the exact actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be only partially known, infinite, too expensive to compute, etc. An alternative and interesting approach is to approximate the semantics. This is interesting, among other reasons, because a well understood technique already exists, abstract interpretation, which provides *safe* approximations of the program semantics. Our first objective is to present the implications of the use of *approximations* of both the intended and actual semantics in the verification and debugging process.

2.1 Approximating Program Semantics

We start by recalling some basic concepts from abstract interpretation. In this technique, a program is interpreted over a non-standard domain called the *abstract* domain D_α which is simpler than the *concrete* domain D, and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program is computed (or approximated) by replacing the operators in the program by their abstract counterparts.

The concrete and abstract domains are related via a pair of monotonic mappings: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, which relate the two domains by a Galois insertion (or a Galois connection) [13]. We will denote by $[\![P]\!]_\alpha$ the result of abstract interpretation for a program P. Typically, abstract

interpretation guarantees that $[\![P]\!]_\alpha$ is an over-approximation of the abstract semantics of the program itself, $\alpha([\![P]\!])$. Thus, we have that $[\![P]\!]_\alpha \supseteq \alpha([\![P]\!])$, which we will denote as $[\![P]\!]_{\alpha^+}$. Alternatively, the analysis can be designed to safely under-approximate the actual semantics, and then we have that $[\![P]\!]_\alpha \subseteq \alpha([\![P]\!])$, which we denote as $[\![P]\!]_{\alpha^-}$.

2.2 Abstract Verification and Debugging

The key idea in our approach is to use the abstract approximation $[\![P]\!]_\alpha$ directly in verification and debugging tasks. As we will see, the possible loss of accuracy due to approximation prevents full verification in general. However, and interestingly, it turns out that in many cases useful verification and debugging conclusions can still be derived by comparing the approximations of the actual semantics of a program to the (also possibly approximated) intended semantics.

A number of approaches have already been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. Abstractions were used in the context of algorithmic debugging in [27]. Abstract interpretation for debugging of imperative programs has been studied by Bourdoncle [3], and for the particular case of algorithmic debugging of logic programs by Comini et al. [12] (making use of partial specifications) and [10].

In our approach we actually compute the abstract approximation $[\![P]\!]_\alpha$ of the actual semantics of the program $[\![P]\!]$ and compare it directly to the (also approximate) intention (which is given in terms of *assertions* [30]), following almost directly the scheme of Table 1. This approach can be very attractive in programming systems where the compiler already performs such program analysis in order to use the resulting information to, e.g., optimize the generated code. I.e., in these cases the compiler will compute $[\![P]\!]_\alpha$ anyway.

For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Thus, for comparison we need in principle $\alpha([\![P]\!])$. Using abstract interpretation, we can usually only compute instead $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$. Thus, it is interesting to study the implications of comparing $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$.

In Table 2 we propose (sufficient) conditions for correctness and completeness w.r.t. $\mathcal{I}_\alpha$, which can be used when $[\![P]\!]$ is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses which over-approximate the actual semantics (i.e., those denoted as $[\![P]\!]_{\alpha^+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred for the program is incompatible with the abstract specification, i.e., when $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset$. We also note that it will only be possible to prove completeness if the abstraction is *precise*, i.e., $[\![P]\!]_\alpha = \alpha([\![P]\!])$. According to Table 2 only $[\![P]\!]_{\alpha^-}$ can

Table 2. Validation problems using approximations

Property	Definition	Sufficient condition
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

be used to this end, and in the case we are discussing $[\![P]\!]_{\alpha^+}$ holds. Thus, the only possibility is that the abstraction is precise.

On the other hand, if analysis under-approximates the actual semantics, i.e., in the case denoted $[\![P]\!]_{\alpha^-}$, it will be possible to prove completeness and incorrectness. In this case, partial correctness and incompleteness can only be proved if the analysis is precise.

If analysis information allows us to conclude that the program is incorrect or incomplete w.r.t. $\mathcal{I}_\alpha$, an (abstract) symptom has been found which ensures that the program does not satisfy the requirement. Thus, debugging should be initiated to locate the program construct responsible for the symptom.

More details about the theoretical foundation of our approach can be found in [7,31].

3 A Practical Framework and Its Implementation

Using the ideas outlined above, we have developed a framework [25,29] capable of combined static and dynamic validation, and debugging for CLP programs, using semantic approximations, and which can be integrated in an advanced program development environment comprising a variety of co-existing tools [16].

This framework has been implemented as a generic preprocessor composed of several tools. Figure 1 depicts the overall architecture of the system. Hexagons represent the different tools involved and arrows indicate the communication paths among the different tools.

Program verification and detection of errors is first performed at compile-time by using the sufficient conditions shown in Table 2. I.e., by inferring properties of the program via abstract interpretation-based static analysis and comparing this information against (partial) specifications written in terms of assertions. Such assertions are linguistic constructions which allow expressing properties of programs.

Classical examples of assertions are type declarations (e.g., in the context of (C)LP those used by [26,32,5]). However, herein we are interested in supporting a much more powerful setting in which assertions can be of a much more general nature, stating additionally other properties, some of which cannot always be determined statically for all programs. These properties may include properties

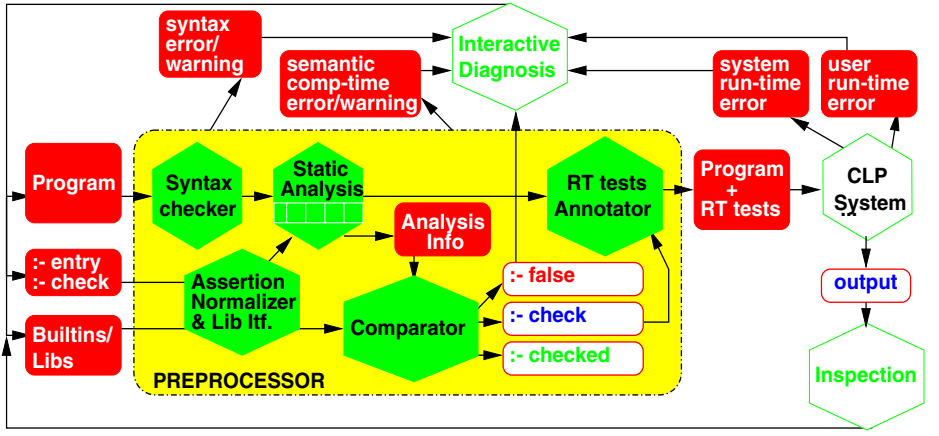


Fig. 1. Architecture of the Preprocessor

defined by means of user programs and extend beyond the predefined set which may be natively understandable by the available static analyzers. Also, in the proposed framework only a small number of (even zero) assertions may be present in the program, i.e., the assertions are *optional*. In general, we do not wish to limit the programming language or the language of assertions unnecessarily in order to make the validity of the assertions statically decidable (and, consequently, the proposed framework needs to deal throughout with *approximations*). We also propose a concrete language of assertions which allows writing this kind of (partial) specifications for CLP [30].

The assertion language is also used by the preprocessor to express both the information inferred by the analysis and the results of the comparisons performed against the specifications.[1] As can be derived from Table 2, these comparisons can result in proving statically (i.e., at compile-time) that the assertions hold (i.e., they are validated) or that they are violated, and thus bugs have been detected. User-provided assertions (or *parts* of assertions) which cannot be statically proved nor disproved are optionally translated into run-time tests. Both the static and the dynamic checking are provably *safe* in the sense that all errors flagged are definite violations of the specifications.

The practical usefulness of the framework is illustrated by what is arguably the first and most complete implementation of these ideas: CiaoPP,[2] the Ciao system preprocessor [29,24]. Ciao is a public-domain, next-generation constraint logic programming system, which supports ISO-Prolog, but also, selectively for each module, extensions and restrictions such as, for example, pure logic programming, constraints, functions, objects, or higher-order. Ciao is specifically

[1] Interestingly, the assertions are also quite useful for generating documentation automatically (see [23]).

[2] A demonstration of the system was performed at the meeting.

designed to a) be highly extensible and b) support modular program analysis, debugging, and optimization. The latter tasks are performed in an integrated fashion by CiaoPP.

CiaoPP, which incorporates analyses developed by several groups in the LP and CLP communities, uses abstract interpretation to infer properties of program predicates and literals, including types, modes and other variable instantiation properties, constraint independence, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc. It processes modules separately, performing incremental analysis. CiaoPP can find errors at compile-time (or perform partial verification) by checking how programs call system libraries. This is possible since the expected behaviour of system predicates is also given in terms of assertions This allows detecting errors in user programs even if they contain no assertions. Also, the preprocessor can detect errors as well by checking the assertions present in the program or in other modules used by the program. As already mentioned, assertions are completely optional. Nevertheless, if the program is not correct, the more assertions are present in the program the more likely it is for errors to be automatically detected. Thus, for those parts of the program which are potentially buggy or for parts whose correctness is crucial, the programmer may decide to invest more time in writing assertions than for other parts of the program which are more stable. In addition, CiaoPP also performs program transformations and optimizations such as multiple abstract specialization, parallelization (including granularity control), and inclusion of run-time tests for assertions which cannot be checked completely at compile-time.

Finally, the implementation of the preprocessor is generic in that it can be easily customized to different CLP systems and dialects and in that it is designed to allow the integration of additional analyses in a simple way. As a particularly interesting example, the preprocessor has been adapted for use with the CHIP CLP(FD) system. This has resulted in CHIPRE, a preprocessor for CHIP which has been shown to detect non-trivial programming errors in CHIP programs. In the next section we show an example of a debugging session with CHIPRE. More information on the system can be found in [29].

4 A Sample Debugging Session with CHIPRE

In this section we will show some of the capabilities of our debugging framework through a sample session with CHIPRE, an implemented instance of the framework. Consider Figure 2, which contains a tentative version of a CHIP program for solving the *ship* scheduling problem, a typical CLP(FD) benchmark.

Often, the results of static analysis are good indicators of bugs, even if no assertion is given. This is because "strange" results often correspond to bugs. An important observation is that plenty of static analyses, such as modes and regular types, compute over-approximations of the success sets of predicates. Then, if such an over-approximation corresponds to the empty set then this implies that such predicate never succeeds. Thus, unless the predicate is dead-code, this often

```
solve(Upper,Last,N,Dis,Mis,L,Sis):-
        length(Sis, N),
        Sis :: 0..Last,
        Limit :: 0..Upper,
        End :: 0..Last,
        set_precedences(L, Sis, Dis),
        cumulative(Sis, Dis, Mis, unused, unsed, Limit, End, unused),
        min_max(labeling(Sis), End).

labeling([]).
labeling([H|T]):-
        delete(X,[H|T],R,0,most_constrained),
        indomain(X),
        labeling(R).

set_precedences(L, Sis, Dis):-
        Array_starts=..[starts|Sis], % starts(S1,S2,S3,...)
        Array_durations=..[durations|Dis], % durations(D1,D2,D3,...)
        initialize_prec(L,Array_starts),
        set_pre_lp(1, array_starts, Array_durations).

set_pre_lp([], _, _).
set_pre_lp([After#>=Before|R], Array_starts, Array_durations):-
        arg(After, Array_starts, S2),
        arg(Before, Array_starts, S1),
        arg(Before, Array_durations, D1),
        S2 #>= S1 + D1,
        set_pre_lp(R, Array_starts, array_durations).

initialize_prec(_,_).
```

Fig. 2. A tentative *ship* program in CHIP

indicates that the code for the predicate is erroneous since every call either fails finitely (or raises an error) or loops. If analysis is goal-dependent and thus also computes an over-approximation of the calling states to the predicate, predicates which are dead-code can often be identified by having an over-approximation of the calling states which corresponds to the empty set.

We now preprocess the current version of our example program using *regular type* [35,14,21,20,33] analysis. Our implementation of regular types is goal-dependent and thus computes over-approximations of both the success set and calling states of all predicates. In addition, our analysis also computes over-approximations of the values of variables at each program point. Once analysis information is available, the preprocessor automatically checks the consistency of the analysis results and we get the following messages:

```
WARNING: Literal set_precedences(L, Sis, Dis)
           at solve/7/1/5 does not succeed!
WARNING: Literal set_pre_lp(l, array_starts, Array_duration)
           at set_precedences/3/1/4 does not succeed!
```

The first warning message refers to a literal (in particular, the 5th literal in the 1st clause of `solve/7`) which calls the predicate `set_precedences/3`, whose success type is empty. Also, even if the success type of a predicate is not empty, i.e., there may be some calls which succeed, it may be possible to detect that at a certain program point the given call to the predicate cannot succeed because the type of the particular call is incompatible with the success type of the predicate. This is the reason for the second warning message. Note that this kind of reasoning can only be made if (1) the static analysis used infers properties which are *downwards closed*, i.e., once they hold they keep on being valid during forward execution and (2) analysis computes descriptions at each program point which, as already mentioned, is the case with our regular type analysis. Note that the predicate `set_pre_lp/3` can only succeed if the value at the first argument is compatible with a list. However, the call `set_pre_lp(l, array_starts, Array_duration)` has the constant `l` at the first argument position. This is actually a bug, as the constant `l` should instead be the variable `L`. Once we correct this bug, in subsequent preprocessing of the program both warning messages disappear. In fact, the first one was also a consequence of the same bug which propagated to the calling predicates of `set_precedences/3`.

4.1 Aiding the Analyzer

In the ship program, all initial queries to the program are intended to be to the `solve` predicate. However, the compiler has no way to automatically determine this. Thus, in the absence of more information, the most general possible calls have to be assumed for *all* predicates in the program.[3] One way to alleviate this is to provide *entry* assertion(s) which are assumed to cover all possible initial calls to the program. Even the simplest entry declaration which can be given for predicate `solve`, i.e., '`:- entry solve/7.`', is very useful for goal-dependent static analysis. Since it is the only `entry` assertion, the only calls to the rest of the predicates in the program are those generated during computations of `solve/7`. This allows analysis to start from the predicate `solve/7` only, instead of from all predicates. Reducing the number (and generality) of starting points for goal-dependent analysis by means of `entry` declarations often leads to increased precision and reduced analysis times. However, analysis will still make no assumptions regarding the arguments of the calls to `solve/7` since there is no further information available. This could be improved using a more accurate entry declaration such as the following:

[3] Note that this can be partly alleviated with a strict module system such as that of Ciao [8], in which only exported predicates of a module can be subject to initial queries.

```
:- entry solve/7 : int * int * int * list(int) * list(int) * list * term.
```

It gives the types of the seven arguments, and describes more precisely the valid input data. Note that the assertion above also specifies a *mode* for the calling patterns. The first three arguments are *required* to be instantiated to integers. The forth and fifth must be fully instantiated to lists of integers. The sixth argument is (only) required to be instantiated to a list skeleton. Finally, the seventh argument can be any possible term. Note that, by default, our assertion language interprets properties in assertions as *instantiation* properties. However, the assertion language also allows the use of *compatibility* properties if so desired [30].

4.2 Assertions for System Predicates

Consider a new version of the *ship* program, after correcting the typo involving `L` and introducing the (simple) entry declaration '`:- entry solve/7.`'. When preprocessing the program the following messages are issued:

```
ERROR: Builtin predicate
        cumulative(Sis,Dis,Mis,unused,unsed,Limit,End,unused)
        at solve/7/1/6 is not called as expected (argument 5):
         Called:    ^unsed
         Expected:  intlist_or_unused

ERROR: Builtin predicate arg(After,Array_starts,S2)
        at set_pre_lp/3/2/1 is not called as expected (argument 2):
         Called:    ^array_starts
         Expected:  struct
```

Which indicate that the program is still definitely incorrect. Note that the preprocessor could not detect this without the extra precision allowed by the `entry` assertion. In error messages involving regular types, one important issue is not to confuse term constructors with type constructors. In order to improve the readability and conciseness of the error messages, the marker `^` is used to distinguish terms (constants) from regular types (which represent regular sets of terms). By default, values represent regular types. However, if they are marked with `^` they represent constants. In our example, `intlist_or_unused` is a type since it is not marked with `^` whereas `^unsed` is a constant. Note that though it is always possible to define a regular type which contains a single constant such as `unsed` and distinguish terms from types by the context in which the value appears, we opt by introducing the marker `^` ("quote") since in our experience this improves readability of error messages. Note that defining such type explicitly instead would require inventing a new name for it and providing the definition of the type together with the error message.

Coming back to the pending error messages, the first message is due to the fact that the constant `unused` has been mistakingly typed as `unsed` in the fifth argument of the call to the CHIP builtin predicate `cumulative/8`. As indicated in the error message, this predicate requires the fifth argument to be of type

`intlist_or_unused` which was defined when writing assertions for the system predicates in CHIP and which indicates that such argument must be either the constant `unused` or a list of integers.

The automatic detection of this error at compile-time has been possible because the CHIP builtins have been provided with assertions that describe their intended use. Though system predicates are in principle considered correct under the assumption that they are called with valid input data, it is often useful to check that they are indeed called with valid input data. In fact, existing CLP systems perform this checking at run-time. The existence of such assertions allows checking the calls to system predicates at compile-time in addition to run-time in CLP systems which originally do not perform compile-time checking.

In the second message we have detected that we call the CHIP builtin predicate `arg/3` with the second argument bound to `array_starts` which is a constant (as indicated by the marker `^`) and thus of arity zero. This is incompatible with the expected call type `struct`, i.e., a structure with arity strictly greater than zero. In the current version of CHIP, this will generate a run-time error, whereas in other systems such as Ciao and SICStus, this call would fail but would not raise an error. Though we know the program is incorrect, the literal where the error is flagged, `arg(After, Array_starts, S2)` is apparently correct. We correct the first error and leave detection of the cause for the second error for later.

The different behaviour of seemingly identical builtin predicates (such as `arg/3` in the example above) in different systems further emphasizes the benefits of describing builtin predicates by means of assertions. They can be used for easily customizing static analysis for different systems, as assertions are easier to understand by naive users of the analysis than the hardwired internal representation used in ad-hoc analyzers for a particular system.

4.3 Assertions for User-Defined Predicates

Up to now we have seen that the preprocessor is capable of issuing a good number of error and warning messages even if the user does not provide any `check` assertions (assertions that the system should check to hold). We believe that this is very important in practice. However, adding assertions to programs can be useful for several reasons. One is that they allow further semantic checking of the programs, since the assertions provided encode a partial specification of the user's intention, against which the analysis results can be compared. Another one is that they also allow a form of diagnosis of the error symptoms detected, so that in some cases it is possible to automatically locate the program construct responsible for the error.

Consider again the pending error message from the previous iteration over the ship program. We know that the program is incorrect because (global) type analysis tells us that the variable `Array_starts` will be bound at run-time to the constant `array_starts`. However, by just looking at the definition of predicate `set_pre_lp` it is not clear where this constant comes from. This is because the cause of this problem is not in the definition of `set_pre_lp` but rather in that the predicate is being used incorrectly (i.e., its precondition is violated). We thus

introduce the following `calls` assertion, which describes the expected calls to the predicate:

```
:- calls set_pre_lp(A,B,C): (struct(B),struct(C)).
```

In this assertion we require that both the second and third parameters of the predicate, i.e., `B` and `C` are structures with arity greater than zero, since in the program we are going to access the arguments in the structure of `B` and `C` with the builtin predicate `arg/3`.

The next time our ship program is preprocessed, having added the `calls` assertion, besides the pending error message of above regarding `arg/3`, we also get the following one:

```
ERROR: false assertion at set_precedences/3/1/4
       unexpected call (argument 2):
         Called:   ^array_starts
         Expected: struct
```

This message tells us the exact location of the bug, the fourth literal of the first clause for predicate `set_precedences/3`. This is because we have typed the constant `array_starts` instead of the variable `Array_starts` in such literal.

Thus, as shown in the example above, user-provided `check` assertions may help in locating the actual cause for an error. Also, as already mentioned, and maybe more obvious, user-provided assertions may allow detecting errors which are not easy to detect automatically otherwise.

After correcting the bug located in the previous example, preprocessing the program once again produces the following error message:

```
ERROR: false assertion at set_pre_lp/3/2/5
       unexpected call (argument 3):
         Called:   ^array_durations
         Expected: struct
```

which would not be automatically detected by the preprocessor without user-provided assertions. The obvious correction is to replace `array_durations` in the recursive call to `set_pre_lp` in its second clause with `Array_durations`.

After correcting this bug, preprocessing the program with the given assertions does not generate any more messages. Besides, the user provided `calls` assertion would have been proved by analysis.

Additionally, if some part of an assertion for a user-defined predicate has not been proved nor disproved during compile-time checking, it can be checked at run-time in the classical way, i.e., run-time tests are added to the program which encode in some way the given assertions. Introducing run-time tests by hand into a program is a tedious task and may introduce additional bugs in the program. In the preprocessor, this is performed automatically upon user's request.

Compile-time checking of assertions is conceptually more powerful than run-time checking. However, it is also more complex. Since the results of compile-time checking are valid for *any* query which satisfies the existing `entry` declarations,

compile-time checking can be used both to detect that an assertion is violated and to prove that an assertion holds for any valid query, i.e., the assertion is validated. The main problem with compile-time checking is that it requires the existence of suitable static analyses which are capable of proving the properties of interest. For conciseness, we have shown the possibilities of our system using only a (regular) type analysis. However, the system is generic in that any program property (for which a suitable analysis exists in the system) can be used for debugging. As mentioned before, currently CiaoPP can infer types, modes and other variable instantiation properties, constraint independence, non-failure of predicates, determinacy, bounds on computational cost, bounds on sizes of terms in the program, and other properties.

More info: For more information, full versions of selected papers and technical reports, and/or to download Ciao and other related systems please access `http://www.clip.dia.fi.upm.es/`.

Acknowledgments. This work has been funded in part by MCYT projects CUBICO (TIC02-0055), EDIPIA (TIC99-1151) and ADELA (HI2000-0043), and by EU IST FET project ASAP (IST-2001-38059).

References

1. K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.
2. K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.
3. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.
4. J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.
5. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
6. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
7. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
8. D. Cabeza and M. Hermenegildo. The Ciao Module System: A New Module System for Prolog. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

9. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
10. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.
11. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.
12. M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.
13. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
14. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
15. P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.
16. P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer-Verlag, September 2000.
17. W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.
18. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.
19. G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.
20. J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.
21. J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.
22. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.
23. M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.
24. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

25. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
26. P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.
27. Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.
28. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
29. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.
30. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
31. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.
32. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.
33. C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.
34. E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.
35. E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.

CGRASS: A System for Transforming Constraint Satisfaction Problems

Alan M. Frisch[1], Ian Miguel[1], and Toby Walsh[2]

[1] AI Group
Dept. Computer Science
University of York, York, England
{frisch,ianm}@cs.york.ac.uk
[2] Cork Constraint Computation Center
University College Cork
Ireland
tw@4c.ucc.ie

Abstract. Experts at modelling constraint satisfaction problems (CSPs) carefully choose model transformations to reduce greatly the amount of effort that is required to solve a problem by systematic search. It is a considerable challenge to automate such transformations and to identify which transformations are useful. Transformations include adding constraints that are implied by other constraints, adding constraints that eliminate symmetrical solutions, removing redundant constraints and replacing constraints with their logical equivalents. This paper describes the CGRASS (Constraint Generation And Symmetry-breaking) system that can improve a problem model by automatically performing transformations of these kinds. We focus here on transforming individual CSP instances. Experiments on the Golomb ruler problem suggest that producing good problem formulations solely by transforming problem instances is, generally, infeasible. We argue that, in certain cases, it is better to transform the problem class than individual instances and, furthermore, it can sometimes be better to transform formulations of a problem that are more abstract than a CSP.

1 Introduction

Constraint satisfaction is a successful technology for tackling a wide variety of search problems including resource allocation, transportation and scheduling. Constructing an effective model of a constraint satisfaction problem (CSP) is, however, a challenging task as new users typically lack specialised expertise. One difficulty is in identifying transformations, which are sometimes complex, that can dramatically reduce the effort needed to solve a problem by systematic search (see, for example, [15]). Such transformations include adding constraints that are implied by other constraints in the problem, adding constraints that eliminate symmetrical solutions to the problem, removing redundant constraints and replacing constraints with their logical equivalents. Unfortunately, outside a highly focused domain like planning (see, for example, [6]), there has been little research on how to perform such transformations automatically.

Our initial focus is on transforming individual CSP instances. The CGRASS system (Constraint GeneRation And Symmetry-breaking) is described and illustrated via the

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 15–30, 2003.

Golomb ruler problem [15], a difficult combinatorial problem with many applications. Our results suggest that making transformations to single problem instances alone is not practical on large instances. This can be remedied in two ways. First, by operating on a parameterised formulation of a problem *class*, which can be much more compact than the large instances in the class. In addition, every inference made holds for all instances of the class. A second improvement is to reason with formulations at higher levels of abstraction. We can construct *refinement* rules which, when applied to abstract formulations, produce effective concrete formulations. We conjecture that certain inferences will be easier at higher levels of abstraction. CGRASS provides a basic platform to achieve these goals. The rules we have developed form a template for creating new rules for similar reasoning about problems and at higher levels of abstraction.

2 Architecture of CGRASS

The implementation of CGRASS discussed here takes a problem instance as input. A problem instance consists of a finite set of domain variables, each with an associated finite domain (either explicitly or as bounds) and constraints over these variables. Output is created in the same simple format. Hence, very little effort is necessary to translate CGRASS' output into the required input for a variety of existing solvers.

CGRASS captures common patterns in the hand-transformation of constraint satisfaction problems in *transformation rules*. A transformation rule is a condition-action pair in the style of a production rule system (see [11], chapter 5). A rule-based architecture offers several potential advantages for the task of transforming CSPs automatically. Rules can be given very strong pre-conditions to limit the transformations to those that are likely to produce a problem that is simpler to solve. Furthermore, rules can act at a very high level. For example, they can perform complex rewriting, simplifications, and transformations. Such steps might require long fine-grained sequences of transformations to justify at the level of individual inference rules.

The structure of a CGRASS transformation rule can be demonstrated using an example. Consider the following pair of constraints, where the domains of x_1, x_2, x_3 are all $\{1, \ldots, 9\}$ and the domain of y is $\{1, \ldots, 100\}$:

$$x_1 + x_2 + x_3 = y$$
$$\text{allDifferent}(x_1, x_2, x_3).$$

Reasoning about the domains of x_1, x_2, x_3 and the fact that they are all-different gives:

$$6 \leq x_1 + x_2 + x_3$$
$$x_1 + x_2 + x_3 \leq 24$$

Substituting these inequalities back into the original equation clearly provides stronger bounds on y than bounds consistency alone.

The `allDiffSum` rule presented in Figure 1 achieves this transformation. The form of CGRASS' rules is adapted from the *methods* used to capture common proof patterns in proof planning [2]. The condition part of the rule consists of the input and pre-condition fields. The action part comprises the post-conditions and the add and delete lists, as explained below.

```
          Name: allDiffSum
         Input: algebraic(Constraint), allDifferent(Vars)
Pre-conditions: containsSimpleSum(Constraint, Sum),
                SumVars = involvesVars(Sum),
                supersetEq(Vars, SumVars)
Post-conditions: LB = allDiffLB(Sum),
                 UB = allDiffUB(Sum)
           Add: LB <= Sum, Sum <= UB
        Delete:
   Explanation: "Given ",Sum,"such that all variables
                 involved are all-different, we can infer",
                 LB<=Sum," and ",Sum<=UB
```

Fig. 1. The `allDiffSum` method.

The input field specifies a pattern against which a subset of the constraint set must match before the rule can be applied. CGRASS uses a rich pattern matching language specialised to reasoning about sets of constraints. This includes the ability to restrict matching to objects of certain types, such as a single constant or variable. CGRASS also supports several more powerful matching instructions, such as `algebraic(C)` and `allAlgebraic(C)`, which match C with an arbitrary algebraic constraint and the algebraic subset of the current constraint set respectively. In the example, `Constraint` matches $x_1 + x_2 + x_3 = y$ and `Vars` matches $\{x_1, x_2, x_3\}$.

Pre-conditions must be met before the post-conditions can be executed. Conditions are composed from primitive Boolean functions, such as $=$ (which also performs assignment), $!=$ (disequality) and $<$, as well as specialised functions which can be used to perform more complex operations on the input constraints. The specialised functions are written directly in Java (the native language of CGRASS), hence there is no restriction as to the operations that these functions can perform. The current set of specialised functions is readily extensible by writing new Java functions, although this does require some knowledge of how CGRASS works internally.

In the example of Figure 1, `containsSimpleSum(Constraint, Sum)` unifies `Sum` with a sub-term of `Constraint` composed of a sum with integer coefficients, i.e. $x_1 + x_2 + x_3$. `involvesVars(Sum)` returns the list of variables involved in `Sum`, which must be a subset or or equal to `Vars`. Finally, `allDiffLB(Sum)` and `allDiffUB(Sum)` calculate lower and upper bounds of a sum of all-different variables by considering the bounds of each individual variable involved. These bounds are used to add the two new constraints.

The size of the constraint set on which CGRASS operates neither increases nor decreases monotonically. This is because some of CGRASS' rules add new constraints, whereas others replace a constraint by a tighter one, or eliminate redundant constraints. For this reason, we replace the rule 'output' field normally used by proof planning by 'add' and 'delete' lists as used in classical planning. Both fields contain sets of constraints. In the case of the delete list, each element of this set is matched against and removed from the current constraint set.

In order for the user to see how a new model was derived, CGRASS' rules construct textual explanations of their application. For flexibility, sub-sections of the text can be predicated on the state of Boolean variables local to the rule.

2.1 Operation

CGRASS performs pure forward chaining, transforming one set of constraints into another. CGRASS' rules are sorted in descending order of priority. Given an input constraint set, CGRASS iterates over the sorted rule list to find the highest priority applicable rule. This rule is used to transform the constraint set, which is then used as input to the same process. Currently, CGRASS terminates when none of its rules are applicable. This is possible because of the strength of the pre-conditions attached to each rule. As the rule database grows in size and complexity, however, this may be insufficient. At some point the decision must be made to stop making transformations and start searching for a solution. We may in the future have to add an executive in the style of a proof critic [10] which terminates CGRASS when future rewards look poor.

Non-monotonicity carries with it the danger of looping. Unless a rule deletes some of its input constraints, its preconditions continue to hold. Hence, the rule can repeatedly fire *ad infinitum*. To avoid the problem CGRASS employs a history mechanism, which maintains a record of all constraints that have been added to the constraint set. Rules are not allowed to add a constraint which was added to the constraint set previously, even if the constraint in question was subsequently removed. The intuition behind this approach is that a constraint is removed either if it is redundant or it is transformed into some more useful form. Restoring a previously removed constraint is therefore a retrograde step.

2.2 Normalisation

CGRASS transforms the constraint set into a normal form at the start of operation and any time the constraint set is modified. The normal form used is inspired by that used in the HARTMATH computer algebra system[1]. This enables CGRASS to deal easily with associative and commutative operators, allowing it to test for simple syntactic equivalence instead of semantic equivalence. Normalisation also reduces the number and complexity of rules needed. For example, inequalities are always rearranged into the form $x < y$ or $x \leq y$. Hence the input to a rule never has to match $y > x$ or $y \geq x$, halving the number of rules in some cases.

We define a total order over the types of expressions that CGRASS supports. The constraint set is transformed into a minimal state with respect to this order. Constants are at the top of the order, followed by variables, fractions, sums and products. Further down the order are constraint types such as equalities, disequations, inequalities and special constraints such as 'all-different'. Expressions of different type are ordered via their position in the order. Expressions of the same type are ordered recursively; each type has an associated rule of self-comparison. The base case is where two constants or two variables are compared. In the former case, the comparison is by value, with least first and in the latter the comparison is lexicographically by name. Sums and products are represented in a 'flattened' form, hence their arguments are simply sorted using the above comparison to maintain a lexicographic order. Similarly, the lexicographically least side of an equation or disequation is forced to be the left hand side.

As an example, consider the following pair of constraints:

$$x_8 + x_7 \not= x_6 + x_5$$
$$x_4 * 2 + x_3 = 2 * x_1 + x_2$$

[1] `http://www.hartmath.org`

Transforming them to normal form results in:

$$x_2 + 2 * x_1 = x_3 + 2 * x_4$$
$$x_5 + x_6 \not= x_7 + x_8$$

Equality is higher in the type order than disequality, hence the re-ordering of the two constraints. The sums are ordered internally and recursively, then re-ordered as appropriate to the constraint.

Simplification procedures consist of the collection of like terms, cancellation and the removal of common factors. Consider the following example:

$$2 * 6 * x_1 + 4 * x_2 = 6 * x_1 + x_3 * 2 * 2 + 6 * x_1$$

Following lexicographical ordering, we collect the constants and occurrences of x_1:

$$4 * x_2 + 12 * x_1 = 4 * x_3 + 12 * x_1$$

Next we perform cancellation:

$$4 * x_2 = 4 * x_3$$

Finally, we remove the common factor:

$$x_2 = x_3$$

Lexicographic ordering and simplification are interleaved until no further change is possible. They reduce the workload of CGRASS substantially, both in providing a syntactic test for equality and avoiding such simplification routines being written as explicit rules. The latter saving is substantial: a larger rule base means more work in matching against the constraint set at each iteration of the CGRASS inference loop.

3 Transformation Rules

We illustrate the transformation rules currently implemented in CGRASS via a small instance of the Golomb ruler problem (available at `http://www.csplib.org` as `prob006`). A Golomb ruler is a set of n ticks at integer points on a ruler of length m such that all the inter-tick distances are unique. Given n, the problem is to minimise m. The longest known optimal ruler has 21 marks and is of length 333. Such rulers have practical applications in radio astronomy and X-ray crystallography [5]. Smith et al. [14] used the Golomb ruler as the basis of an interesting exercise in modelling CSPs and identified a number of implied constraints by hand.

We begin with a concise model of the problem with n ticks represented by variables $x_1, \ldots, x_n$, each with domain $\{0, \ldots, n^2\}$. Note that n^2 is an empirical upper bound, sufficient for small instances, that we use throughout. It is easy to show that 2^n is a conservative upper bound in general. Alternatively, some solvers allow domains with no upper bound, which are suitable for this type of minimisation problem.

minimise: $max_i(x_i)$

$$\{(x_i - x_j \not= x_k - x_l) \mid i, j, k, l \in [1, n] \wedge (i \not= j) \wedge (k \not= l) \wedge (i \not= k \vee j \not= l)\}$$

The second element is a set of constraints, each element of which is input to CGRASS. Taken literally, this is a poor model. The constraints are quaternary, and will be delayed by most solvers. There is also a large amount of symmetry present, some of which is discussed below. However, it serves to illustrate how CGRASS can make a substantial improvement to a basic model.

We focus on the 3-tick ruler for the purpose of this example. The basic model produces 30 constraints. CGRASS' initial normalisation of the constraint set immediately reduces this number to 12. This is achieved in two ways. Firstly, constraints with reflection symmetry across a disequation are identical following normalisation, hence only one copy is kept. Secondly, multiple constraints can simplify to the same constraint. For example,

$$x_1 - x_2 \not= x_1 - x_3, \text{ and } x_2 - x_1 \not= x_3 - x_1$$

both simplify to $x_2 \not= x_3$.

Hence, a large saving is made before CGRASS has performed any rule application. Table 1 presents the formulation of the problem at this point. The 'minimise' statement is omitted throughout for brevity.

Table 1. 3-tick Golomb ruler. Initial formulation following normalisation.

$x_1 \neq x_2$	$x_1 \neq x_3$	$x_2 \neq x_3$
$x_1 - x_2 \neq x_2 - x_1$	$x_1 - x_2 \neq x_2 - x_3$	$x_1 - x_2 \neq x_3 - x_1$
$x_1 - x_3 \neq x_2 - x_1$	$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$
$x_2 - x_1 \neq x_3 - x_2$	$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$

3.1 Symmetry Breaking

Symmetry is inherent in many CSPs. Given a set of symmetrical variables, it is possible to permute their assignments in a (non)solution to obtain another (non)solution. This can lead to expensive exploration of fruitless branches of the search tree. Hence, it is important to be able to remove or reduce symmetries automatically. CGRASS does this by adding symmetry breaking constraints. Although symmetry breaking constraints are not implied themselves (they do not follow from the initial model), they can be useful for generating further implied constraints. Indeed, often the most useful constraints can be derived only after some or all symmetry has been broken. Hence, CGRASS attempts to detect and break symmetry as a pre-processing step.

CGRASS begins by looking for symmetrical variables, i.e. pairs of variables with identical domains such that, if all occurrences of this pair in the constraint set are exchanged and the constraint set is re-normalised, it returns to its original state. Candidate variables with the same number of occurrences in the constraint set are first grouped together before comparisons are made. The transitivity of symmetry is exploited to minimise the number of pairs of variables that are compared. Efficiency is further improved by making pairwise comparisons of normalised constraint sets.

This process partitions the variables into symmetry classes. The elements of each class are formed into a list, and ordered lexicographically. For each list of variables, say $x_1, \ldots, x_n$, symmetry is broken by adding the constraints

$$x_1 \leq x_2, \quad x_2 \leq x_3, \quad \ldots, \quad x_{n-1} \leq x_n$$

Implied inequalities, such at $x_1 \leq x_3$, are not useful since enforcing bounds consistency on the original set of constraints also enforces generalised arc consistency on the implied constraint.

Symmetry testing on the 3-tick ruler reveals that the variables x_1, x_2 and x_3 are symmetrical. This symmetry can be broken by adding two constraints: $x_1 \leq x_2$ and $x_2 \leq x_3$.

It is also possible to identify symmetries among non-atomic terms. This is potentially an expensive process, hence CGRASS adopts a heuristic approach, only comparing terms that are likely to be symmetrical. These heuristics are based on *structural equivalence*. Two terms are structurally equivalent if they are identical when explicit variable names in each are replaced with a common indistinguishable marker. For example,

$$\frac{x_1}{x_2 * x_3}, \frac{x_4}{x_5 * x_6}$$

become:

$$\frac{\#}{\# * \#}, \frac{\#}{\# * \#}$$

and are therefore structurally equivalent. Each pair of variables, x_1 and x_4, x_2 and x_5, and x_3 and x_6 are exchanged throughout the constraint set before re-normalisation and a check for equivalence. This process does not reveal any further symmetries in the example problem, but is useful in general (see [8], for example).

Working from the symmetry breaking constraints above, CGRASS fires the rule `strengthenInequality`, as presented in Figure 2. This is one example of a number of simple but useful rules to which CGRASS ascribes a high priority during rule selection. Other examples are various instances of the rules `nodeConsistency` and `boundsConsistency` which deal with the filtering of domain elements. These rules are not only cheap to fire, but often result in a reduction in the size of the constraint set. This promotes efficiency by leaving fewer constraints for the more complicated rules to attempt to match against.

Indeed, the `boundsConsistency` rule can now fire, pruning the domains of x_1, x_2 and x_3 according to their strict ordering. This leaves the problem in the formulation as presented in Table 2. Clearly, $x_1 \not= x_3$ is redundant. One could foresee the addition of a relatively simple rule that takes as input a set of strict inequalities and a disequation in order to detect and remove such a redundancy.

3.2 Introduce

The model as it stands still contains 9 quaternary constraints. One powerful means of reducing the arity of these constraints is to introduce one or more new variables which the `eliminate` rule (see below) then uses to replace sub-terms within them. Therefore, we have developed the `introduce` rule, as presented in Figure 3. Since this rule introduces

Conditions:

1. There exist two expressions, $x \neq y$ and $x \leq y$

Actions:

1. Add a new constraint of the form, $x < y$
2. Delete $x \neq y$ and $x \leq y$

Fig. 2. The `strengthenInequality` rule.

Table 2. Formulation following symmetry-breaking and bounds consistency.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$x_1 - x_2 \neq x_2 - x_1$	$x_1 - x_2 \neq x_2 - x_3$	$x_1 - x_2 \neq x_3 - x_1$
$x_1 - x_3 \neq x_2 - x_1$	$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$
$x_2 - x_1 \neq x_3 - x_2$	$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$

new terms, it has a potentially explosive effect. CGRASS therefore assigns it a very low priority, only attempting to introduce new variables when all the simpler rules, which tend to have a reductive effect, are inapplicable. In addition, complex preconditions are attached to `introduce` to prevent its application unless there is strong evidence that the new variable will be useful. First of all, we insist that the sub-term, *Exp*, under consideration contains at least two variables; efficiency is unlikely to be gained if Exp contains fewer than two variables.

Secondly, variables which occur frequently have a wider reaching effect when propagation is performed on them. We require *Exp* to occur at least twice in the constraint set before it can be considered for replacement by a variable. Finally, we check that some other variable is not already defined to be equal to *Exp*. If these conditions are

Conditions:

1. There exists a sub-term, *Exp*, containing two or more variables that occurs more than once.
2. *someVariable* = *Exp* is not already present in the constraint set.

Actions:

1. Generate a new variable, x, with domain derived from *Exp*.
2. Add a constraint of the form $x = Exp$.

Fig. 3. The `introduce` rule.

met, CGRASS generates a new variable, x, calculating the bounds of its domain from the upper and lower bounds on *Exp*.

The sub-term $x_1 - x_2$ in the example meets the input preconditions of `introduce`. CGRASS introduces a new variable, z_0, with domain {-8 .. 6} and imposes the constraint $z_0 = x_1 - x_2$. In order to make use of z_0, however, the companion `eliminate` rule is necessary.

3.3 Eliminate

We have developed multiple versions of `eliminate`, using equalities (Figure 4) and inequalities to perform Gaussian-like elimination of a particular sub-term.

Conditions:

1. There exists a constraint *Lhs* = *CommonExp* such that *CommonExp* is also present in a constraint, c, in the constraint set.
2. Constraint c_{new} is obtained by replacing all occurrences of *CommonExp* by *Lhs* in c.
3. The (post-normalisation) size of c_{new} is less than that of c.
4. c_{new} is not obviously redundant.

Actions:

1. Add c_{new} to the constraint set.
2. Remove c from the constraint set.

Fig. 4. The `Eliminate(equality)` rule.

To ensure that `eliminate` has a reductive effect, the resulting constraint must have a smaller number of constituent terms than the original. Also, we perform simple checks for redundancy such as, in the case of equality, the left hand side being syntactically identical to the right hand side. Finally, when eliminating with equality the original constraint is removed following elimination in order to avoid cluttering the constraint set. `Eliminate` has a higher priority than `introduce`: there is no point in introducing variables for common terms that will be eliminated anyway.

Following the introduction of $z_0 = x_1 - x_2$ in the example, various instances of `eliminate` can fire. For instance, `eliminate(equality)` can be used to substitute z_0 into a number of the quaternary disequations, reducing the complexity of each. Furthermore, `eliminate(inequality)` eliminates x_1 in favour of x_2 in $z_0 = x_1 - x_2$, using $x_1 < x_2$ to give: $z_0 < 0$. This unary constraint immediately triggers the `nodeConsistency` rule, reducing the domain of z_0 to {-8 .. 0}. The problem is left in the formulation presented in Table 3.

CGRASS now introduces, and eliminates with, a further two variables, $z_1 = x_2 - x_3$ and $z_2 = x_3 - x1$. This leads to the much-improved formulation presented in Table 4.

Table 3. Formulation following introduction of and elimination with z_0.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_0 \neq x_2 - x_3$	$z_0 \neq x_3 - x_1$
$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$	$x_1 - x_3 \neq -z_0$
$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$	$x_3 - x_2 \neq -z_0$

Table 4. Formulation following introduction of and elimination with z_1, z_2.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_1 = x_2 - x_3$	$z_2 = x_3 - x_1$
$z_0 \neq z_1$	$z_0 \neq z_2$	$z_1 \neq z_2$

3.4 All-Different

One further rule available to CGRASS is `genAllDiff` which, as its name suggests, attempts to generate an all-different constraint from a clique of not-equals constraints. An all-different constraint is desirable because of the powerful propagation rules available for it within constraint solvers [13]. Since maximal-clique identification is an NP-complete problem, CGRASS uses a fast approximation algorithm [1] to find a clique quickly. Typically this is the maximal clique.

In the example, `genAllDiff` successfully replaces the disequations involving the z variables with a single all-different constraint. This leads to the final problem formulation, as shown in Table 5. This rule has a lower priority than `introduce`: waiting for `introduce` to be exhausted maximises the chance of finding the largest clique of disequations.

Table 5. Final formulation.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_1 = x_2 - x_3$	$z_2 = x_3 - x_1$
all-different(z_0, z_1, z_2)		

4 Results

We compared the performance of Ilog Solver to find and prove optimality on basic and transformed models of 6 instances of the Golomb ruler problem. The transformed models

are similar to that presented in Table 5. Results are given in Table 6 and Figure 5. The smallest instances are so easy to solve that it is not worth the effort of transformation. For larger instances, however, the transformed model becomes significantly easier to solve, with the gap in performance increasing rapidly with n, the number of ticks.

The number of input constraints generated from the basic model also increases significantly with n. Unsurprisingly, this is accompanied by a marked increase in the time required by CGRASS to make the transformations. On the smaller instances tested, this means that the total time for transformation and solution exceeds the time for solution of the basic model alone. However, as n increases, the time required by CGRASS grows more slowly than the time taken to solve the basic model. Hence, at larger (and therefore more interesting) values of n, a net benefit is evident. Furthermore, the effects of a relatively simple implementation are also apparent: as the size of the input grows, CGRASS' rule application rate quickly deteriorates. A more sophisticated implementation that avoids repeated blind traversal of the constraint set would provide a considerable improvement in CGRASS' performance.

One way of overcoming the problem of overwhelming CGRASS on an input instance with many constraints is to use it interactively. Given the final model of the 3-tick ruler, it is not difficult for a human to see how this model could be generalised to a formulation for the entire problem class. The comparative results of the basic and final models presented in Table 6 indicate that the effort expended on such a process could easily be justified as n grows larger. A machine learning tool such as HR [3] or Progol [12] might be used to aid in the generalisation process. Some promising results in this regard are reported in [4].

5 Transforming Problem Formulations

The previous section shows that the cost of transforming a problem instance can grow rapidly with its size. A way around this difficulty would be to transform a formulation of the problem rather than a formulation of an instance. Problem formulations usually involve parameters and quantification (or set comprehension) for expressing sets of constraints, so to automate the transformation of problem formulations we must "lift" the CGRASS transformation rules to handle these constructs.

Table 6. Results: Golomb ruler. Hardware: 2GHz Athlon XP, 256Mb RAM. Software: Java 1.4.1, Ilog Solver 5.2.

	Ticks	3	4	5	6	7	8
Basic Model	Size (constraints)	31	133	381	871	1,723	3,081
	Choice-points	12	107	3,637	111,101	4,602,921	160,644,147
	Solution Time	0.01s	0.01s	0.4s	12.3s	1,220s	126,000s
GRASS	Rule Applications	34	120	342	796	1,603	2,917
v1.0	Time	0.1s	1.3s	20s	236s	2,030s	11,600s
Transformed	Choice-points	5	10	76	561	5,402	46,866
Model	Time	0.01s	0.01s	0.01s	0.04s	0.2s	2.6s

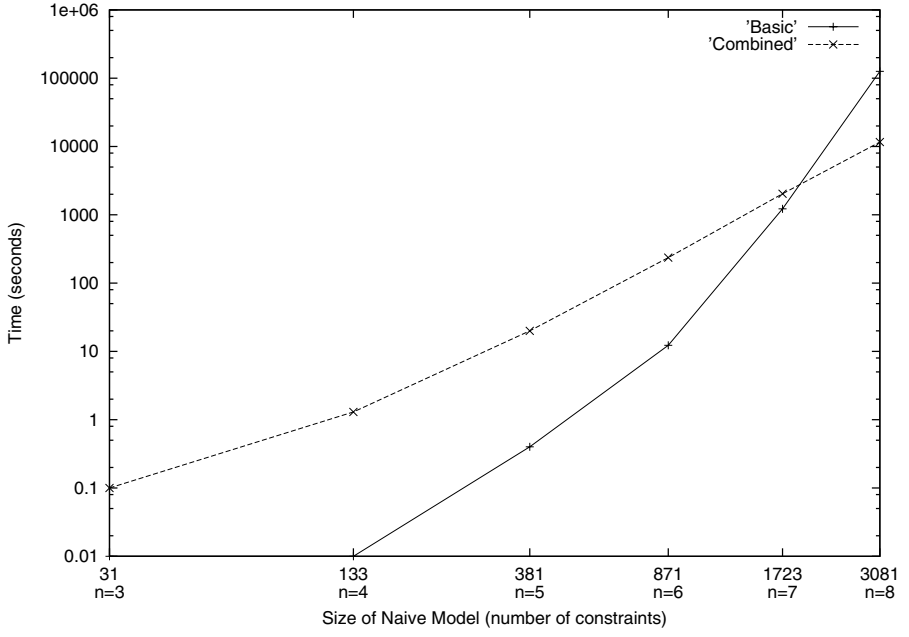


Fig. 5. Shown on a log-log scale are the run-times for solving the basic model versus combined time of CGRASS transformation and solution of transformed model. Each point on the x axis is the problem instance for a given n.

Direct support for quantified constraint expressions would immediately reduce the size of the input in our Golomb ruler example. One such formulation of the problem is:

$$\text{minimise: } max_i(x_i)$$
$$\{(x_i - x_j \not= x_k - x_l) \mid i,j,k,l \in [1,n] \wedge (i \not= j) \wedge (k \not= l) \wedge (i \not= k \vee j \not= l)\}$$

A further benefit is the ability to reason about an entire class of problems rather than particular instances. This is clearly more efficient than repeating a large amount of work for each instance under consideration. However, we should not abandon reasoning about instances altogether; it is likely that some transformations will be valid only for a particular instances of a class. Therefore, we intend to extend—rather than replace—CGRASS' library of transformations to support transformations at the problem level.

Following the introduction of quantified constraint expressions, some transformations remain difficult. For example, one of the key transformations made in Section 4 recognised the symmetry of the tick variables and broke this symmetry via the introduction of weak inequalities. By inspection, detecting this symmetry in the quantified formulation given above is a difficult task. Given the importance of symmetry-breaking, we believe that it is advantageous to start with the problem represented in a significantly more abstract language, much more abstract than, for example, OPL [16].

5.1 Reformulating a Highly Abstract Problem Formulation

This section describes how we might reformulate an abstract description of the Golomb ruler problem, one which is not formally a CSP, into an efficient CSP formulation for input to a constraint solver. Such a reformulation involves two kinds of operations: *refinements* that replace an abstract construct with one that is more concrete, and *transformations*, such as those currently performed by CGRASS, that improve the efficiency of a formulation but do not change its level of abstraction. As we have not yet specified rules to perform these reformulations, our aim here is to illustrate the kind of reformulations that we plan to embody in our rules.

We begin with a natural language description of this problem:

- Given n, put n ticks on a ruler of size m such that all the inter-tick distances are unique. Minimise m.

We cannot expect CGRASS to work with this level of input, hence the user must make the initial transformation shown below. We chose this formulation because it closely mimics the natural language statement of the problem.

1. Given n find $T \subseteq \{0, \ldots, n^2\}$ subject to:
2. Minimise $max(T)$
3. $|T| = n$
4. $\{\text{distance}(\{x, y\}) \not= \text{distance}(\{x', y'\}) | \{x, y\} \subseteq T, \{x', y'\} \subseteq T, \{x, y\} \not= \{x', y'\}\}$
5. $\{\text{distance}(\{x, y\}) = |x - y| \mid \{x, y\} \subseteq T\}$

Note that $\{x, y\}$ denotes a set of size 2 and that (5) is not part of the natural language specification, but rather encodes user knowledge of the definition of distance.

A crucial feature of this abstract formulation is that it does not distinguish individual ticks in the way that the x_i tick variables of Section 4 do. The x_i variables distinguish ticks by saying "this is the first tick" and "this is the second tick" and so forth. Distinguishing otherwise indistinguishable objects introduces symmetry into a problem which, for the sake of efficiency, must then be broken. Our approach is to start with a formulation that is sufficiently abstract that it does not distinguish otherwise indistinguishable objects. As the formulation is refined to a CSP these objects will necessarily become distinguished since in a CSP all objects are distinguished. Our view is that refinement rules should introduce symmetry breaking constraints at the point they introduce symmetry, that is, at the point they introduce distinction among otherwise indistinguishable objects.

We begin by replacing the occurrences of $distance$ in (4) by the definition given in (5).

6. $\{|x - y| \not= |x' - y'| \mid \{x, y\} \subseteq T, \{x', y'\} \subseteq T, \{x, y\} \not= \{x', y'\}\}$

Now we no longer need the definition, so we discard (5) from the formulation.

Next, we need a general refinement rule:

> To refine a set variable of fixed cardinality n drawn from a set A of size m, totally ordered by $\leq$, introduce a set S of n decision variables, $\{s_1, \ldots, s_n\}$. The domain of each s_i is A. Break symmetry among the s_i variables by adding the constraint $s_1 < s_2 < \cdots < s_n$. We could also build into this the simple bounds consistency argument by taking the domain of each s_i to be $\{A_i, \ldots, A_{m-n+i}\}$, where A_i is the i^{th} element in the ordering of the elements of A.

Applying this rule results in the following problem formulation:

10. Given n, find $S = \{s_1, s_2, \ldots, s_n\}$,
    where each s_i has domain $\{i-1, \ldots, n^2 - n + i - 1\}$.
11. $s_1 < s_2 < \cdots < s_n$
12. Minimise($max(S)$)
13. $\{|x - y| \not= x' - y' \mid \{x, y\} \subseteq S, \{x', y'\} \subseteq S, \{x, y\} \not= \{x', y'\}\}$

In the case of our example, a worthwhile transformation is to introduce a set of distance variables, $d_{\{x,y\}}$.

14. $\{d_{\{x,y\}} = |x - y| \mid \{x, y\} \subseteq S\}$

This is a lifted form of the introduce rule of Section 4. We can now perform a lifted version of eliminate; substituting (14) into (13) gives:

15. $\{d_{\{x,y\}} \not= d_{\{x',y'\}} \mid \{x, y\} \subseteq S, \{x', y'\} \subseteq S, \{x, y\} \not= \{x', y'\}\}$

It should not be difficult to notice that (15) defines a clique. A lifted version of our all-different introduction rule would replace (15) with:

16. all-diff($\{d_{\{x,y\}} | \{x, y\} \subseteq S\}$)

At this level of abstraction we now use symmetry breaking constraint (11) to simplify (12) to:

17. Minimise(s_n)

The set of all variables x and y such that $\{x, y\} \subseteq S$ is equivalent to the set of pairs of variables x and y such that $x \prec y$ and $x, y \in S$, where $\prec$ is an arbitrary total ordering of S. In this particular case, we choose $\prec$ to be defined as $s_i \prec s_j \leftrightarrow i < j$. Below we shall see the significance of this choice. Hence, (14) and (16) can be refined to:

18. all-diff($\{d_{s_i,s_j} | s_i, s_j \in S \text{ and } i < j\}$)
19. $\{d_{s_i,s_j} = |s_i - s_j| \mid s_i, s_j \in S \text{ and } i < j\}$

Now, using (11), (19) can be simplified to

20. $\{d_{s_i s_j} = s_j - s_i \mid s_i, s_j \in S \text{ and } i < j\}$

The final problem formulation is as follows.

- Given n find S=$\{s_1, s_2, \ldots, s_n\}$,
  where each s_i has domain $\{i-1, \ldots, n^2 - n + i - 1\}$.
- $s_1 < s_2 < \cdots < s_n$
- Minimise(s_n)
- all-diff($\{d_{s_i,s_j} | s_i, s_j \in S \text{ and } i < j\}$)
- $\{d_{s_i,s_j} = s_j - s_i \mid s_i, s_j \in S \text{ and } i < j\}$

With n taken as three, it is equivalent to the final representation in Table 5. Symmetry amongst the tick variables has been broken, and all-different difference variables for the inter-tick distances have been introduced.

Working with a more abstract input language that uses sets allows us to avoid the problem of detecting symmetry. We also retain the advantage of making valid transformations for the whole problem class. Furthermore we avoid the overwhelming size of the input that inevitably causes slowdown when transforming naive representations of individual instances. Refinement rules allow us to move to progressively more concrete levels of abstraction as necessary to perform transformations at the appropriate level.

6 Conclusions

We have described CGRASS, a system for the automatic transformation of a naive model of a constraint satisfaction problem into one that requires significantly less effort to solve. CGRASS adopts a rule-based architecture, transforming a model via the application of rules which encapsulate modelling expertise. The set of rules described here should be viewed as a representative sample. It is not complete in any sense, and we will continue to extend it in future.

The current implementation of CGRASS is able to transform individual instances only. Results obtained from experiments on the Golomb ruler problem suggest that this approach is impractical in general. We have discussed the need for the ability to reformulate problem classes, both to avoid dealing with large instances and in order that the inferences made hold for every instance of the class. We have also argued that it is important to reason at higher levels of abstraction. Refinement rules allow the systematic creation of effective concrete models from abstract formulations, and we conjecture that transformations at higher levels of abstraction will sometimes prove easier than at more concrete levels. As an example, we have outlined how the Golomb ruler problem might be effectively transformed from a high level description into a good model. Space prevents us from giving further examples here, but elsewhere [7] we have applied a similar process to the SONET network design problem.

A principal element of future work is to extend CGRASS' set of rules to allow it to reason about parameterised problem classes and at higher levels of abstraction. In particular, we will introduce refinement rules to move from higher to lower levels of abstraction as necessary. We will also consider whether CGRASS' rules can be implemented via constraint handling rules [9]. Possible obstacles include more complex pattern matching, the use of best-first search as the rule base grows in complexity, and the use of an executive which decides when to stop inferring and start searching.

Acknowledgements. This project is supported by EPSRC Grant GR/N16129[2]. The third author is supported by Science Foundation Ireland. We thank Brahim Hnich for discussions about refinement. Julian Richardson's adaptation of PRESS to manipulate inequalities and its success at automatically generating a number of implied constraints influenced our approach. Finally we thank our anonymous referees for their useful comments.

References

1. R. Boppana and M. M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 32:180–196, 1992.
2. A. Bundy. A science of reasoning. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.
3. S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag, 2002.
4. S. Colton and I. Miguel. Constraint generation via automated theory formation. In T. Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 575–579, 2001.

[2] `http://www.cs.york.ac.uk/aig/projects/implied/index.html`

5. A.K. Dewdney. Computer recreations. *Scientific American*, pages 16–20, December 1985.
6. M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1169–1176, 1997.
7. A.M. Frisch, B. Hnich, I. Miguel, B.M. Smith, and T. Walsh. Towards model reformulation at multiple levels of abstraction. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems*, pages 42–56, 2002.
8. A.M. Frisch, I. Miguel, and T. Walsh. Extensions to proof planning for generating implied constraints. In *Proceedings of Calculemus-01*, pages 130–141, 2001.
9. T. Frühwirth. Theory and practice of constraint handling rules. In P. Stuckey and K. Marriot, editors, *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, volume 37(1–3), pages 95–138, 1998.
10. A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proof. In *Proceedings of LPAR'92, Lecture Notes in Artificial Intelligence 624*. Springer-Verlag, 1992. Also available as Research Report 592, Dept of AI, Edinburgh University.
11. G.F. Luger and W. A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 1998.
12. S Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
13. J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on AI*, pages 362–367. American Association for Artificial Intelligence, 1994.
14. B. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the 16th National Conference on AI*, pages 182–187. AAAI, 2000.
15. B.M. Smith, K. Stergiou, and T. Walsh. Modelling the Golomb ruler problem. In *Proceedings of the IJCAI-99 Workshop on Non-Binary Constraints*. International Joint Conference on Artificial Intelligence, 1999.
16. P. van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

Interchangeability in Soft CSPs

Stefano Bistarelli[1,2], Boi Faltings[3], and Nicoleta Neagu[3]

[1] Istituto di Informatica e Telematica, CNR, Pisa, Italy
Stefano.Bistarelli@iit.cnr.it,
[2] Dipartimento di Scienze, Universitá "G. D'annunzio", Pescara, Italy
bista@sci.unich.it,
[3] Artificial Intelligence Laboratory (LIA), EPFL, Ecublens, Switzerland
boi.faltings|nicoleta.neagu@epfl.ch

Abstract. Substitutability and interchangeability in constraint satisfaction problems (CSPs) have been used as a basis for search heuristics, solution adaptation and abstraction techniques. In this paper, we consider how the same concepts can be extended to *soft* constraint satisfaction problems (SCSPs).
We introduce two notions: *threshold* α and *degradation* δ for substitutability and interchangeability, ($_\alpha$substitutability/interchangeability and $^\delta$substitutability/interchangeability respectively). We show that they satisfy analogous theorems to the ones already known for hard constraints. In $_\alpha$interchangeability, values are interchangeable in any solution that is better than a threshold α, thus allowing to disregard differences among solutions that are not sufficiently good anyway. In $^\delta$interchangeability, values are interchangeable if their exchange could not degrade the solution by more than a factor of δ.
We give efficient algorithms to compute ($^\delta/_\alpha$)interchangeable sets of values for a large class of SCSPs.

1 Introduction

Substitutability and interchangeability in CSPs have been introduced by Freuder ([12]) in 1991 with the intention of improving search efficiency for solving CSP. Interchangeability has since found other applications in abstraction frameworks ([14,20,12,8]) and solution adaptation ([19,15]). One of the difficulties with interchangeability has been that it does not occur very frequently.

In many practical applications, constraints can be violated at a cost, and solving a CSP thus means finding a value assignment of minimum cost. Various frameworks for solving such soft constraints have been proposed [13,10,16,11, 18,5,6,2]. The soft constraints framework of c-semirings [5,2] has been shown to express most of the known variants through different instantiations of its operators, and this is the framework we are considering in this paper.

The most straightforward generalization of interchangeability to soft CSP would require that exchanging one value for another does not change the quality of the solution at all. This generalization is likely to suffer from the same weaknesses as interchangeability in hard CSP, namely that it is very rare.

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 31–46, 2003.

Fortunately, soft constraints also allow weaker forms of interchangeability where exchanging values may result in a degradation of solution quality by some measure δ. By allowing more degradation, it is possible to increase the amount of interchangeability in a problem to the desired level. We define ${}^{\delta}$substitutability/interchangeability as a concept which ensures this quality. This is particularly useful when interchangeability is used for solution adaptation.

Another use of interchangeability is to reduce search complexity by grouping together values that would never give a sufficiently good solution. In ${}_{\alpha}$substitutability/interchangeability, we consider values interchangeable if they give equal solution quality in all solutions better than α, but possibly different quality for solutions whose quality is $\leq \alpha$.

Just like for hard constraints, full interchangeability is hard to compute, but can be approximated by neighbourhood interchangeability which can be computed efficiently and implies full interchangeability. We define the same concepts for soft constraints, and prove that neighborhood implies full (${}^{\delta}/_{\alpha}$)substitutability/interchangeability. We give algorithms for neighborhood (${}^{\delta}/_{\alpha}$)substitutability/interchangeability, and we prove several interesting and useful properties of the concepts. Finally, we give two examples where (${}^{\delta}/_{\alpha}$)interchangeability is applied to solution adaptation in configuration problems with two different soft constraint frameworks: delay and cost constraints, and show its usefulness in these practical contexts.

2 Background

2.1 Soft CSPs

Several formalization of the concept of *soft constraints* are currently available. In the following, we refer to the one based on c-semirings [2,4,5,7], which can be shown to generalize and express many of the others [3]. A soft constraint may be seen as a constraint where each instantiations of its variables has an associated value from a partially ordered set which can be interpreted as a set of preference values. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ($\times$) and comparison ($+$) of tuples of values and constraints. This is why this formalization is based on the concept of c-semiring, which is just a set plus two operations.

Semirings. A semiring is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that: 1. A is a set and $\mathbf{0}, \mathbf{1} \in A$; 2. $+$ is commutative, associative and $\mathbf{0}$ is its unit element; 3. $\times$ is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element. A c-semiring is a semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that: $+$ is idempotent, $\mathbf{1}$ is its absorbing element and $\times$ is commutative. Let us consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that (see [5]): 1. $\leq_S$ is a partial order; 2. $+$ and $\times$ are monotone on $\leq_S$; 3. $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; 4. $\langle A, \leq_S\rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = lub(a, b)$ (where *lub* is the *least upper bound*). Moreover, if $\times$ is idempotent, then: $+$ distributes

over $\times$; $\langle A, \leq_S \rangle$ is a complete distributive lattice and $\times$ its *glb* (*greatest lower bound*). Informally, the relation $\leq_S$ gives us a way to compare semiring values and constraints. In fact, when we have $a \leq_S b$, we will say that b *is better than* a. In the following, when the semiring will be clear from the context, $a \leq_S b$ will be often indicated by $a \leq b$.

Constraint Problems. Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a finite domain D, a *constraint* is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring. By using this notation we define $\mathcal{C} = \eta \to A$ as the set of all possible constraints that can be built starting from S, D and V.

Note that in this *functional* formulation, each constraint is a function (as defined in [7]) and not a pair (as defined in [4,5]). Such a function involves all the variables in V, but it depends on the assignment of only a finite subset of them. So, for instance, a binary constraint $c_{x,y}$ over variables x and y, is a function $c_{x,y} : V \to D \to A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$. We call this subset the *support* of the constraint. More formally, consider a constraint $c \in \mathcal{C}$. We define its support as $supp(c) = \{v \in V \mid \exists \eta, d_1, d_2 . c\eta[v := d_1] \not= c\eta[v := d_2]\}$, where

$$\eta[v := d]v' = \begin{cases} d & \text{if } v = v', \\ \eta v' & \text{otherwise.} \end{cases}$$

Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the assignment $v := d_1$ (that is the operator [ ] has precedence over application). Note also that $c\eta$ is the application of a constraint function $c : V \to D \to A$ to a function $\eta : D \to A$; what we obtain, is a semiring value $c\eta = a$.

A *soft constraint satisfaction problem* is a pair $\langle C, con \rangle$ where $con \subseteq V$ and C is a set of constraints: *con* is the set of variables of interest for the constraint set C, which however may concern also variables not in *con*. Note that a classical CSP is a SCSP where the chosen c-semiring is: $S_{CSP} = \langle \{false, true\}, \vee, \wedge, false, true \rangle$. Fuzzy CSPs [17] can instead be modeled in the SCSP framework by choosing the c-semiring $S_{FCSP} = \langle [0, 1], max, min, 0, 1 \rangle$. Many other "soft" CSPs (Probabilistic, weighted, ...) can be modeled by using a suitable semiring structure ($S_{prob} = \langle [0, 1], max, \times, 0, 1 \rangle$, $S_{weight} = \langle \mathcal{R}, min, +, +\infty, 0 \rangle$, ...).

Fig. 1 shows the graph representation of a fuzzy CSP. Variables and constraints are represented respectively by nodes and by undirected (unary for c_1 and c_3 and binary for c_2) arcs, and semiring values are written to the right of the corresponding tuples. The variables of interest (that is the set *con*) are represented with a double circle. Here we assume that the domain D of the variables contains only elements a and b and c.

Combining and projecting soft constraints. Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times_S c_2\eta$. In words, combining two constraints means building a new constraint whose support involves all the

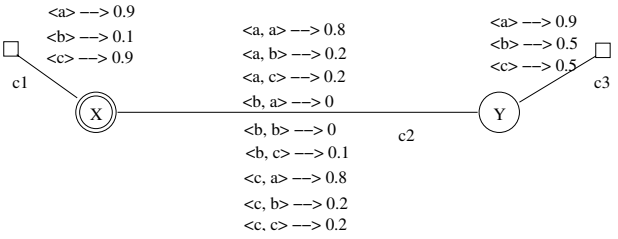


Fig. 1. A fuzzy CSP.

variables of the original ones, and which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate subtuples. It is easy to verify that $supp(c_1 \otimes c_2) \subseteq supp(c_1) \cup supp(c_2)$.

Given a constraint $c \in \mathcal{C}$ and a variable $v \in V$, the *projection* of c over $V - \{v\}$, written $c \Downarrow_{(V-\{v\})}$ is the constraint c' s.t. $c'\eta = \sum_{d \in D} c\eta[v := d]$. Informally, projecting means eliminating some variables from the support. This is done by associating with each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. In short, combination is performed via the multiplicative operation of the semiring, and projection via the additive one.

Solutions. A *solution* of an SCSP $P = \langle C, con \rangle$ is the constraint $Sol(P) = (\bigotimes C) \Downarrow_{con}$. That is, we combine all constraints, and then project over the variables in *con*. In this way we get the constraint with support (not greater than) *con* which is "induced" by the entire SCSP. Note that when all the variables are of interest we do not need to perform any projection.

For example, the solution of the fuzzy CSP of Fig. 1 associates a semiring element to every domain value of variable x. Such an element is obtained by first combining all the constraints together. For instance, for the tuple $\langle a, a \rangle$ (that is, $x = y = a$), we have to compute the minimum between 0.9 (which is the value assigned to $x = a$ in constraint c_1), 0.8 (which is the value assigned to $\langle x = a, y = a \rangle$ in c_2) and 0.9 (which is the value for $y = a$ in c_3). Hence, the resulting value for this tuple is 0.8. We can do the same work for tuple $\langle a, b \rangle \rightarrow 0.2$, $\langle a, c \rangle \rightarrow 0.2$, $\langle b, a \rangle \rightarrow 0$, $\langle b, b \rangle \rightarrow 0$, $\langle b, c \rangle \rightarrow 0.1$, $\langle c, a \rangle \rightarrow 0.8$, $\langle c, b \rangle \rightarrow 0.2$ and $\langle c, c \rangle \rightarrow 0.2$. The obtained tuples are then projected over variable x, obtaining the solution $\langle a \rangle \rightarrow 0.8$, $\langle b \rangle \rightarrow 0.1$ and $\langle c \rangle \rightarrow 0.8$.

2.2 Interchangeability

Interchangeability in constraint networks was first proposed by Freuder [12] to capture equivalence among values of a variable in a discrete constraint satisfaction problem. Value $v = a$ is *substitutable* for $v = b$ if for any solution where

$v = a$, there is an identical solution except that $v = b$. Values $v = a$ and $v = b$ are *interchangeable* if they are substitutable both ways.

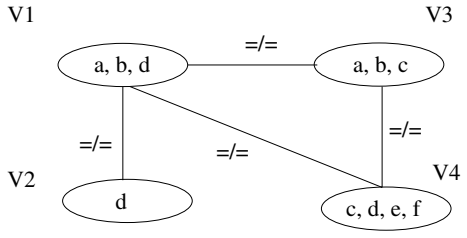


Fig. 2. An example of CSP with interchangeable values.

Interchangeability offers three important ways for practical applications:

- by pruning the interchangeable values, which are redundant in a sense, the problem space can be simplified.
- interchangeability can be used as a solution updating tool; this can by used for user-interaction, can help users in taking decisions by offering alternatives, planning, scheduling ...
- can structure and classify the solution space.

Full Interchangeability considers all constraints in the problem and checks if a value a and b for a certain variable v can be interchanged without affecting the global solution. In the CSP in Fig. 2 (taken from [9]), d, e and f are fully interchangeable for v_4. This is because we inevitably have $v_2 = d$, which implies that v_1 cannot be assigned d in any consistent global solution. Consequently, the values d, e and f can be freely permuted for v_4 in any global solution.

There is no efficient algorithm for computing full Interchangeability, as it may require computing all solutions. The localized notion of *Neighbourhood Interchangeability* considers only the constraints involving a certain variable v. In this notion, a and b are *neighbourhood interchangeable* if for every constraint involving v, for every tuple that admits $v = a$ there is an otherwise identical tuple that admits $v = b$, and vice-versa. In Fig. 2, e and f are neighbourhood interchangeable for v_4.

Freuder showed that neighbourhood interchangeability always implies full interchangeability and can therefore be used as an approximation. He also provided an efficient algorithm (Algorithm 1) for computing neighborhood interchangeability [12], and investigated its use for preprocessing CSP before searching for solutions [1]. Every node in the discrimination tree (Fig. 1) corresponds to a set of assignments to variables in the neighbourhood of v that are compatible with some value of v itself. Interchangeable values are found by the fact that they follows the same path and fall into the same ending node. Fig. 3 shows an example of execution of Algorithm 1 for variable v_4. Domain values e and f are shown to be interchangeable.

Create the root of the discrimination tree for variable v_i;
Let $D_{v_i} = \{$the set of domain values d_{v_i} for variable $v_i\}$;
Let $Neigh(\{v_i\}) = \{$all neighborhood variables v_j of variable $v_i\}$;
for all $d_{v_i} \in D_{v_i}$ **do**
  for all $v_j \in Neigh(\{v_i\})$ **do**
    for all $d_{v_j} \in D_{v_j}$ s.t. d_{v_j} is consistent with d_{v_i} for v_i **do**
      if there exists a child node corresponding to $v_j = d_{v_j}$ **then**
        move to it,
      else
        construct such a node and move to it;
  Add $v_i, \{d_{v_i}\}$ to annotation of the node;
  Go back to the root of the discrimination tree.

Algorithm 1: Discrimination Tree for variable v_i.

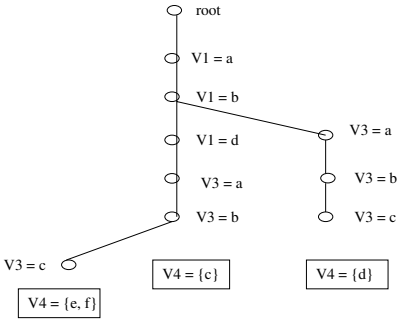


Fig. 3. An example of CSP with computation of neighborhood interchangeable values.

3 Interchangeability in Soft CSPs

In soft CSPs, there is not any crisp notion of consistency. In fact, each tuple is a possible solution, but with different level of preference. Therefore, in this framework, the notion of interchangeability becomes finer: to say that values a and b are interchangeable we have also to consider the assigned semiring level.

More precisely, if a domain element a assigned to variable v can be substituted in each tuple solution with a domain element b without obtaining a worse semiring level we say that b is full substitutable for a.

Definition 1 (Full Substitutability (FS)). *Consider two domain values b and a for a variable v, and the set of constraints C; we say that b is Full Substitutable for a on v ($b \in FS_v(a)$) if and only if*

$$\bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$$

When we restrict this notion only to the set of constraints C_v that involves variable v we obtain a local version of substitutability.

Definition 2 (Neighborhood Substitutability (NS)). *Consider two domain values b and a for a variable v, and the set of constraints C_v involving*

v; we say that b is neighborhood substitutable for a on v ($b \in NS_v(a)$) if and only if

$$\bigotimes C_v\eta[v := a] \leq_S \bigotimes C_v\eta[v := b]$$

When the relations hold in both directions, we have the notion of Full/Neighborhood interchangeability of b with a.

Definition 3 (Full and Neighborhood Interchangeability (*FI* and *NI*)). *Consider two domain values b and a, for a variable v, the set of all constraints C and the set of constraints C_v involving v. We say that b is fully interchangeable with a on v ($FI_v(a/b)$) if and only if $b \in FS_v(a)$ and $a \in FS_v(b)$, that is*

$$\bigotimes C\eta[v := a] = \bigotimes C\eta[v := b].$$

We say that b is Neighborhood interchangeable with a on v ($NI_v(a/b)$) if and only if $b \in NS_v(a)$ and $a \in NS_v(b)$, that is

$$\bigotimes C_v\eta[v := a] = \bigotimes C_v\eta[v := b].$$

This means that when a and b are interchangeable for variable v they can be exchanged without affecting the level of any solution.

Two important results that hold in the crisp case can be proven to be satisfied also with soft CSPs: transitivity and extensivity of interchangeability/substituability.

Theorem 1 (Extensivity: $NS \implies FS$ and $NI \implies FI$). *Consider two domain values b and a for a variable v, the set of constraints C and the set of constraints C_v involving v. Then, neighborhood (substituability) interchangeability implies full (substituability) interchangeability.*

Theorem 2 (Transitivity: $b \in NS_v(a), a \in NS_v(c) \implies b \in NS_v(c)$). *Consider three domain values a, b and c, for a variable v. Then,*

$$b \in NS_v(a), a \in NS_v(c) \implies b \in NS_v(c).$$

Similar results hold for FS, NI and FI.

As an example of interchangeability and substitutability consider the fuzzy CSP represented in Fig. 1. The domain value c is neighborhood interchangeable with a on x ($NI_x(a/c)$); in fact, $c_1 \otimes c_2\eta[x := a] = c_1 \otimes c_2\eta[x := c]$ for all η. The domain values c and a are also neighborhood substitutable for b on x ($\{a, c\} \in NS_v(b)$). In fact, for any η we have $c_1 \otimes c_2\eta[x := b] \leq c_1 \otimes c_2\eta[x := c]$ and $c_1 \otimes c_2\eta[x := b] \leq c_1 \otimes c_2\eta[x := a]$.

3.1 Degradations and Thresholds

In soft CSPs, any value assignment is a solution, but may have a very bad preference value. This allows broadening the original interchangeability concept to one that also allows degrading the solution quality when values are exchanged. We call this ${}^{\delta}$interchangeability, where δ is the *degradation* factor.

When searching for solutions to soft CSP, it is possible to gain efficiency by not distinguishing values that could in any case not be part of a solution of sufficient quality. In ${}_{\alpha}$interchangeability, two values are interchangeable if they do not affect the quality of any solution with quality better than α. We call α the *threshold* factor.

Both concepts can be combined, i.e. we can allow both degradation and limit search to solutions better than a certain threshold (${}^{\delta}_{\alpha}$interchangeability). By extending the previous definitions we can define thresholds and degradation version of full/neighbourhood substitutability/interchangeability.

Definition 4 (${}^{\delta}$Full Substitutability (${}^{\delta}FS$)). *Consider two domain values b and a for a variable v, the set of constraints C and a semiring level δ; we say that b is* ${}^{\delta}$full Substitutable *for a on v ($b \in {}^{\delta}FS_v(a)$) if and only if for all assignments η,*

$$\bigotimes C\eta[v := a] \times_S \delta \leq_S \bigotimes C\eta[v := b]$$

Definition 5 (${}_{\alpha}$Full Substitutability (${}_{\alpha}FS$)). *Consider two domain values b and a, for a variable v, the set of constraints C and a semiring level α; we say that b is* ${}_{\alpha}$full substitutable *for a on v ($b \in {}_{\alpha}FS_v(a)$) if and only if for all assignments η,*

$$\bigotimes C\eta[v := a] \geq \alpha \implies \bigotimes C\eta[v := a] \leq_S \bigotimes C\eta[v := b]$$

Similarly all the notion of ${}^{\delta}/_{\alpha}$Neighborhood Substitutability (${}^{\delta}/_{\alpha}NS$) and of ${}^{\delta}/_{\alpha}$Full/Neighborhood Interchangeability (${}^{\delta}/_{\alpha}FI/NI$) can be defined (just considering the relation in both directions and changing C with C_v).

As an example consider Fig. 1. The domain values c and b for variable y are ${}_{0.2}$Neighborhood Interchangeable. In fact, the tuple involving c and b only differ for the tuple $\langle b, c\rangle$ that has value 0.1 and for the tuple $\langle b, b\rangle$ that has value 0. Since we are interested only to solutions greater than 0.2, these tuples are excluded from the match. The meaning of degradation assume different meanings when instantiated to different semirings:

1. fuzzy CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

$$min(min_{c\in C}(c\eta[v := a]), \delta) \leq min_{c\in C}(c\eta[v := b])$$

which means that changing $v := b$ to $v := a$ does not make the solution worse than before or worse than δ. In the practical case where we want to only consider solutions with a quality better than δ, this means that substitution will never put a solution out of this class.

2. weighted CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

$$\sum_{c \in C} c\eta[v := a] + \delta \geq \sum_{c \in C} c\eta[v := b]$$

which means that the penalty for the solution does not increase by more than a factor of δ. This allows for example to express that we would not want to tolerate more than δ in extra cost. Note, by the way, that $\leq_S$ translates to $\geq$ in this version of the soft CSP.

3. probabilistic CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

$$(\prod_{c \in C} c\eta[v := a]) \cdot \delta \leq \prod_{c \in C} c\eta[v := b]$$

which means that the solution with $v = b$ is not degraded by more than a factor of δ from the one with $v = a$.

4. crisp CSP: $b \in {}^{\delta}FS_v(a)$ gets instantiated to:

$$(\bigwedge_{c \in C} c\eta[v := a]) \wedge \delta \Rightarrow (\bigwedge_{c \in C} c\eta[v := b])$$

which means that when $\delta = true$, whenever a solution with $v = a$ satisfies all constraints, so does the same solution with $v = b$. When $\delta = false$, it is trivially satisfied (i.e. δ is too loose a bound to be meaningful).

3.2 Properties of Degradations and Thresholds

As it is very complex to determine full interchangeability/substitutability, we start by showing the fundamental theorem that allows us to approximate ${}^{\delta}/_{\alpha}FS/FI$ by ${}^{\delta}/_{\alpha}NS/NI$:

Theorem 3 (Extensivity). *${}^{\delta}$neighbourhood substitutability implies ${}^{\delta}$full substitutability and ${}_{\alpha}$neighbourhood substitutability implies ${}_{\alpha}$full substitutability.*

This theorem is of fundamental importance since it gives us a way to approximate full interchangeability by neighborhood interchangeability which is much less expensive to compute.

Theorem 4 (Transitivity using thresholds and degradations). *Consider three domain values a, b and c, for a variable v. Then,*

$$b \in {}^{\delta_1}NS_v(a), a \in {}^{\delta_2}NS_v(c) \implies b \in {}^{\delta_1 \times \delta_2}NS_v(c) \text{ and}$$
$$b \in {}_{\alpha_1}NS_v(a), a \in {}_{\alpha_2}NS_v(c) \implies b \in {}_{\alpha_1+\alpha_2}NS_v(c)$$

Similar results holds for FS, NI, FI.

In particular when $\alpha_1 = \alpha_2 = \alpha$ and $\delta_1 = \delta_2 = \delta$ we have:

Corollary 1 (Transitivity and equivalence classes). *Consider three domain values a, b and c, for a variable v. Then,*

– *Threshold interchangeability is a transitive relation, and partitions the set of values for a variable into equivalence classes, that is*

$$b \in {}_{\alpha}NS_v(a), a \in {}_{\alpha}NS_v(c) \implies b \in {}_{\alpha}NS_v(c)$$
$${}_{\alpha}NI_v(b/a), {}_{\alpha}NI_v(a/c) \implies {}_{\alpha}NI_v(b/c)$$

– *If the $\times_S$-operator is idempotent, then degradation interchangeability is a transitive relation, and partitions the set of values for a variable into equivalence classes, that is*

$$b \in {}^{\delta}NS_v(a), a \in {}^{\delta}NS_v(c) \implies b \in {}^{\delta}NS_v(c)$$
$${}^{\delta}NI_v(b/a), {}^{\delta}NI_v(a/c) \implies {}^{\delta}NI_v(b/c)$$

By using degradations and thresholds we have a nice way to decide when two domain values for a variable can be substituable/interchangeable. In fact, by changing the α or δ parameter we can obtain different results.

In particular we can show that an extensivity results for the parameters hold. In fact, it is straightforward to notice that if two values are ${}^{\delta}_{\alpha}$substitutable, they have to be also ${}^{\delta'}_{\alpha'}$substitutable for any $\delta' \leq \delta$ and $\alpha' \geq \alpha$.

Theorem 5 (Extensivity for α and δ). *Consider two domain values a and b, for a variable v, two thresholds α and α' s.t. $\alpha \leq \alpha'$ and two degradations δ and δ' s.t. $\delta \geq \delta'$. Then,*

$$a \in {}^{\delta}NS_v(b) \implies a \in {}^{\delta'}NS_v(b) \text{ and } a \in {}_{\alpha}NS_v(b) \implies a \in {}_{\alpha'}NS_v(b)$$

Similar results holds for FS, NI, FI.

As a corollary when threshold and degradation are $\mathbf{0}$ or $\mathbf{1}$ we have some special results.

Corollary 2. *When $\alpha = \mathbf{0}$ and $\delta = \mathbf{1}$, we obtain the non approximated versions of NS. When $\alpha = \mathbf{1}$ and $\delta = \mathbf{0}$, all domain values are substitutable.*

$$\forall a, b,\ a \in {}_{\mathbf{0}}NS_v(b) \text{ and } a \in {}^{\mathbf{1}}NS_v(b) \iff a \in NS(b)$$
$$\forall a, b,\ a \in {}_{\mathbf{1}}NS_v(b) \text{ and } a \in {}^{\mathbf{0}}NS_v(b)$$

Similar results holds for FS, NI, FI.

3.3 Computing ${}^{\delta}/_{\alpha}$-Substitutability/Interchangeability

The result of Theorem 1 is fundamental since it gives us a way to approximate full substituability/interchangeability by neighbourhood substituability/interchangeability which is much less costly to compute.

The most general algorithm for neighborhood substituability/interchangeability in the soft CSP framework is to check for each pair of values whether

the condition given in the definition holds or not. This algorithm has a time complexity exponential in the size of the neighbourhood and quadratic in the size of the domain (which may not be a problem when neighbourhoods are small).

Better algorithms can be given when the times operator of the semiring is idempotent. In this case, instead of considering the combination of all the constraint C_v involving a certain variable v, we can check the property we need (NS/NI and their relaxed versions ${}^{\delta}_{\alpha}NS/NI$) on each constraint itself.

Theorem 6. *Consider two domain values b and a, for a variable v, and the set of constraints C_v involving v. Then we have:*

$$\forall c \in C_v . c\eta[v := a] \leq_S c\eta[v := b] \implies b \in NS_v(a)$$
$$(\forall c \in C_v . c\eta[v := a] \geq \alpha \implies c\eta[v := a] \leq_S c\eta[v := b]) \implies b \in {}_{\alpha}NS_v(a)$$

If the times operator of the semiring is idempotent we also have:

$$\forall c \in C_v . c\eta[v := a] \times_S \delta \leq_S c\eta[v := b] \implies b \in {}^{\delta}NS_v(a)$$

By using Theorem 6 (and Corollary 1 for ${}^{\delta}/_{\alpha}NS$) we can find substitutable/interchangeable domain values more efficiently. Algorithm 2 shows an algorithm that can be used to find domain values that are Neighborhood Interchangeable. It uses a data structure similar to the *discrimination trees*, first introduced by Freuder in [12] . Algorithm 2 can compute different versions of

1: Create the root of the discrimination tree for variable v_i
2: Let $C_{v_i} = \{c \in C \mid v_i \in supp(c)\}$
3: Let $D_{v_i} = \{$the set of domain values d_{v_i} for variable $v_i\}$
4: **for all** $d_{v_i} \in D_{v_i}$ **do**
5: **for all** $c \in C_v$ **do**
6: execute Algorithm NI-Nodes(c, v, d_{v_i}) to build the nodes associated with c
7: Add $v_i, \{d_{v_i}\}$ to annotation of the last build node,
8: Go back to the root of the discrimination tree.

Algorithm 2:Algorithm to compute neighbourhood interchangeable sets for variable v_i.

neighbourhood interchangeability depending on the algorithm $NI-nodes$ used. Algorithm 3 shows the simplest version without threshold or degradation. The

1: **for all** assignments η_c to variables in $supp(c)$ **do**
2: compute the semiring level $\beta = c\eta_c[v_i := d_{v_i}]$,
3: **if** there exists a child node corresponding to $\langle c = \eta_c, \beta \rangle$ **then**
4: move to it,
5: **else**
6: construct such a node and move to it.

Algorithm 3: NI-Nodes(c, v, d_{v_i}) for Soft-NI.

algorithm is very similar to that defined by Freuder in [12], and when we consider the semiring for classical CSPs $S_{CSP} = \langle \{false, true\}, \vee, \wedge, false, true \rangle$ and all constraints are binary, it computes the same result. Notice that for each node we add also an information representing the cost of the assignment η_c.

When all constraints are binary, considering all constraints involving variable v is the same as considering all variables connected to v by a constraint, and our algorithm performs steps as that given by Freuder.

We can determine the complexity of the algorithm by considering that the algorithm calls $NI - Nodes$ for each $k - ary$ constraint exactly once for each value of each the k variables; this can be bounded from above by $k * d$ with d the maximum domain size. Thus, given m constraints, we obtain a bound of

$$O(m * k * d * O(AlgorithmNI-nodes)).$$

The complexity of $AlgorithmNI-nodes$ strictly depends on the size of the domain d and from the number of variables k involved in each constraint and is given as

$$O(AlgorithmNI-nodes) = d^{k-1}.$$

For complete constraint graphs of binary constraints ($k = 2$), we obtain the same complexity bound of $O(n^2d^2)$ as Freuder in [12].

1: **for all** assignments η_c to variables in $supp(c)$ s.t. $\beta = c\eta_c[v_i := d_{v_i}]$ and $\alpha \leq_S \beta$ **do**
2: **if** there exists a child node corresponding to $\langle c = \eta_c, \beta \rangle$ **then**
3: move to it,
4: **else**
5: construct such a node and move to it.

Algorithm 4: NI-Nodes(c, v, d_{v_i}) for Soft $_\alpha NI$.

1: **for all** assignments η_c to variables in $supp(c)$ **do**
2: compute the semiring level $\beta = c\eta_c[v_i := d_{v_i}]$,
3: **if** there exists a child node corresponding to $\langle c = \eta_c, \beta', \bar{\beta} \rangle$ with $(\bar{\beta} \leq \beta) \wedge (\beta \times \delta \leq \beta')$ **then**
4: move to it and change the label to $\langle c = \eta_c, glb(\beta', \beta), \bar{\beta} + (\beta \times \delta) \rangle$,
5: **else**
6: construct the node $\langle c = \eta_c, \beta, \beta \times \delta \rangle$ and move to it.

Algorithm 5: NI-Nodes(c, v, d_{v_i}) for Soft $^\delta NI$.

Algorithms for the relaxed versions of NI are obtained by substituting different versions of Algorithm 3. For $_\alpha NI$, the algorithm needs to only consider tuples whose semiring value is greater than α, as shown in Algorithm 4. For

$^{\delta}NI$, the algorithm needs to only consider tuples that can cause a degradation by more than δ, as shown in Algorithm 5. The idea here is to save in each node the information needed to check at each step $^{\delta}NS$ in both directions. In a semiring with total order, the information represent the "interval of degradation". As both algorithms consider the same assignments as Algorithm 3, their complexity remains unchanged at $O(d^{k-1})$.

4 An Example

Fig. 4 shows the graph representation of a CSP which might represent a car configuration problem. A product catalog might represent the available choices

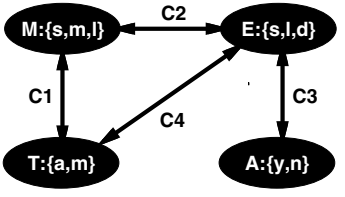


Fig. 4. Example of a CSP modeling car configuration. It has 4 variables: M = model, T = transmission, A = Air Conditioning, E = Engine.

through a soft CSP. With different choices of semiring, the CSP of Fig. 4 can represent different problem formulations; a possible example follows:

Example 1 An optimization criterion might be the time it takes to build the car. Delay is determined by the time it takes to obtain the components and to reserve the resources for the assembly process. For the delivery time of the car, only the longest delay would matter. This could be modelled by the semiring $< \Re^+, min, max, +\infty, 0 >$[1], with the binary constraints:

$$C_1 = \begin{array}{cc|ccc} & & \multicolumn{3}{c}{M} \\ & & s & m & l \\ \hline T & a & \infty & 3 & 4 \\ & m & 2 & 4 & \infty \end{array} \quad C_2 = \begin{array}{cc|ccc} & & \multicolumn{3}{c}{M} \\ & & s & m & l \\ \hline & s & 2 & 3 & \infty \\ E & l & 30 & 3 & 3 \\ & d & 2 & 3 & \infty \end{array} \quad C_3 = \begin{array}{cc|ccc} & & \multicolumn{3}{c}{E} \\ & & s & l & d \\ \hline A & y & 5 & 4 & 7 \\ & n & 0 & 30 & 0 \end{array} \quad C_4 = \begin{array}{cc|ccc} & & \multicolumn{3}{c}{E} \\ & & s & l & d \\ \hline T & a & \infty & 3 & \infty \\ & m & 4 & 10 & 3 \end{array}$$

and unary constraints C_M, C_E, C_T and C_A that model the time to obtain the components:

$$C_M = \frac{s\ m\ l}{2\ 3\ 3} \quad C_E = \frac{s\ l\ d}{3\ 2\ 3} \quad C_T = \frac{a\ m}{1\ 2} \quad C_A = \frac{y\ n}{3\ 0}$$

△

[1] This semiring and the fuzzy one are similar, but the first uses an opposite order. Let us call this semiring *opposite-fuzzy*.

Let us now consider the variable E of Example 1 and compute ${}^{\delta}/_{\alpha}NS/NI$ between its values by using Definition 4 and Definition 5. In Fig. 5 directed arcs are added when the source can be ${}^{\delta}/_{\alpha}$substituted to the destination node. It is easy to see how the occurrences of ${}^{\delta}/_{\alpha}NS$ change, depending on δ and α degrees.

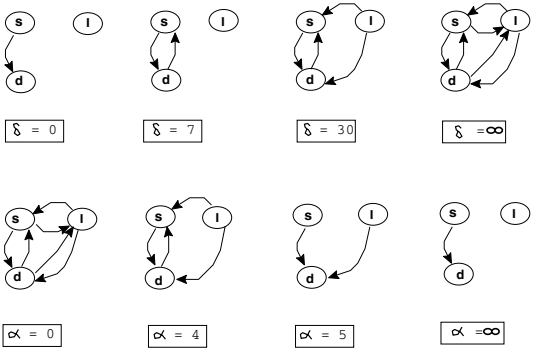


Fig. 5. Example of how δ-substitutability and α-substitutability varies in the opposite-fuzzy CSP over the values of variable E.

We can notice that when δ takes value 0 (the **1** of the optimization semiring), small degradation is allowed in the CSP tuples when the values are substituted; thus only value s can be substituted for value d. As δ increases in value (or decreases from the semiring point of view) higher degradation of the solutions is allowed and thus the number of substitutabilities increase with it.

In the second part of Fig. 5 we can see that for $\alpha = 0$ all the values are interchangeable (in fact, since there are no solutions better than $\alpha = 0$, by definition all the elements are ${}_{\alpha}$interchangeable).

For a certain threshold ($\alpha = 4$) values s and d are ${}_{\alpha}$interchangeable and value l can substitute values s and d. Moreover, when α is greater than 5 we only have that s can substitute d.

We will show now how to compute interchangeabilities by using the Discrimination Tree algorithm. In Fig. 6 the Discrimination Tree is described for variable M when $\alpha = 2$ and $\alpha = 3$. We can see that values m and l for variable M are ${}_2$interchangeable whilst there are no interchangeabilities for $\alpha = 3$.

5 Conclusions

Interchangeability in CSPs has found many applications for problem abstraction and solution adaptation. In this paper, we have shown how the concept can be extended to soft CSPs in a way that maintains the attractive properties already known for hard constraints.

The two parameters α and δ allow us to express a wide range of practical situations. The threshold α is used to eliminate distinctions that would not

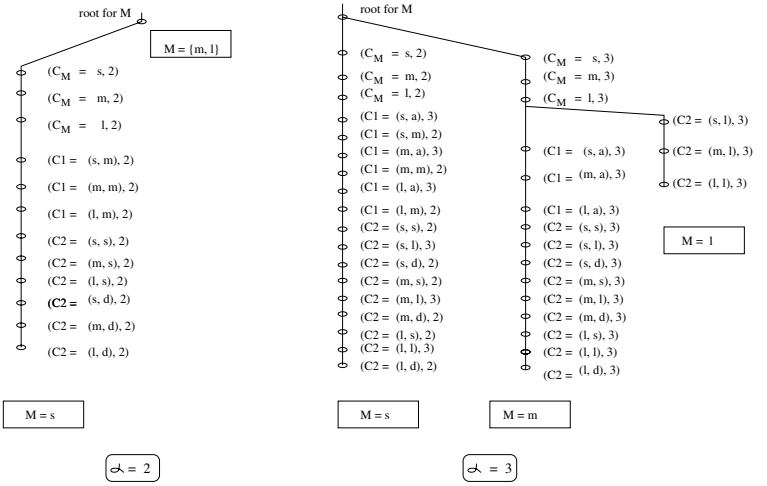


Fig. 6. Example of a search of α-interchangeability computing by the use of discrimination trees.

interest us anyway, while the allowed degradation δ specifies how precisely we want to optimize our solution. We are now conducting a detailed investigation on how variation of these parameters affects interchangeability on random problems.

References

[1] Benson, B., Freuder, E.: Interchangeability preprocessing can improve forward checking search. In: Proc. of the 10th ECAI, Vienna, Austria. (1992) 28–30

[2] Bistarelli, S.: Soft Constraint Solving and programming: a general framework. PhD thesis, Dipartimento di Informatica, Università di Pisa, Italy (2001)

[3] Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G.: Semiring-based CSPs and Valued CSPs: Frameworks, properties, and comparison. CONSTRAINTS: An international journal. Kluwer **4** (1999)

[4] Bistarelli, S., Montanari, U., Rossi, F.: Constraint Solving over Semirings. In: Proc. IJCAI95, San Francisco, CA, USA, Morgan Kaufman (1995)

[5] Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based Constraint Solving and Optimization. Journal of the ACM **44** (1997) 201–236

[6] Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based Constraint Logic Programming: Syntax and Semantics. ACM Transactions on Programming Languages and System (TOPLAS) **23** (2001) 1–29

[7] Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. In: Proc. ESOP, 2002, Grenoble, France. LNCS, Springer-Verlag (2002)

[8] Choueiry, B.Y.: Abstraction Methods for Resource Allocation. PhD thesis, EPFL PhD Thesis no 1292 (1994)

[9] Choueiry, B.Y., Noubir, G.: On the computation of local interchangeability in discrete constraint satisfaction problems. In: Proc. of AAAI-98. (1998) 326–333

[10] Dubois, D., Fargier, H., Prade, H.: The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In: Proc. IEEE International Conference on Fuzzy Systems, IEEE (1993) 1131–1136
[11] Fargier, H., Lang, J.: Uncertainty in constraint satisfaction problems: a probabilistic approach. In: Proc. European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU). Volume 747 of LNCS., Springer-Verlag (1993) 97–104
[12] Freuder, E.C.: Eliminating interchangeable values in constraint satisfaction problems. In: Proc. of AAAI-91, Anaheim, CA (1991) 227–233
[13] Freuder, E., Wallace, R.: Partial constraint satisfaction. AI Journal **58** (1992)
[14] Haselbock, A.: Exploiting interchangeabilities in constraint satisfaction problems. In: Proc. of the 13th IJCAI. (1993) 282–287
[15] Neagu, N., Faltings, B.: Exploiting interchangeabilities for case adaptation. In: In Proc. of the 4th ICCBR01. (2001)
[16] Ruttkay, Z.: Fuzzy constraint satisfaction. In: Proc. 3rd IEEE International Conference on Fuzzy Systems. (1994) 1263–1268
[17] Schiex, T.: Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In: Proc. 8th Conf. of Uncertainty in AI. (1992) 269–275
[18] Schiex, T., Fargier, H., Verfaille, G.: Valued Constraint Satisfaction Problems: Hard and Easy Problems. In: Proc. IJCAI95, San Francisco, CA, USA, Morgan Kaufmann (1995) 631–637
[19] Weigel, R., Faltings, B.: Interchangeability for case adaptation in configuration problems. In: Proc. of the AAAI98 Spring Symposium on Multimodal Reasoning, Stanford, CA. (1998) TR SS-98-04.
[20] Weigel, R., Faltings, B.: Compiling constraint satisfaction problems. Artificial Intelligence **115** (1999) 257–289

Towards Automated Reasoning on the Properties of Numerical Constraints

Lucas Bordeaux, Eric Monfroy, and Frédéric Benhamou

Institut de Recherche en Informatique de Nantes (IRIN), France
{bordeaux,monfroy,benhamou@irin.univ-nantes.fr}

Abstract. A general approach to improving constraint solving is to take advantage of information on the structure and particularities of the considered constraints. Specific properties can determine the use of customized solvers, or they can be used to improve solver cooperation and propagation strategies. We propose a framework in which properties are seen as *abstractions* of the underlying constraints, and relate them to the literature on abstract reasoning. We mainly exemplify this framework on numerical constraints, where such properties as monotonicity and convexity are important. In particular, we show how *deductions* can be made on such constraints to dynamically infer properties. We overview connections with recent works, and we give guidelines and examples on how this kind of tool can be integrated into existing or customized constraint-solvers.

1 Introduction

It is well-known that the tractability of constraint satisfaction strongly depends on the *properties* enjoyed by the considered constraints. For instance, *linear* real-valued (in)equalities (of the form $\sum_i a_i x_i \leq c$) can be handled efficiently using linear programming techniques. Linear problems stand at the intersection between two important classes of problems, namely *monotonic* and *convex* problems. For both of these classes, specialized algorithms can be found in the non-linear programming [9,13] and constraint programming literature [15,16]. Moreover, there is no reason to restrict the list of useful properties to the aforementioned, and an intelligent constraint solver should take into account any useful information on the structure and the special features of the constraints. Information on properties can be useful in solver cooperation, where several solvers can be adapted to different parts of the problem. Properties of narrowing operators can be used to enhance propagation strategies, and one can even attempt to improve general-purpose algorithms in the case of constraints satisfying a given property (see the example in Section 5). However, and despite the long-standing interest in solving particular classes of functions, the issue of devising logical tools to represent properties and perform reasoning on these properties has been discussed only very recently, starting with a preliminary version of this paper [2,10,14].

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 47–61, 2003.

In this paper, we argue that properties are best understood as *abstractions* of the functions or constraints they represent, hence we briefly shed light to the connection with the *abstract interpretation* framework. We then suggest how, using a simple deduction framework, property information can be inferred in a syntax-directed manner; this deduction framework is exemplified on numerical constraints built over a basic set of operators like $+$, exp, etc. Last, we discuss the notion of *property-aware* algorithms, and suggest how property information can help in such fundamental computational problems as optimization and constraint solving.

1.1 A Simple Example

To motivate our advocacy that curve properties matter, consider the toy problem of finding the minimum value for the curve x/y (Figure 1) when x and y range over $[1, 100]$. It turns out that this curve is monotonic in the following sense: on any point we consider in $[1, 100]^2$, increasing x always yields higher values for the result, while increasing y yields lower values. This information prevents us from using costly techniques for this problem, since the minimum is obtained when x takes its lowest value (namely 1) and y takes its highest value (100).

Determining that function x/y be monotonic on $[1, 100]^2$ was easy since division is a basic operation. It becomes more challenging to determine properties of complex functions constructed by *composition* of basic operations. However, this is not always a difficult task. For instance, raising a function to the exponential does not change its monotonicity, and the minimum for function $\exp(x/y)$ is still obtained for $x = 1,\ y = 100$. On the contrary, the minimum of function $-\exp(x/y)$ is reached on $x = 100,\ y = 1$.

This kind of reasoning is made formal in Section 4, where a proof-theoretic presentation of an inference algorithm is discussed. Here is a formalization of the kind of deductions we obtain using this framework:

$$\dfrac{\dfrac{\dfrac{}{[1,100]^2 \ \vdash_x \ x/y : \oslash} \quad \dfrac{}{[\frac{1}{100}, 100] \ \rhd \ \exp : \oslash}}{[1,100]^2 \ \vdash_x \ \exp(x/y) : \oslash} \quad \dfrac{}{[0, 10^{100}] \ \rhd - : \oslash\!\!\!\!\searrow}}{[1,100]^2 \ \vdash_x \ -\exp(x/y) : \oslash\!\!\!\!\searrow}$$

Each horizontal line in this proof separates a conclusion (below) from its assumptions. The conclusion should be read as "on the intervals $x \in [1, 100]$ and $y \in [1, 100]$, function $-\exp(x/y)$ is *decreasing* on x". Symbols $\oslash\!\!\!\!\searrow$ and $\oslash$ are abstractions which stand, respectively, for decreasing and increasing monotonicity properties. Other technicalities shall be discussed later in the paper.

This simple example shows two noticeable features of our approach. First, property inference is dynamic: it takes into account the current range of the variables. This reflects the fact that, for instance, x/y is monotonic on x on $[1, 100]^2$, but not on $[-10, 10]^2$. Second, computing with abstractions ($\oslash$, etc.) instead of actual constraints can be regarded as a qualitative reasoning on the *shape* of the constraint. In many cases, if we work on the region $[1, 100]^2$, viewing the constraint as a plane whose lower and upper corners are respectively $(1, 1)$ and $(100, 100)$ might be a convenient abstraction level.

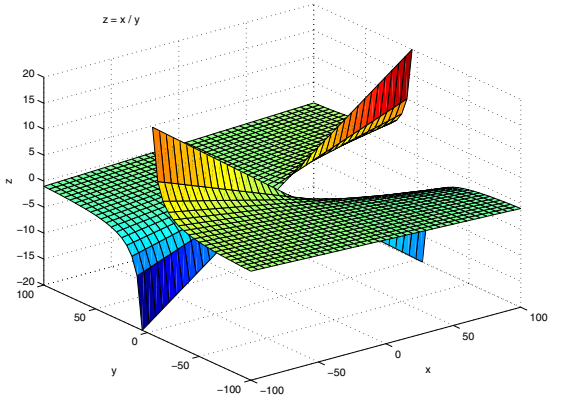


Fig. 1. The curve for division shows 4 monotonic parts: on $\mathbb{R}^- \times \mathbb{R}^-$, on $\mathbb{R}^- \times \mathbb{R}^+$, on $\mathbb{R}^+ \times \mathbb{R}^-$, and on $\mathbb{R}^+ \times \mathbb{R}^+$ (matlab ©).

1.2 Outline of the Paper

Next Section (2) introduces basic material and notations. We then list some properties of interest and introduce abstractions to reify these properties (Section 3). These abstractions allow some kind of automated reasoning, which is formalized in Section 4. Section 5 discusses the practical relevance of this reasoning, and the paper ends up with a conclusion (Section 6).

2 Basic Concepts and Notations

Let $\mathbb{D}$ be an ordered *domain* of computation (reals, integers, ...), and $\mathbb{D}^+$ (*resp.* $\mathbb{D}^-$) be the set of its positive (*resp.* negative) values. Let $\mathcal{V} = \{x_1, \ldots, x_n\}$ be a set of *variable* names.

2.1 Functions and Tuples

A *tuple* t maps each variable x of $\mathcal{V}$ to a value t_x ranging over $\mathbb{D}$ (it is hence a closed substitution). A *(total-) function* f maps each tuple t to a value in $\mathbb{D}$, noted $f(t)$.

Laws of addition and product by an element of $\mathbb{D}$ are piecewise-defined on tuples, using the ring operations on $\mathbb{D}$. We associate to each variable x a *unit* tuple $\vec{x}$ with value 1 on x and 0 elsewhere. Every tuple t may be written in the form $\sum t_x . \vec{x}$. and the tuple $t + \lambda \vec{x}$ is obtained by increasing the value of t on x by λ.

2.2 Interval Evaluation

In some places we shall need to compute the *range* of a function f. Computing the exact range is a difficult task in general, but *overestimates* shall be suitable

for our needs. Safe overestimates can be obtained through Interval Arithmetics [12,9], which associates an interval counterpart to each arithmetic operation (*e.g.,* $[a,b]-[c,d]=[a-d,b-c]$). Given a function f and a subset S of its possible inputs, $f(S)$ shall denote the (approximation of) the set of corresponding outputs.

3 Abstractions for Constraint Properties

We define a framework where properties are seen as constraint *abstractions*. The Abstract Interpretation framework [3] helps defining a sound representation with a clear semantics. This framework is in some sense a dynamic generalization of the static inference of *signs* in the context of program-analysis [3,11].

We select a set of properties which are of special relevance to numerical problem-solving, namely *monotonicity*, *convexity*, and *injectivity*. We define representations for these properties.

3.1 Properties Considered

Next figure (2) illustrates these definitions in the univariate case.

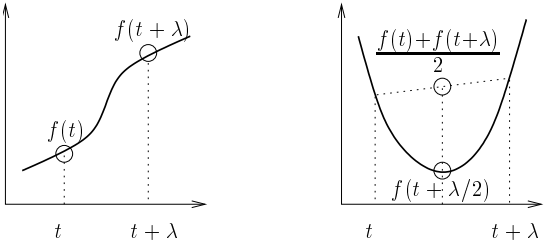


Fig. 2. An *increasing* function (left) - which is also *injective* - and an *upper-convex* one (right).

Monotonicity — Monotonicity means that increasing the value of a tuple on some variable x always yields higher (*resp.* lower) values - informally: "when x increases, so does (*resp.* so does not) $f(x)$". More formally:

Definition 1. [Monotonicity] *A function f is said to be:*

- increasing *on x if (for each tuple t)*
  $\forall \lambda > 0,\ f(t+\lambda \vec{x}) \geq f(t)$ *"when x increases, so does $f(x)$"*
- decreasing *on x if (for each tuple t)*
  $\forall \lambda > 0,\ f(t+\lambda \vec{x}) \leq f(t)$ *"when x increases, $f(x)$ decreases"*

– independent *on x if it is both increasing and decreasing*
$$\forall \lambda > 0,\ f(t + \lambda \overrightarrow{x}) = f(t)$$

A function is said to be monotonic *on variable x if it is either increasing or decreasing on x.*

Convexity — Convexity is a second-order property of monotonicity, which states that the *slopes* of the function are monotonic - informally: convex functions have a specific $\cup$ or $\cap$ shape.

Definition 2. [**Convexity**] *A function f is said to be:*

– upper-convex *on x if (for each tuple t)*
$$\forall \lambda > 0,\ f(t + \tfrac{\lambda}{2} \overrightarrow{x}) \geq \tfrac{1}{2}.(f(t) + f(t + \lambda \overrightarrow{x}))$$
– lower-convex *on x if (for each tuple t)*
$$\forall \lambda > 0,\ f(t + \tfrac{\lambda}{2} \overrightarrow{x}) \leq \tfrac{1}{2}.(f(t) + f(t + \lambda \overrightarrow{x}))$$
– affine *on x if it is both lower- and upper- convex*
$$\forall \lambda > 0,\ f(t + \tfrac{\lambda}{2} \overrightarrow{x}) = \tfrac{1}{2}(f(t) + f(t + \lambda \overrightarrow{x}))$$

A function is said to be convex *on variable x if it is either lower- or upper-convex on x.*

Injectivity — The ability to express *strict* inequalities is recovered via the *injectivity* property, stating that no two points have the same image, and which we define as a separate property for technical convenience:

Definition 3. [**Injectivity**] *A function f is said to be:*

– *injective on x if (for each tuple t)*
$$\forall \lambda > 0,\ f(t + \lambda \overrightarrow{x}) \neq f(t)$$

Note that all these definitions are n-dimensional generalizations of the univariate definitions illustrated by figure 2, where the property considered holds for a given *axis*. In particular, our definition of the *convexity* property is weaker than the one used in non-linear optimization. It corresponds to several notions (*row-*, *interval-*, and *axis-* convexity [5,1,15]) investigated in the literature on constraints.

3.2 Representing Properties

Properties represent *classes* of functions. For instance, several classes are defined with respect to monotonicity on a given variable: increasing and decreasing functions, functions which have none of these properties and functions for which both properties hold (*i.e., independent* functions).

To reason on properties, classes of functions are represented by *abstract symbols*. Operations on values can be extended to these symbols while preserving well-defined semantic properties. Following [3], the meaning of each abstract symbol we introduce is defined as the *concrete* class of functions it represents.

Definition 4. [Abstract Symbol] *We define the following sets of abstract symbols (the associated semantics is stated informally):*

Property	*Abs. Symb.*	*Meaning*
Monotonicity	$\oslash$	*increasing*
	$\oslash$	*decreasing*
	$\ominus$	*independent*
Convexity	$\cup$	*upper-convex*
	$\cap$	*lower-convex*
	$\otimes$	*affine*
Injectivity	$\not\oplus$	*injective*

Each set is completed with the $\top$ *and* $\bot$ *symbols, which respectively represent the class of all functions (with no special property) and the empty class of functions. Symbols for each property are partially ordered as depicted in Figure 3.*

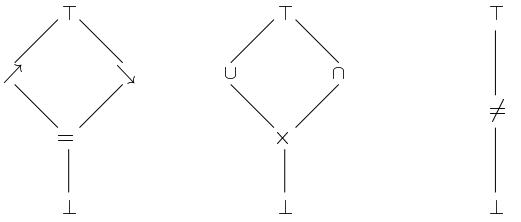


Fig. 3. Abstract lattices for the properties of monotonicity (left), convexity (middle), and injectivity (right).

We use the circled notation throughout the text to avoid confusion between the property symbols and the other signs used in the paper. Note that the ordering on symbols mimics the inclusion order on the classes they represent - for instance the class of affine functions is contained in the class of upper-convex ones.

This ordering allows us to define the usual lattice operations: the least upper bound or *lub* ($a \vee b = \min\{c \mid c \geq a \text{ and } c \geq b\}$) and the greatest lower bound or *glb* ($a \wedge b = \max\{c \mid c \leq a \text{ and } c \leq b\}$). For instance, on the lattice of monotonicity properties, $\oslash \vee \oslash$ gives $\top$ and $\oslash \wedge \ominus$ gives $\ominus$. Since our lattices are symmetric, we denote $-\alpha$ the symmetric of property α; for instance $-\oslash$ gives $\oslash$, and $-\ominus$ gives $\ominus$.

3.3 Combining Symbols

In Abstract Interpretation, one can use the cartesian product of several lattices to obtain a finer set of properties [4]. Figure 4 gives a sample (in the univariate case) of the wide range of function properties expressed in our framework using a *triplet* of symbols, standing for monotonicity, convexity, and injectivity respectively.

Definition 5. *A* property *is a triplet ABC, where the letters stand (in this order) for a monotonicity, a convexity, and an injectivity symbol.*

We allow the omission of the non-informative symbols $\top$ from the triplet. For instance, symbol $\varnothing$ is seen as a valid property, since it is understood as $\varnothing\top\top$. The triple-top property $\top\top\top$ shall be noted $\top$. Note that a partial ordering is naturally defined on properties as $ABC \leq A'B'C'$ iff $A \leq A'$ and $B \leq B'$ and $C \leq C'$. for instance it should be clear that $\ominus\otimes \leq \varnothing\otimes \leq \varnothing \leq \top$.

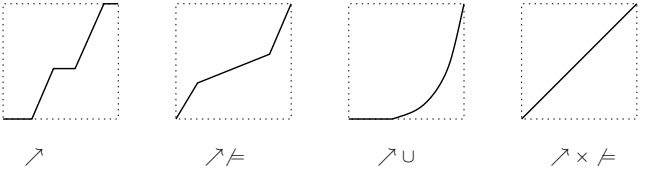


Fig. 4. Several kinds of increasing functions.

Of course, all our definitions have been stated with respect to a given variable, and it is possible to use the information obtained for each variable to get an n-dimensional qualitative representation of the *shape* of the curve. For instance, the curve obtained on Fig. 5, left-hand-side is typical of something $\ominus\otimes\top$ on x and $\oslash\cup\not\in$ on$y$[1].

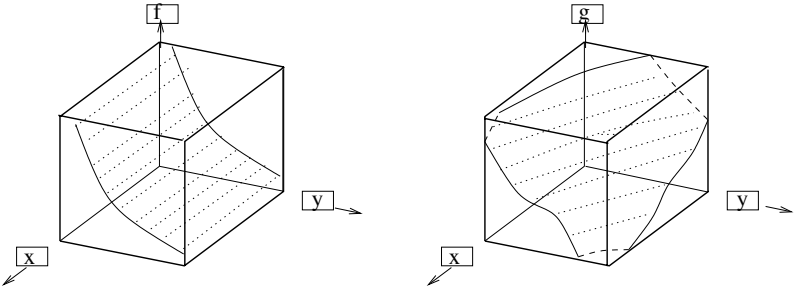


Fig. 5. Combining the information along each axis.

[1] Obviously, the most precise information we could define on a function would be a vector of properties of the form $\langle\ominus\otimes\top, \oslash\cup\not\in\rangle$. Yet, to avoid heavy notation, we shall always reason on *one variable at once*. The reasoning between several variables is independent, and it is straightforward to adapt our deduction rules to higher dimensions.

4 Reasoning on Properties

Now that properties are reified as symbols on which anything we need is defined (in our case, a partial ordering), we can manipulate these objects and make deductions. Quite naturally, the aim of these deductions shall be to infer properties which *provably* hold for some functions. There exists several ways to ensure *correctness* of the deductions. We shall adopt the following definition (which is strongly related to the correctness of constraint-solvers defined, for instance, in [1]):

Definition 6. [Correctness] *Let α be an* abstraction *function and γ be the* concretization *function, which maps each symbol to the set of functions it represents. The abstraction is correct if α and γ form a* Galois connection *[3] between the concrete and abstract lattices i.e., (F stands for a class of functions, s is a symbol)*

$$\alpha(F) \leq s \quad \Leftrightarrow \quad F \subseteq \gamma(s) \tag{1}$$

4.1 Axioms

The first kind of information we wish to store is related to the basic operators (+, *, ...) of the language. We call *axioms* the following kind of sentence:

Definition 7. *An* axiom *is a statement of the form:*

$$\begin{array}{rlll} & I & \rhd \sim : P & (\sim \text{ is an unary operator}) \\ or & I_1, I_2 & \rhd \diamond : P_1, P_2 & (\diamond \text{ is a binary operator}) \end{array}$$

where the I_is and intervals, $\sim$ and $\diamond$ are (respectively) unary or binary operators, and the P_is are properties. For instance, the binary case should be read as "when its first/second arguments range over the intervals I_1 and I_2, operator $\diamond$ is P_1 on its first argument and $P2$ on its second argument".

It is easy to create a database of axioms for all of the basic operators +, *, etc., since their curves are easily decomposed into a small number of parts where properties can be identified (the overline indicates that the following axioms do not need any assumption to be proved):

$$\overline{\mathbb{D} \rhd \exp : \varnothing \cup \not\ni} \qquad \overline{\mathbb{D}^+ \rhd \log : \varnothing \cap \not\ni}$$

$$\overline{\mathbb{D}^- \rhd x^{2a} : \not\ni \cup \not\ni} \qquad \overline{\mathbb{D}^+ \rhd x^{2a} : \varnothing \cup \not\ni}$$

$$\overline{\mathbb{D}^- \rhd x^{2a+1} : \varnothing \cap \not\ni} \qquad \overline{\mathbb{D}^+ \rhd x^{2a+1} : \varnothing \cup \not\ni}$$

$$\overline{\mathbb{D}^+ \rhd \sqrt[2a]{x} : \varnothing \cap \not\ni}$$

$$\overline{\mathbb{D}^- \rhd \sqrt[2a+1]{x} : \varnothing \cup \not\ni} \qquad \overline{\mathbb{D}^+ \rhd \sqrt[2a+1]{x} : \varnothing \cap \not\ni}$$

Note that these axioms should just be read as, for instance "function exp is increasing (on its only argument) over $\mathbb{D}$". We can define binary axioms:

$$\frac{}{\mathbb{D}, \mathbb{D} \quad \triangleright \quad + : \varnothing\otimes\not\varnothing, \varnothing\otimes\not\varnothing}$$

$$\frac{}{\mathbb{D}, \mathbb{D} \quad \triangleright \quad - : \varnothing\otimes\not\varnothing, \searrow\otimes\not\varnothing}$$

$$\frac{}{\mathbb{D}^+, \mathbb{D}^+ \quad \triangleright \quad * : \varnothing\otimes\not\varnothing, \varnothing\otimes\not\varnothing} \qquad \frac{}{\mathbb{D}^+, \mathbb{D}^- \quad \triangleright \quad * : \searrow\otimes\not\varnothing, \varnothing\otimes\not\varnothing}$$

$$\frac{}{\mathbb{D}^-, \mathbb{D}^+ \quad \triangleright \quad * : \varnothing\otimes\not\varnothing, \searrow\otimes\not\varnothing} \qquad \frac{}{\mathbb{D}^-, \mathbb{D}^- \quad \triangleright \quad * : \searrow\otimes\not\varnothing, \searrow\otimes\not\varnothing}$$

We just give one example of how to prove these results. The generalization to any other axiom of the list is straightforward:

Rule 1

$$\frac{}{\mathbb{D}^+, \mathbb{D}^+ \quad \triangleright \quad * : \varnothing\otimes\not\varnothing, \varnothing\otimes\not\varnothing}$$

Proof. Consider particular values $x, x' \in \mathbb{D}^+$ and $y \in \mathbb{D}^+$, with $x < x'$. Since the product is positive, we have $x.y \leq x'.y$ ($\varnothing$); furthermore, since both x and y are non-zero, we have $x.y \not= x'.y$ ($\not\varnothing$) Last, we have $(x.y + x'.y)/2 = (x.y)/2 + (x'.y)/2$ ($\otimes$).

4.2 Facts

Whereas axioms state properties of the basic operators ("operator exp is $\varnothing$"), the statements we would like to deduce have a slightly more complex form since they involve complex terms with *variables* ("function $x + log(y)$ is increasing upper-convex on y"). We use the following definition:

Definition 8. *A* fact *is a statement of the form*

$$x_1 \in I_1, \ldots, x_n \in I_n \quad \vdash_x \quad f : P$$

where the x_is are the variables, the I_is and intervals bounding these variables, f is an expression, and P is a property. This statement should be read as "within the box $I_1 \times \ldots \times I_n$, expression f is P on x".

The notion of *correctness* of a fact is obtained from equation 1: $P \leq P' \Leftrightarrow \phi \in \gamma(P')$, where ϕ is the restriction of function f to the space delimited by the constraints of C. An example of (correct) fact is

$$x \in [0, 10], y \in [0, 10] \quad \vdash_x \quad 2x.(y + 1) : \varnothing\otimes\not\varnothing$$

which is read: "function $2x.(y+1)$ is increasing, affine and surjective on variable x within the box $x \in [0, 10], y \in [0, 10]$". We first define very simple rules to deduce facts; since these rules do not have assumptions, we call them *elementary rules*:

Rule 2 *Constants and variables.*

$$\frac{}{(whatever) \quad \vdash_x \quad x : \varnothing \otimes \not\varnothing}$$

$$\frac{}{(whatever) \quad \vdash_x \quad y : \ominus \otimes \top} \quad (if\ y \not= x)$$

$$\frac{}{(whatever) \quad \vdash_x \quad c : \ominus \otimes \top} \quad (for\ each\ constant\ c)$$

The previous rules simply state that "x is increasing on x" and other trivial statements. Next rules states that if something holds over $[-\infty, +\infty]$, it also holds over (say) $[1, 3]$:

Rule 3 *Domain reduction.*

$$\frac{x_1 \in I_1, \ldots, x_n \in I_n \quad \vdash_x \quad f : P}{x_1 \in I'_1, \ldots, x_n \in I'_n \quad \vdash_x \quad f : P} \quad if \quad \forall j \in 1 \ldots n \ \ (I'_j \subseteq I_j)$$

Technically, another kind of rules could be defined, namely *weakening* rules ($\frac{x:M^+}{x:M^-}$ if $M^- \geq M^+$) to ensure that a rule applying to a general type applies to a more particular type, but we shall assume this filtering to be implicit. Elementary rules are formally needed to initialize the deduction process but, to simplify things, we shall sometimes omit mentioning them when writing examples of deductions (applying these rules is usually straightforward).

4.3 Syntax-Directed Deduction Rules

While elementary facts are deduced "from scratch" and are used to bootstrap the deduction machinery. Rules make it possible to deduce more facts from the elementary ones.

The main class of deductions one would like to perform concerns the compositional inference which determines the properties of a complex expression as a function of the properties of its subexpressions. For instance, given the properties of $3x$ and those of $\log y$, which properties hold for $3x + \log y$? For the sake of clarity, we first restrict ourselves to the sole property of *monotonicity* throughout this subsection. We have the following scheme of rules (over the line are the assumptions, below is the conclusion of the rule):

Rule 4 *Unary composition ("If we compose a function increasing on x with an increasing function, we obtain a function increasing on x").*

$$\frac{x_1 \in I_1, \ldots, x_n \in I_n \ \vdash_x \ g : M \qquad g(I_1, \ldots, I_n) \ \rhd \ f : \varnothing}{x_i \in I_i \ \vdash_x \ f(g) : M}$$

$$\frac{x_1 \in I_1, \ldots, x_n \in I_n \ \vdash_x \ g : M \qquad g(I_1, \ldots, I_n) \ \rhd \ f : \not\varnothing}{x_i \in I_i \ \vdash_x \ f(g) : -M}$$

This example illustrates the distinction between so-called *facts* and *axioms*: f is a unary *operator* which is monotonic in its only argument. As a consequence, if we construct an expression $f(g)$ where g is (say) $\oslash$ on x, the resulting function shall be increasing/decreasing on x. Note that f has to be monotonic *all over the image of* g; as mentioned earlier, this image can be estimated using simple interval evaluation. Binary operators are handled very similarly:

Rule 5 *Composition with a binary operator.*

$$\frac{x_i \in I_i \ \vdash_x \ l : M_1 \qquad x_i \in I_i \ \vdash_x \ r : M_2 \qquad l(I), r(I) \ \rhd \ f : M, M'}{x_i \in I_i \ \vdash_x \ f(l,r) : \pm M_1 \vee \pm M_2}$$

where each $\pm$ *depends on the monotonicity of* M *or* M' *(i ranges from* 1 *to* n*;* I *stands for* $I_1, \ldots, I_n$*).*

Note the potential loss of information induced by the $\vee$ operation (see 2nd example in next subsection).

4.4 Examples of Deductions

Example 1. Here is the complete deduction obtained within our formalism showing that $2x.y + x$ is increasing on x:

$$\frac{\left\{\begin{array}{l} \dfrac{\dfrac{}{\mathcal{B} \vdash_x 2 : \ominus}}{\mathcal{B} \vdash_x 2x : \oslash} \ \ldots \\ \\ \dfrac{}{\mathcal{B} \vdash_x y : \ominus} \end{array}\right. \qquad \dfrac{\mathbf{P}}{\begin{array}{c}[-20,20][1,2] \\ \rhd \quad * : \oslash\oplus\end{array}}}{\mathcal{B} \ \vdash_x 2x.y : \oslash} \qquad \frac{}{\mathcal{B} \ \vdash_x x : \oslash} \qquad \frac{}{\begin{array}{c}[-40,40][-10,10] \\ \rhd \quad + : \oslash\oslash\end{array}}$$

$$x \in [-10,10], y \in [1,2] \ \vdash_x \ 2x.y + x : \oslash$$

$\mathcal{B}$ (for "box") stands for $x \in [-10,10], y \in [1,2]$; $\mathbf{P}$ stands for a proof that $2x$ is increasing on x over $[-20,20], [1,2]$ — this is not an axiom in our database.

Example 2. This was a positive example where the abstract deductions compute the exact information we wish. Here is a deduction which does not work so well — this is to illustrate the loss of information mentioned above. Such a loss is unavoidable when performing qualitative reasoning.

$$\frac{\dfrac{}{x \in \mathbb{D} \ \vdash_x \ x : \oslash} \qquad \dfrac{\dfrac{}{x \in \mathbb{D} \ \vdash_x \ x : \oslash}}{x \in \mathbb{D} \ \vdash_x \ -x : \ominus} \qquad \dfrac{}{x \in \mathbb{D} \ \rhd + : \oslash\oslash}}{x \in \mathbb{D} \ \vdash_x \ x + (-x) : \oplus}$$

4.5 Other Kinds of Deductions

Reasoning on monotonicity properties represents the most natural form of reasoning using *facts* and *axioms*. It is almost straightforward to deduce from the rule-based formalism a linear-time (syntax-directed) algorithm which outputs a correct approximation of the monotonicity of an expression on a simple domain. Composition rules can be defined in the same spirit for injectivity and convexity properties:

Rule 6 *Composition rule for injectivity.*

$$\frac{\mathcal{B} \ \vdash_x \ g : \not\oplus \qquad g(\mathcal{B}) \ \rhd f : \not\oplus}{\mathcal{B} \ \vdash_x \ f(g) : \not\oplus}$$

Rule 7 *Composition rule for convexity (same rule for $\cap$ with $\not\ominus$).*

$$\frac{\mathcal{B} \ \vdash_x \ g : \cup \qquad g(\mathcal{B}) \ \rhd f : \varnothing\cup}{\mathcal{B} \ \vdash_x \ f(g) : \cup}$$

Proof. (sketched) For each $t \in \Omega$ and $\lambda > 0$, we note $t' = t + \lambda \vec{x}$ and $avg(t, t') = (t + t')/2$. We have $avg(fgx, fgy) \geq f(avg(gx, gy)) \geq f(g(avg(x, y)))$.

Rules can also be defined to make deductions when some properties hold over two contiguous intervals $[l, m]$ and $[m, r]$ (what should we deduce about $[l, r]$?). This turns out to work for both monotonicity and convexity properties[2]:

Rule 8 *Domain extension for monotonicity (similar rules apply to binary operators and to decreasing monotonicity).*

$$\frac{[l, m] \ \rhd \ f : \varnothing \qquad [m, r] \ \rhd \ f : \varnothing}{[l, r] \ \rhd \ f : \varnothing}$$

Rule 9 *Domain extension for convexity (symmetric rule for $\not\ominus$).*

$$\frac{[l, m] \ \rhd \ f : \not\ominus\cup \qquad [m, r] \ \rhd \ f : \varnothing\cup}{[l, r] \ \rhd \ f : \cup}$$

Proof. (sketched) Upper-convexity means the slope is increasing. Since it is negative on $[l, m]$, positive on $[m, r]$, and increasing on both, it is increasing all over $[l, r]$.

Due to lack of space we don't develop examples in all cases. We simply complete example 1 by giving the proof for **P**.

Example 3. (we omit some weakening rule applications)

$$\frac{[-20, 0][1, 2] \ \rhd \ * : \varnothing\otimes, \not\ominus\otimes \not\oplus \qquad [0, 20][1, 2] \ \rhd \ * : \varnothing\otimes \not\oplus, \varnothing\otimes \not\oplus}{[-20, 20][1, 2] \ \rhd \ * : \varnothing\otimes, \top\otimes}$$

[2] We knowingly state these rules on *axioms*: domain extension rules are only useful to extend the axiom database when the domain is too large for any axiom to apply.

5 Practical Relevance

The rule-based framework we have defined enables us to (incompletely) detect the properties which are satisfied by the constraints. Two drawbacks of the approach are that the considered constraints may not exhibit interesting properties, and that our deduction framework may fail to detect these properties.

Regarding the first issue, an interesting feature of our approach is that property detection is *dynamic*. The tighter the bounds of the variables, the best information can be inferred. Ultimately, almost any function becomes monotonic and convex on any axis if we consider small intervals. Since the current CP approach to numerical constraints is based on interval narrowing and splitting, opportunities to detect properties arise very often.

Incompleteness is a drawback of any qualitative reasoning system. We have seen in Section 4.4 that even in simple cases, information may fail to be detected; note that this problem usually arises in presence of variables with multiple occurrences. This is not a surprise since variable redundancy as well as a bad syntax are known to cause problems, notably in interval analysis.

While detection and inference of properties have been discussed throughout the paper, much work remains to be done on the use of property information within constraint solvers. We now highlight several approaches.

Applying specialized solvers — The issue of determining classes of constraint problems which can be solved efficiently by specialized algorithms goes back to (at least) [6]. For numerical problems, the properties of monotonicity and convexity have naturally been exploited. Convex constraints have been considered in [15,5]; especially relevant to numerical domains is the $(3, 2)$-relational-consistency technique [15], which guarantees backtrack-free search. Improvements of Arc-Consistency for monotonic constraints can be found in *e.g.,* [16]. Both kinds of properties have been widely studied in non-linear optimization, for instance *local* optimization algorithms become *global* when a strong notion of convexity holds [13].

Improving solver cooperation — Making several algorithms cooperate is known to be a key feature for efficient constraint solving [8], and the integration of such customized algorithms raises interesting issues. A more efficient cooperation could be obtained if the solver were able to deduce which properties hold for which sub-problems. We hope that our contribution shall be a step towards these intelligent cooperation architectures.

Making general-purpose algorithms more "property-aware" — Having properties defined as objects makes it easier to define *property-aware* algorithms which take property information into account. As an example of the possible algorithms that can be devised, consider the usual (natural form) interval evaluation algorithm. This is a general-purpose method which is extensively used in constraint solvers. We present a simple way to use monotonicity information:

Example 4. Consider the function $f : (x, y) \rightarrow 2x.y + y$ on $[-10, 10] \times [1, 2]$. Natural form interval evaluation estimates the image of f to $2.[-10, 10].[1, 2] + [1, 2] = [-39, 42]$. The function turns out to be increasing on x over these intervals (this is easily detected in our framework, as shown in Section 4.4). We can hence deduce that its global minimum is obtained for $x = -10$.

When we instantiate x to value -10, f becomes the simpler function $g : (x, y) \rightarrow -20.y + y$. Function g now turns out to be decreasing on y when $x = -10$ and $y \in [1, 2]$, and its minimum is hence $f(-10, 2) = -38$. Using monotonicity information, not only have we improved the lower bound of the interval evaluation by 1, but we have also got a guaranteed *global* minimum.

Recent horizons — Several research papers have appeared recently on related topics. [14] proposes to model the way cooperative solvers transform the properties of the problem they receive as an input. The purpose is to improve cooperation.

Especially close to our approach is a recent paper by D. Lesaint [10] which discusses the inference of so-called *types* for discrete constraints. These types are indeed properties which are easily identified to those we discuss; though developed independently, this paper applies to *discrete domains* some methods very similar to those we have suggested for numerical constraints. Whereas our syntax-directed reasoning relies on the syntax of arithmetic expressions, Lesaint uses an algebra for finite domains (union, complementation, etc.) with very similar methods. Type inference is hence used to configure the arc-consistency algorithm used to solve the problem. More work should be done to compare the two approaches and hopefully provide unifying frameworks.

6 Conclusion

Constraint properties provide meaningful information and impact the practical tractability of a Constraint Satisfaction Problem. Integrating such information in constraint solvers requires a formal representation (or "reification") of the properties. The first contribution of this paper was to propose *abstraction* as a formal basis for modeling properties. Our main reference was Abstract Interpretation, though other works on abstraction [7] could also be considered. The formalism was just briefly suggested, and a more complete formulation of the abstraction framework can be considered for future work.

Our second contribution was to define a rule-based framework for property inference. Rules for the properties of monotonicity, convexity and injectivity have been detailed, providing evidence that a symbolic approach to property inference is possible.

Finally, we have overviewed some of the actual or possible applications of property-inference for numerical constraint-solving and optimization. It is the main aim of our ongoing work to develop these applications.

Acknowledgments. We acknowledge helpful discussions with E. Petrov and S. Brand. Corrections and notation improvements were suggested by anonymous reviewers. Adopting a proof-theoretic presentation inspired from David Lesaint [10] helped us improving the presentation of preliminary versions of this paper.

References

1. F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *J. of Logic Programming*, 32(1):1–24, 1997.
2. L. Bordeaux, E. Monfroy, and F. Benhamou. Raisonnement automatique sur les propriétés de contraintes numériques. In *Actes des 11èmes J. Franc. de Programmation en Logique et par Contraintes (JFPLC)*, Nice, France, 2002. Hermès. in french.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th Annual ACM Symp. on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York, NY.
4. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. of Logic Programming*, 13(2-3):103–179, 1992.
5. Y. Deville, O. Barette, and P. Van Hentenryck. Constraint satisfaction over connected row convex constraints. *Artificial Intelligence*, 109(1-2):243–271, 1999.
6. E. C. Freuder. A sufficient condition for backtrack-free search. *J. of the ACM*, 29(1):24–32, 1982.
7. F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 56(2-3):323–390, 1992.
8. L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-interval cooperation in constraint programming. In *Proc. of the Int. Symp. on Symbolic and Algebraic Computation (ISSAC)*, London, Ontario, 2001. ACM Press.
9. E. R. Hansen. *Global Optimization Using Interval Analysis*. Pure and Applied Mathematics. Marcel Dekker Inc., 1992.
10. D. Lesaint. Inferring constraint types in constraint programming. In *Proc. of the 8th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, LNCS, pages 492–507, Ithaca, NY, 2002. Springer.
11. K. Marriott and P.-J. Stuckey. Approximating interaction between linear arithmetic constraints. In *Proc. of the International Symposium on Logic Programming (ISLP)*, pages 571–585, Ithaca, NY, 1994. MIT Press.
12. R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.
13. S. Nash and A. Sofer. *Linear and Nonlinear Programming*. McGraw Hill, 1996.
14. E. Petrov and E. Monfroy. Automatic analysis of composite solvers. In *Proc. of the 14th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, Washington, 2002. IEEE Press. To appear.
15. D. Sam-Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1(1-2):85–118, 1996.
16. Z. Yuanlin and R. H. C. Yap. Arc consistency on n-ary monotonic and linear constraints. In *Proc. of the 6th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, LNCS, pages 470–483, Singapore, 2000. Springer.

Domain-Heuristics for Arc-Consistency Algorithms

Marc R.C. van Dongen

Cork Constraint Computation Centre
Computer Science Department, University College Cork
Western Road, Cork, Ireland,
dongen@cs.ucc.ie

Abstract. Arc-consistency algorithms are widely used to prune the search-space of Constraint Satisfaction Problems (CSPs). They use *support-checks* (also known as consistency-checks) to find out about the properties of CSPs. They use *arc-heuristics* to select the next constraint and *domain-heuristics* to select the next values for their next support-check. We will investigate the effects of domain-heuristics by studying the average time-complexity of two arc-consistency algorithms which use different domain-heuristics. We will assume that there are only two variables. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic based on the notion of a *double-support check*. We will discuss the consequences of our simplification about the number of variables in the CSP and we will carry out a case-study for the case where the domain-sizes of the variables is two. We will present relatively simple formulae for the *exact* average time-complexity of both algorithms as well as simple bounds. As a and b become large $\mathcal{L}$ will require about $2a + 2b - 2\log_2(a) - 0.665492$ checks on average, where a and b are the domain-sizes and $\log_2(\cdot)$ is the base-2 logarithm. $\mathcal{D}$ requires an average number of support-checks which is below $2\max(a, b) + 2$ if $a + b \geq 14$. Our results demonstrate that $\mathcal{D}$ is the superior algorithm. Finally, we will provide the result that on average $\mathcal{D}$ requires strictly fewer than two checks more than any other algorithm if $a + b \geq 14$.

1 Introduction

Arc-consistency algorithms are widely used to prune the search-space of Constraint Satisfaction Problems (CSPs). Arc-consistency algorithms require *support-checks* (also known as consistency-checks in the constraint literature) to find out about the properties of CSPs. They use *arc-heuristics* and *domain-heuristics* to select their next support-check. Arc-heuristics operate at *arc-level* and select the constraint that will be used for the next check. Domain-heuristics operate at *domain-level*. Given a constraint, they decide which values will be used for the next check. Certain kinds of arc-consistency algorithms use heuristics which are, in essence, a combination of arc-heuristics and domain-heuristics.

We will investigate the effect of domain-heuristics by studying the average time-complexity of two arc-consistency algorithms which use different domain-heuristics. We will assume that there are only two variables. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic based on the notion of a *double-support check*. Experimental evidence already exists to suggest that this *double-support heuristic* is efficient.

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 62–75, 2003.

We will define $\mathcal{L}$ and $\mathcal{D}$ and present a detailed case-study for the case where the size of the domains of the variables is two. We will show that for the case-study $\mathcal{D}$ is superior on average.

We will present relatively simple *exact* formulae for the average time-complexity of both algorithms as well as simple bounds. As a and b become large $\mathcal{L}$ will require about $2a + 2b - 2\log_2(a) - 0.665492$ checks on average, where a and b are the domain-sizes and $\log_2(\cdot)$ is the base-2 logarithm. $\mathcal{D}$ will require an average number of support-checks which is below $2\max(a, b) + 2$ if $a + b \geq 14$. Therefore, $\mathcal{D}$ is the superior algorithm. Finally, we will provide an "optimality" result that on average $\mathcal{D}$ requires strictly fewer than two checks more than any other algorithm if $a + b \geq 14$.

As part of our analysis we will discuss the consequences of our simplifications about the number of variables in the CSP.

The relevance of this work is that the double-support heuristic can be incorporated into any existing arc-consistency algorithm. Our optimality result is informative about the possibilities and limitations of domain-heuristics.

The remainder of this paper is organised as follows. In Section 2 we shall provide basic definitions and review constraint satisfaction. A formal definition of the lexicographical and double-support algorithms will be presented in Section 3. In that section we shall also carry out our case-study. In Section 4 we shall present our average time-complexity results and shall compare these results in Section 5. Our conclusions will be presented in Section 6.

2 Constraint Satisfaction

2.1 Basic Definitions

A *Constraint Satisfaction Problem* (or CSP) is a tuple (X, D, C), where X is a set containing the variables of the CSP, D is a function which maps each of the variables to its domain, and C is a set containing the constraints of the CSP.

Let (X, D, C) be a CSP. For the purpose of this paper we will assume that $X = \{\alpha, \beta\}$, that $\emptyset \subset D(\alpha) \subseteq \{1, \ldots, a\}$, that $\emptyset \subset D(\beta) \subseteq \{1, \ldots, b\}$, and that $C = \{M\}$, where M is a constraint between α and β. We shall discuss the consequences of our simplifications in Section 2.3.

For the purpose of this paper a constraint M between α and β is an a by b zero-one matrix, i.e. a matrix with a rows and b columns whose entries are either zero or one. It should be pointed out that our results do not depend on the use of two-dimensional matrices or arrays at all. The use of a matrix in this paper is only for convenience of notation. A tuple (i, j) in the Cartesian product of the domains of α and β is said to *satisfy* a constraint M if $M_{ij} = 1$. Here M_{ij} is the j-th column of the i-th row of M. A value $i \in D(\alpha)$ is said to be *supported* by $j \in D(\beta)$ if $M_{ij} = 1$. Similarly, $j \in D(\beta)$ is said to be supported by $i \in D(\alpha)$ if $M_{ij} = 1$. M is called *arc-consistent* if for every $i \in D(\alpha)$ there is a $j \in D(\beta)$ which supports i and vice versa.

We will denote the set of all a by b zero-one matrices by $\mathbb{M}^{ab}$. We will call matrices, rows of matrices and columns of matrices *non-zero* if they contain more than zero ones, and call them *zero* otherwise.

The *row-support/column-support* of a matrix is the set containing the indices of its non-zero rows/columns. An *arc-consistency algorithm* removes all the unsupported values from the domains of the variables of a CSP until this is no longer possible. A *support-check* is a test to find the value of an entry of a matrix. We will write $M_{ij}^{?}$ for the support-check to find the value of M_{ij}. An arc-consistency algorithm has to carry out a support-check $M_{ij}^{?}$ to find out about the value of M_{ij}. The time-complexity of arc-consistency algorithms is expressed in the number of support-checks they require to find the supports of their arguments.

If we assume that support-checks are not duplicated then at most ab support-checks are needed by any arc-consistency algorithm for any a by b matrix. For a zero a by b matrix each of these ab checks is required. The worst-case time-complexity is therefore exactly ab for any arc-consistency algorithm. In this paper we shall be concerned with the average time-complexity of arc-consistency algorithms.

If $\mathcal{A}$ is an arc-consistency algorithm and M an a by b matrix, then we will write $\mathrm{checks}_{\mathcal{A}}(M)$ for the number of support-checks required by $\mathcal{A}$ to compute the row and column-support of M.

Let $\mathcal{A}$ be an arc-consistency algorithm. The *average time-complexity* of $\mathcal{A}$ *over* $\mathbb{M}^{ab}$ is the function $\mathrm{avg}_{\mathcal{A}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$, where

$$\mathrm{avg}_{\mathcal{A}}(a,b) = \sum_{M \in \mathbb{M}^{ab}} \mathrm{checks}_{\mathcal{A}}(M)/2^{ab}.$$

Note that it is implicit in our definition of average time-complexity that it is just as likely for a check to succeed as it is for it to fail. $\mathcal{A}$ is called *non-repetitive* if it does not repeat support-checks.

A support-check $M_{ij}^{?}$ is said to *succeed* if $M_{ij} = 1$ and said to *fail* otherwise. If a support-check succeeds it is called *successful* and *unsuccessful* otherwise.

Let a and b be positive integers, let M be an a by b zero-one matrix, and let $\mathcal{A}$ be an arc-consistency algorithm. The *trace* of M with respect to $\mathcal{A}$ is the sequence

$$(\, i_1, j_1, M_{i_1 j_1} \,), (\, i_2, j_2, M_{i_2 j_2} \,), \ldots , (\, i_l, j_l, M_{i_l j_l} \,), \tag{1}$$

where $l = \mathrm{checks}_{\mathcal{A}}(M)$ and $M_{i_k j_k}^{?}$ is the k-th support-check carried out by $\mathcal{A}$, for $1 \leq k \leq l$. The *length* of the trace in Equation (1) is defined as l. Its *k-th member* is defined by $(\, i_k, j_k, M_{i_k j_k} \,)$, for $1 \leq k \leq l$.

An interesting property of traces of non-repetitive algorithms is the one formulated as the following theorem which is easy to prove.

Theorem 1 (Trace Theorem). *Let $\mathcal{A}$ be a non-repetitive arc-consistency algorithm, let M be an a by b zero-one matrix, let t be the trace of M with respect to $\mathcal{A}$, and let l be the length of t. There are 2^{ab-l} members of $\mathbb{M}^{ab}$ whose traces with respect to $\mathcal{A}$ are equal to t.*

The theorem will turn out to be convenient later on because it will allow us to determine the "savings" of traces of non-repetitive arc-consistency algorithms without too much effort.

$M_{ij}^{?}$ is called a *single-support check* if, just before the check was carried out, the row-support status of i was known and the column-support status of j was unknown, or

vice versa. A successful single-support check $M^?_{ij}$ leads to new knowledge about one thing. Either it leads to the knowledge that i is in the row-support of M where this was not known before the check was carried out, or it leads to the knowledge that j is in the column-support of M where this was not known before the check was carried out. $M^?_{ij}$ is called a *double-support check* if, just before the check was carried out, both the row-support status of i and the column-support status of j were unknown. A successful double-support check $M^?_{ij}$ leads to new knowledge about *two* things. It leads to the knowledge that i is in the row-support of M and that j is in the column-support of M where neither of these facts were known to be true just before the check was carried out. A domain-heuristic is called a *double-support heuristic* if it prefers double-support checks to other checks.

On average it is just as likely that a double-support check will succeed as it is that a single-support check will succeed—in both cases one out of two checks will succeed on average. The potential payoff of a double-support check is twice as large as that of a single-support check. This is our first indication that at domain-level arc-consistency algorithms should prefer double-support checks to single-support checks.

Our second indication that arc-consistency algorithms should prefer double-support checks is the insight that in order to minimise the total number of support-checks it is a necessary condition to maximise the number of successful double-support checks.

In the following section we shall point out a third indication—more compelling than the previous two—that arc-consistency algorithms should prefer double-support checks to single-support checks. Our average time-complexity results will demonstrate that the double-support heuristic is superior to lexicographical heuristic which is usually used.

2.2 Related Literature

In 1977, Mackworth presented an arc-consistency algorithms called AC-3 [6]. Together with Freuder he presented a lower bound of $\mathbf{\Omega}(ed^2)$ and an upper bound of $\mathbf{O}(ed^3)$ for the worst-case time-complexity [7]. Here, e is the number of constraints and d is the maximum domain-size.

AC-3, as presented by Mackworth, is not an algorithm as such; it is a *class* of algorithms which have certain data-structures in common and treat them similarly. The most prominent data-structure used by AC-3 is a *queue* which initially contains each of the pairs (α, β) and (β, α) for which there exists a constraint between α and β. The basic machinery of AC-3 is such that *any* tuple can be removed from the queue. For a "real" implementation this means that heuristics determine the choice of the tuple that was removed from the queue. By selecting a member from the queue, these heuristics determine the constraint that will be used for the next support-checks. Such heuristics will be called *arc-heuristics*.

Not only are there arc-heuristics for AC-3, but also there are heuristics which, given a constraint, select the values in the domains of the variables that will be used for the next support-check. Such heuristics we will call *domain-heuristics*.

Experimental results from Wallace and Freuder clearly indicate that arc-heuristics influence the average performance of arc-consistency algorithms [13]. Gent *et al* have made similar observations [5].

Bessière, Freuder, and Régin present another class of arc-consistency algorithms called AC-7 [1]. AC-7 is an instance of the AC-Inference schema, where support-checks are saved by making inference. In the case of AC-7 inference is made at domain-level, where it is exploited that $M_{ij} = M_{ji}^T$, where $\cdot^T$ denotes transposition. AC-7 has an optimal upper bound of $\mathbf{O}(ed^2)$ for its worst-case time-complexity and has been reported to behave well on average.

In their paper, Bessière, Freuder, and Régin present experimental results that the AC-7 approach is superior to the AC-3 approach. They present results of applications of MAC-3 and MAC-7 to real-world problems. Here MAC-i is a backtracking algorithm which uses AC-i to maintain arc-consistency during search [8].

In [12] van Dongen and Bowen present results from an experimental comparison between AC-7 and AC-3_b which is a cross-breed between Mackworth's AC-3 and Gaschnig's DEE [4]. At the domain-level AC-3_b uses a double-support heuristic. In their setting, AC-7 was equipped with a lexicographical arc and domain-heuristic. AC-3_b has the same worst-case time-complexity as AC-3. In van Dongen and Bowen's setting it turned out that AC-3_b was more efficient than AC-7 for the majority of their $30,420$ random problems. Also AC-3_b was more efficient on average. These are surprising results because AC-3_b—unlike AC-7—has to repeat support-checks because it cannot remember them. The results are an indication that domain-heuristics can improve arc-consistency algorithms.

AC-3_d is another arc-consistency algorithm which uses a double-support heuristic [11,10]. It also maintains the benign space-complexity of AC-3. Experimental results suggest that AC-3_d is a good algorithm. For example, in all test cases between AC-3, AC-3_d and AC-7 which were considered in [11,10] AC-3_d was always first or second best. The test cases considered both time on the wall and support-checks. For about one out of two cases AC-3_d was the best. For the time on the wall these results are not as accurate as possible because algorithms were not carried out on the same machine (but times were adjusted to overcome the difference in CPU time). At the moment we are in the process of carrying out a comprehensive comparison between AC-3_d and AC-2001 [2]. Initial results are encouraging. We will report on our findings in a future paper.

2.3 The General Problem

In this section we shall discuss the reasons for, and the consequences of, our decision to study only two-variable CSPs.

One problem with our choice is that we have eliminated the effects that arc-heuristics have on arc-consistency algorithms. Wallace and Freuder, and Gent *et al* showed that arc-heuristics have effects on the performance of arc-consistency algorithms [13,5]. Our study does not properly take the effects of arc-heuristics into account. However, later in this section we will argue that a double-support heuristic should be used at domain-level.

Another problem with our simplification is that we cannot properly extrapolate average results for two-variable CSPs to the case where arc-consistency algorithms are used as part of MAC algorithms. For example, in the case of a two-variable CSP, on average about one out of every two support-checks will succeed. This is not true in MAC search because most time is spent at the leaves of the search-tree and most support-checks in

that region will fail. A solution would be to refine our analysis to the case where different ratios of support-checks succeed.

We justify our decision to study two-variable CSPs by two reasons. Our first reason is that at the moment the general problem is too complicated. We study a simpler problem hoping that it will provide insight in the effects of domain-heuristics in general and the successfulness of the double-support heuristic in particular.

Our second reason to justify our decision to study two-variable CSPs is that some arc-consistency algorithms find support for the values in the domains of two variables at the same time. For such algorithms a good domain-heuristic is important and it can be found by studying two-variable CSPs only.

3 Two Arc-Consistency Algorithms

3.1 Introduction

In this section we shall introduce two arc-consistency algorithms and present a detailed case-study to compare the average time-complexity of these algorithms for the case where the domain-sizes of both variables is two. The two algorithms differ in their domain-heuristic. The algorithms under consideration are a *lexicographical algorithm* and a *double-support algorithm*. The lexicographical algorithm will be called $\mathcal{L}$. The double-support algorithm will be called $\mathcal{D}$. We shall see that for the problem under consideration $\mathcal{D}$ outperforms $\mathcal{L}$.

3.2 The Lexicographical Algorithm $\mathcal{L}$

In this section we shall define the *lexicographical arc-consistency algorithm* called $\mathcal{L}$ and discuss its application to two by two matrices. We shall first define $\mathcal{L}$ and then discuss the application.

$\mathcal{L}$ does not repeat support-checks. $\mathcal{L}$ first tries to establish its row-support. It does this for each row in the lexicographical order on the rows. When it seeks support for row i it tries to find the lexicographical smallest column which supports i. After $\mathcal{L}$ has computed its row-support, it tries to find support for those columns whose support-status is not yet known. It does this in the lexicographical order on the columns. When $\mathcal{L}$ tries to find support for a column j, it tries to find it with the lexicographically smallest row that was not yet known to support j.

Pseudo code for $\mathcal{L}$ is depicted in Figure 1. It is left to the reader to verify that the space-complexity of the algorithm is $\mathbf{O}(d)$, where $d = \max(a, b)$. It is a straightforward exercise to prove correctness [10].

Figure 2 is a graphical representation of all traces with respect to $\mathcal{L}$. Each different path from the root to a leaf corresponds to a different trace with respect to $\mathcal{L}$. Each trace of length l is represented in the tree by some unique path that connects the root and some leaf via $l-1$ internal nodes. The root of the tree is an artificial 0-th member of the traces. The nodes/leaves at distance l from the root correspond to the l-th members of the traces, for $0 \leq l \leq ab = 4$.

Nodes in the tree are decision points. They represent the support-checks which are carried out by $\mathcal{L}$. A branch of a node n that goes straight up represents the fact that

```
constant UNSUPPORTED = −1;

procedure L( a by b constraint M ) = begin
   /* Initialisation. */
   for each row r do begin
      rsupp[r] := UNSUPPORTED;
      for each column c do
         checked[r][c] := UNSUPPORTED;
   end;
   for each column c do
      csupp[c] := UNSUPPORTED;

   /* Find row-support. */
   for each row r do begin
      c := 1;
      while (c ≤ b) and (rsupp[r] = UNSUPPORTED) do begin
         if (M?rc = 1) then begin
            rsupp[r] := c;
            csupp[c] := r;
         end;
         c := c + 1;
      end;
      if (rsupp[r] = UNSUPPORTED) then
         remove_row(r);
   end;

   /* Complete column-support. */
   for each column c do begin
      r := 1;
      while (r ≤ a) and (csupp[c] = UNSUPPORTED) do begin
         if (not checked[r][c]) then
            if (M?rc = 1) then
               csupp[c] := r;
         r := r + 1;
      end;
      if (csupp[c] = UNSUPPORTED) then
         remove_col(c);
   end;
end;
```

Fig. 1. Algorithm $\mathcal{L}$

a support-check, say $M^?_{ij}$, is successful. A branch to the right of that same node n represents the fact that the same $M^?_{ij}$ is unsuccessful. The successful and unsuccessful support-checks are also represented at node-level. The i-th row of the j-th column of a node does not contain a number if the check $M^?_{ij}$ has not been carried out. Otherwise, it contains the number M_{ij}. It is only by studying the nodes that it can be found out which support-checks have been carried out so far.

Remember that we denote the number of rows by a and the number of columns by b. Note that there are $2^{ab} = 2^4 = 16$ different two by two zero-one matrices, and that traces of different matrices with respect to $\mathcal{L}$ can be the same. To determine the average time-complexity of $\mathcal{L}$ we have to add the lengths of the traces of each of the matrices and divide the result by 2^{ab}. Alternatively, we can compute the average number of support-checks if we subtract from ab the sum of the *average savings* of each of the matrices. Here, the *savings* of a matrix M are given by $ab - l$ and its *average savings* are

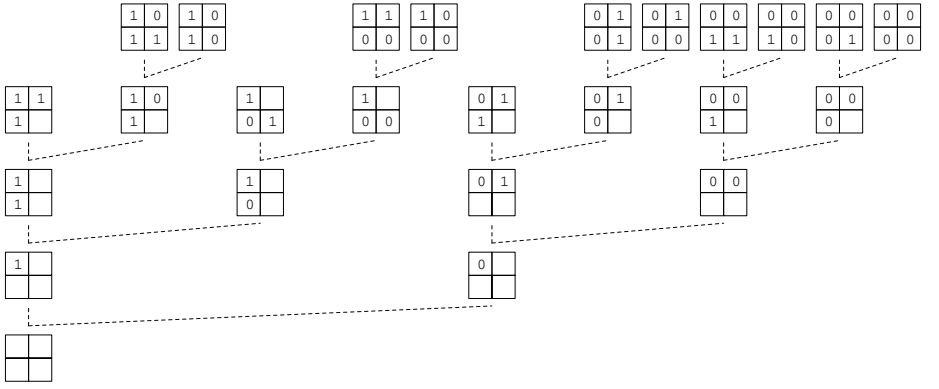


Fig. 2. Traces of $\mathcal{L}$. Total number of support-checks is 58

given by $(ab-l)/2^{ab}$, where l is the length of the trace of M with respect to $\mathcal{L}$. Similarly, we define the *(average) savings* of a trace to be the sum of the (average) savings of all the matrices that have that trace.

It is interesting to notice that all traces of $\mathcal{L}$ have a length of at least three. Apparently, $\mathcal{L}$ is not able to determine its support in fewer than three support-checks—not even if a matrix does not contain any zero at all. It is not difficult to see that $\mathcal{L}$ will always require at least $a + b - 1$ support-checks regardless of the values of a and b.

$\mathcal{L}$ could only have terminated with two checks had both these checks been successful. If we focus on the strategy $\mathcal{L}$ uses for its second support-check for the case where its first support-check was successful we shall find the reason why it cannot accomplish its task in fewer than three checks. After $\mathcal{L}$ has successfully carried out its first check $M^?_{11}$ it needs to learn only *two* things. It needs to know if 2 is in the row-support *and* it needs to know if 2 is in the column-support. The next check of $\mathcal{L}$ is $M^?_{21}$. Unfortunately, this check can only be used to learn *one* thing. *Regardless of whether the check* $M^?_{12}$ *succeeds or fails*, another check *has* to be carried out.

If we consider the case where the check $M^?_{22}$ was carried out as the second support-check we shall find a more efficient way of establishing the support. The check $M^?_{22}$ is a double-support check. It offers the potential of learning about *two* new things. If the check is successful then it offers the knowledge that 2 is in the row-support *and* that 2 is in the column-support. Since this was all that had to be found out the check $M^?_{22}$ offers the potential of termination after two support-checks. What is more, one out of every two such checks will succeed. Only if the check $M^?_{22}$ fails do more checks have to be carried out. Had the check $M^?_{22}$ been used as the second support-check, checks could have been saved on average.

Remember that the same trace in the tree can correspond to different matrices. The Trace Theorem states that if l is the length of a trace then there are exactly 2^{ab-l} matrices which have the same trace. The shortest traces of $\mathcal{L}$ are of length $l_1 = 3$. $\mathcal{L}$ finds exactly $s_1 = 3$ traces whose lengths are l_1. The remaining l_2 traces all have length $l_2 = ab$. Therefore, $\mathcal{L}$ saves $(s_1 \times (ab - l_1) \times 2^{ab-l_1} + s_2 \times (ab - l_2) \times 2^{ab-l_2})/2^{ab} =$

$(3 \times (4-3) \times 2^{4-3} + 0)/2^4 = 3 \times 1 \times 2^1/2^4 = 3/8$ support-checks on average. $\mathcal{L}$ therefore requires an average number of support-checks of $ab - \frac{3}{8} = 4 - \frac{3}{8} = 3\frac{5}{8}$. In the following section we shall see that better strategies than $\mathcal{L}$'s exist.

3.3 The Double-Support Algorithm $\mathcal{D}$

In this section we shall introduce a second arc-consistency algorithm and analyse its average time-complexity for the special case where the number of rows a and the number of columns b are both two. The algorithm will be called $\mathcal{D}$. It uses a double-support heuristic as its domain-heuristic.

Pseudo code for $\mathcal{D}$ is depicted in Figure 3. It is easy to verify that the space-complexity of the algorithm is $\mathbf{O}(a+b)$. The reader is referred to [10] for a correctness proof. Using a simple program transformation technique the two for-each statements at the start of the algorithm can be avoided. For our time-complexity results which are in terms of support-checks these transformations make no difference. The reader is referred to [10] for further detail.

$\mathcal{D}$'s strategy is a bit more complicated than $\mathcal{L}$'s. It will use double-support checks to find support for its rows in the lexicographical order on the rows. It does this by finding for every row the lexicographically smallest column whose support-status is not yet known. When there are no more double-support checks left, $\mathcal{D}$ will use single-support checks to find support for those rows whose support-status is not yet known and then find support for those columns whose support status is still not yet known. When it seeks support for a row/column, it tries to find it with the lexicographically smallest column/row that is not yet known to support that row/column.

We have depicted the traces of $\mathcal{D}$ in Figure 4. It may not be immediately obvious, but $\mathcal{D}$ is more efficient than $\mathcal{L}$. The reason for this is as follows. There are two traces whose length is shorter than $ab = 4$. There is $s_1 = 1$ trace whose length is $l_1 = 2$ and there is $s_2 = 1$ trace whose length is $l_2 = 3$. The remaining s_3 traces each have a length of $l_3 = ab$. Using the Trace Theorem we can use these findings to determine the number of support-checks that are saved on average. The average number of savings of $\mathcal{D}$ are given by $(s_1 \times (ab - l_1) \times 2^{ab-l_1} + s_2 \times (ab - l_2) \times 2^{ab-l_2} + s_3 \times (ab - l_3) \times 2^{ab-l_3})/2^{ab} = (2 \times 2^2 + 1 \times 2^1 + 0)/2^4 = 5/8$. This saves $1/4$ checks more on average than $\mathcal{L}$.

It is important to observe that l_1 has a length of only two and that it is the result of a sequence of two successful double-support checks. It is this trace which contributed the most to the savings. As a matter of fact, this trace by itself saved more than the *total* savings of $\mathcal{L}$.

The strategy used by $\mathcal{D}$ to prefer double-support checks to single-support checks leads to shorter traces. We can use the Trace Theorem to find that that the average savings of a trace are $(ab - l)2^{-l}$, where l is length of the trace. The double-support algorithm was able to produce a trace that was smaller than any of those produced by the lexicographical algorithm. To find this trace had a big impact on the total savings of $\mathcal{D}$. $\mathcal{D}$ was only able to find the short trace because it was the result of a sequence of successful double-support checks and its heuristic forces it to use as many double-support checks as it can. Traces which contain many successful double-support checks contribute much to the total average savings. This is our third and last indication that arc-consistency algorithms should prefer double-support checks to single-support checks.

```
constant UNSUPPORTED = −1;
constant DOUBLE = −2;

procedure D( a by b constraint M ) = begin
    /* Initialisation. */
    for each row r do begin
        rkind[r] := UNSUPPORTED;
        rsupp[r] := UNSUPPORTED;
    end;
    for each column c do
        csupp[c] := UNSUPPORTED;

    /* Find row-support. */
    for each row r do begin
        c := 1;
        /* First try to find support for r using double-support checks. */
        while (c ≤ b) and (rsupp[r] = UNSUPPORTED) do begin
            if (csupp[c] = UNSUPPORTED) then begin
                /* M?rc is a double-support check. */
                if (M?rc = 1) then begin
                    rkind[r] := DOUBLE;
                    rsupp[r] := c;
                    csupp[c] := r;
                end;
            end;
        c := 1;
        /* If r is still unsupported then try to find support using single-support checks. */
        while (c ≤ b) and (rsupp[r] = UNSUPPORTED) do begin
            if (csupp[c] ≠ UNSUPPORTED)then begin
                /* M?rc is a single-support check. */
                if (M?rc = 1) then
                    rsupp[r] := c;
            end;
            c := c + 1;
        end;
        if (rsupp[r] = UNSUPPORTED) then
            remove_row(r);
    end;

    /* Complete column-support. */
    for each column c do begin
        r := 1;
        while (r ≤ a) and (csupp[c] = UNSUPPORTED) do begin
            if (rsupp[r] < c) and (rkind[r] = DOUBLE) then
                if (M?rc = 1) then begin
                    csupp[c] := r;
                    rsupp[r] := c;
                end;
            r := r + 1;
        end
        if (csupp[c] = UNSUPPORTED) then
            remove_col(c);
    end;
end;
```

Fig. 3. Algorithm $\mathcal{D}$

3.4 A First Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we have studied the average time-complexity of the lexicographical algorithm $\mathcal{L}$ and the double-support algorithm $\mathcal{D}$ for the case where the size of the domains

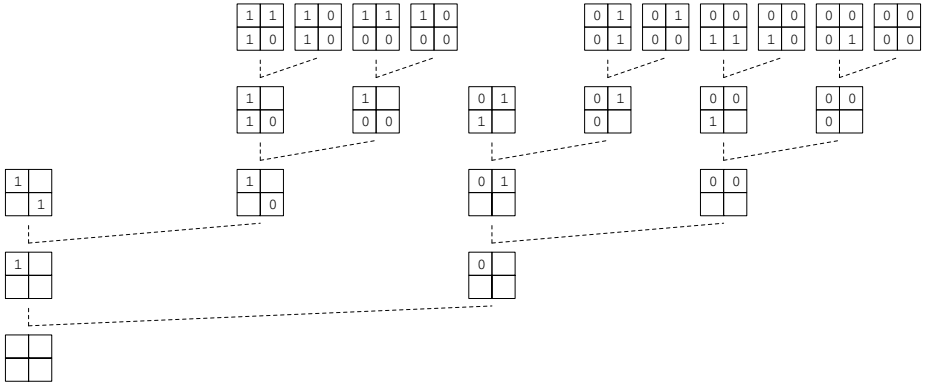


Fig. 4. Traces of $\mathcal{D}$. Total number of support-checks is 54

is two. We have found that for the problem under consideration $\mathcal{D}$ was more efficient on average than $\mathcal{L}$.

4 Average Time-Complexity Results

In this section we shall present average time-complexity results for $\mathcal{L}$ and $\mathcal{D}$. The reader is referred to [9] for proof and further information.

Theorem 2 (Average Time Complexity of $\mathcal{L}$). *Let a and b be positive integers. The average time complexity of $\mathcal{L}$ over $\mathbb{M}^{ab}$ is* exactly $\operatorname{avg}_{\mathcal{L}}(a,b)$*, where*

$$\operatorname{avg}_{\mathcal{L}}(a,b) = a(2-2^{1-b}) + (1-b)2^{1-a} + 2\sum_{c=2}^{b}(1-2^{-c})^a.$$

Following [3, Page 59] we obtain the following accurate estimate.

Corollary 1.

$$\operatorname{avg}_{\mathcal{L}}(a,b) \approx 2a + 2b - 2\log_2(a) - 0.665492.$$

Here, $\log_2(\cdot)$ is the base-2 logarithm. This estimate is already good for relatively small a and b. For example, if $a = b = 10$ then the absolute error

$$|\operatorname{avg}_{\mathcal{L}}(a,b) - (2a + 2b - 2\log_2(a) - 0.665492)| / \operatorname{avg}_{\mathcal{L}}(a,b)$$

is less than 0.5%.

We can also explain the results we have obtained for the bound in Corollary 1 in the following way. $\mathcal{L}$ has to establish support for each of its a rows and b columns except for the l columns which were found to support a row when $\mathcal{L}$ was establishing its row-support. Therefore, $\mathcal{L}$ requires about $2a + 2(b-l)$. To find l turns out to be

easy because on average $a/2$ rows will be supported by the first column. From the remaining $a/2$ rows on average $a/4$ rows will be supported by the second column, ..., from the remaining 1 rows on average $1/2$ will find support with the $\log_2(a)$-th column, i.e. $l \approx \log_2(a)$. This informal reasoning demonstrates that on average $\mathcal{L}$ will require about $2a + 2b - 2\log_2(a)$ support-checks and this is almost exactly what we found in Corollary 1.

Theorem 3 (Average Time Complexity of $\mathcal{D}$). *Let a and b be non-negative integers. The average time complexity of $\mathcal{D}$ over $\mathbb{M}^{ab}$ is* exactly $\mathrm{avg}_{\mathcal{D}}(a,b)$, *where* $\mathrm{avg}_{\mathcal{D}}(a,0) = \mathrm{avg}_{\mathcal{D}}(0,b) = 0$, *and*

$$\begin{aligned}\mathrm{avg}_{\mathcal{D}}(a,b) = 2 &+ (b-2)2^{1-a} + (a-2)2^{1-b} + 2^{2-a-b} \\ &- (a-1)2^{1-2b} + 2^{-b}\,\mathrm{avg}_{\mathcal{D}}(a-1,b) \\ &+ (1-2^{-b})\,\mathrm{avg}_{\mathcal{D}}(a-1,b-1)\end{aligned}$$

if $a \neq 0$ and $b \neq 0$

Let a and b be positive integers such that $a + b \geq 14$. The following upper bound for $\mathrm{avg}_{\mathcal{D}}(a,b)$ is presented in [9]:

$$\begin{aligned}\mathrm{avg}_{\mathcal{D}}(a,b) < 2\max(a,b) + 2 \\ - (2\max(a,b) + \min(a,b))2^{-\min(a,b)} \\ - (2\min(a,b) + 3\max(a,b))2^{-\max(a,b)}.\end{aligned}$$

An important result that follows from the upper bound for $\mathrm{avg}_{\mathcal{D}}(a,b)$ is that $\mathcal{D}$ is "almost optimal." The proof relies on the fact that *any* arc-consistency algorithm will require at least about $2\max(a,b)$ checks on average. Therefore, if $14 \leq a + b$, then $\mathrm{avg}_{\mathcal{D}}(a,b) - \mathrm{avg}_{\mathcal{A}}(a,b) < 2$ for any arc-consistency algorithm $\mathcal{A}$, hence the optimality claim.

5 Comparison of $\mathcal{L}$ and $\mathcal{D}$

In this section we shall briefly compare the average time-complexity of $\mathcal{L}$ and $\mathcal{D}$. We already observed in Section 3.2 that the minimum number of support-checks required by $\mathcal{L}$ is $a + b - 1$. In Section 4 we have presented the bound $\mathrm{avg}_{\mathcal{D}}(a,b) < \max(a,b) + 2$, provided that $a + b \geq 14$. If $a + b \geq 14$ and $a = b$ then the *minimum* number of support-checks required by $\mathcal{L}$ is almost the same as the *average* number of support-checks required by $\mathcal{D}$!

Our next observation sharpens the previous observation. It is the observation that on average $\mathcal{D}$ is a better algorithm than $\mathcal{L}$ because its upper bound is lower than the bound that we derived for $\mathcal{L}$. When a and b get large and are of the same magnitude then the difference is about $a + b - 2\log_2((a+b)/2)$ which is quite substantial.

Another important result is the observation that $\mathcal{D}$ is almost "optimal." If $a + b \geq 14$ and if $\mathcal{A}$ is any arc-consistency algorithm then $\mathrm{avg}_{\mathcal{D}}(a,b) - \mathrm{avg}_{\mathcal{A}}(a,b) < 2$. To the best of our knowledge, this is the first such result for arc-consistency algorithms.

6 Conclusions and Recommendations

In this paper we have studied two domain-heuristics for arc-consistency algorithms for the special case where there are two variables. We have defined two arc-consistency algorithms which differ only in the domain-heuristic they use. The first algorithm, called $\mathcal{L}$, uses a lexicographical heuristic. The second algorithm, called $\mathcal{D}$, uses a heuristic which gives preference to double-support checks. We have presented a detailed case-study of the algorithms $\mathcal{L}$ and $\mathcal{D}$ for the case where the size of the domains of the variables is two. Finally, we have presented average time-complexity results for $\mathcal{L}$ and $\mathcal{D}$.

We have defined the notion of a *trace* and have demonstrated the usefulness of this notion. In particular we have shown that the average savings of a trace are $(ab - l)2^{-l}$, where l is the length of the trace and a and b are the sizes of the domains of the variables.

Our average time-complexity results have demonstrated that $\mathcal{D}$ is more efficient than $\mathcal{A}$. Furthermore, we have demonstrated that $\mathcal{D}$ is almost optimal in a certain sense; if $a + b \geq 14$ then $\mathrm{avg}_{\mathcal{D}}(a, b) - \mathrm{avg}_{\mathcal{A}}(a, b) < 2$ for any arc-consistency algorithm $\mathcal{A}$.

The work that was started here should be continued in the form of a refinement of our analysis for the case where only every m-th out of every n-th support-check succeeds. This will provide an indication of the usefulness of the two heuristics under consideration when they are used as part of a MAC-algorithm. Furthermore, we believe that it should be worthwhile to tackle the more complicated problems where there are more than two variables in the CSP and where the constraints involve more than two variables. Finally, we believe that it should be interesting to implement an arc-consistency algorithm which does not repeat support-checks and which comes equipped with our double-support heuristic as its domain-heuristic.

Acknowledgement. This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

References

1. C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA.
2. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the 17 International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 309–315, 2001.
3. P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Mellin transform asymptotics. Technical Report Research Report 2956, INRIA, 1996.
4. J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceeding of the Second Biennial Conference, Canadian Society for the Computational Studies of Intelligence*, pages 268–277, 1978.
5. I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'1997)*, pages 327–340. Springer, 1997.
6. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

7. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–73, 1985.
8. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley & Sons, 1994.
9. M.R.C. van Dongen. *Constraints, Varieties, and Algorithms*. PhD thesis, Department of Computer Science, University College, Cork, Ireland, 2002.
10. M.R.C. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. Technical Report TR-01-2002, Cork Constraint Computation Centre, 2002.
11. M.R.C. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, pages 755–760. Springer, 2002.
12. M.R.C. van Dongen and J.A. Bowen. Improving arc-consistency algorithms with double-support checks. In *Proceedings of the Eleventh Irish Conference on Artificial Intelligence and Cognitive Science (AICS'2000)*, pages 140–149, 2000.
13. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI '92*, pages 163–169, Vancouver, British Columbia, Canada, 1992.

Computing Explanations and Implications in Preference-Based Configurators

Eugene C. Freuder[1], Chavalit Likitvivatanavong[1], Manuela Moretti[2], Francesca Rossi[2], and Richard J. Wallace[1]

[1] Cork Constraint Computation Centre
Department of Computer Science, University College Cork,
Cork, Ireland
{e.freuder,chavalit,rjw}@4c.ucc.ie
[2] Department of Pure and Applied Mathematics,
University of Padova, Italy.
mmoretti@studenti.math.unipd.it,
frossi@math.unipd.it

Abstract. We consider configuration problems with preferences rather than just hard constraints, and we analyze and discuss the features that such configurators should have. In particular, these configurators should provide explanations for the current state, implications of a future choice, and also information about the quality of future solutions, all with the aim of guiding the user in the process of making the right choices to obtain a good solution.
We then describe our implemented system, which, by taking the soft n-queens problem as an example, shows that it is indeed possible, even in this very general context of preference-based configurators, to automatically compute all the information needed for the desired features. This is done by keeping track of the inferences that are made during the constraint propagation enforcing phases.

1 Introduction

One of the most important features of problem solving in an interactive setting is the capacity of the system to provide the user with justifications, or explanations, for its operations. Such justifications are especially useful when the user is interested in what happens at any time during search, because he/she can alter features of the problem to facilitate the problem solving process.

Basically, the aim of an explanation is to show clearly why a system acted in a certain way after certain events. In this context, explanations can be viewed as answers to user questions like the following: Why isn't it possible to obtain a solution? Why is there a conflict between these parts of the system? Why did the system select this component? Why does this parameter have to have this value?

There are various formal definitions of explanation in the literature, which differ with respect to the context in which explanations are put. The basic idea,

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 76–92, 2003.

in the context of searching for a solution, is that an explanation is a set of "elements" of the problem that is sufficient to deduce another "element" whose selection or removal has to be explained [5].

In this paper, we consider explanations in the context of an interactive system that allows the user solve configuration problems. For these kinds of problems, it is essential to take user preferences into account in finding an acceptable solution. However, once these preferences are made known to the system, they will affect its subsequent actions. As a result, answers to questions such as those above will necessarily involve these preferences.

Previous studies of explanation in the context of problem solving have not considered preferences: elements of the problem were simply taken as givens, i.e. as either true or false, and inferences were made on this basis [1,5,7,3]. The present work extends these studies by incorporating preferences into explanations, when appropriate, at all stages of problem solving. As might be expected, this enhancement involves special difficulties both in deriving and revising explanations, which are discussed in later sections.

In addition to providing explanations, interactive systems should be able to show the consequences, or implications, of an action to the user, which may be useful in deciding which choice to make next. In this way, they can provide a sort of "what-if" kind of reasoning, which guides the user towards good future choices. Implications can be viewed as answers to questions like the following: What would happen if part c could only take on these values? What would happen if part c were added again?

Fortunately, this capacity can be implemented with the same machinery that is used to give explanations. In the present work, we show how this feature can also be extended to the case where there are preferences.

2 Configuration Problems

Configurators. A configurator [10,8,13,14] can be regarded as a system which interacts with a user to help him/her to configure a product. A product can be seen as a set of component types, where each type corresponds to a certain finite number of concrete components, and a set of compatibility constraints among subsets of the component types. A user configures a product by choosing a concrete component for each component type, such that all the compatibility constraints as well as personal preferences are satisfied.

Personal preferences are established during interaction with the configurator by incrementally posting "personal" constraints (which may be different from user to user) over the component types. In some cases, component types are not all known in advance, but arise only after the user has posted some of his/her personal constraints.

Examples. A typical example of this scenario is the problem of configuring a personal computer (PC) by choosing all its components. In this case, a compatibility constraint may say that some mother-boards are not compatible with some

processors, and a personal constraint could say, for example, that only IBM hard disks are desired.

Another example may concern the choice of a car, via the selection of all its features (engine, chassis, color, ...). In this case, a compatibility constraint could say that only some engines are available for some chassis, and a personal constraint could say that the car should be red, or that the chassis should be convertible.

A third example, where the component types are not known at the beginning, is a PC configuration problem where the user can choose either a combo CD/DVD driver or two single components. In this case, the new component types (one or two) will appear only after the user has selected either the combo system or the two components.

Desired features for configurators. In the interaction between a configurator and a user, the user usually makes successive choices over the component types, specifying in this way his/her personal constraints, and the configurator should help him/her in several respects, mainly in trying to avoid conflicting situations, where the user choices and the compatibility constraints are not consistent [6]. In fact, such conflicting situations would not lead to any complete product configuration. Examples of desired help from the configurator are:

- the configurator should show the consequences of any possible future user choice: this can help making the next choice in a way that avoids conflicts both at the next step and also at the subsequent steps;
- as soon as a conflict arises, the configurator should show why such a conflict has arisen, and what choice(s) could be retracted to come back to a non-conflicting situation;
- a user choice could make some other choices unfeasible (because of the compatibility constraints): such unfeasible choices should be shown and justified to the user.

Constraint-based configurators. Constraint-based technology is currently used in many configurators to both model and solve configuration problems [1,10]. In this approach, component types are represented by variables, having as many values as the concrete components, and both compatibility and personal constraints are represented as constraints over subsets of such variables. At present, user choices during the interaction with the configurator are usually restricted to specifying unary constraints, in which a certain value is selected for a variable. However, they could be extended to more general non-unary constraints.

Whenever a choice is made, the corresponding (unary) constraint is added to existing compatibility and personal constraints, and some constraint propagation notion is enforced, for example arc-consistency (AC) [16], to rule out (some of the) future choices that are not compatible with the current choice. While providing justifications based on search is difficult, arc-consistency enforcing has been used as a source of guidance for justifications, and it has been exploited to help the users in some of the scenarios mentioned above. For example, in [5], it is

shown how AC enforcement can be used to provide both justifications for choice elimination, and also guidance for conflict resolution. Other related approaches to providing justifications and explanations can be found in [7,17].

In constraint-based configurators, explanation elements are usually constraints, including previous user assignments. Therefore, an explanation is a set of constraints (either those belonging to the original problem, or assignments made during search) that is sufficient to deduce some other constraints, as for example the removal of a value from a domain, or the forced assignment of a variable.

3 Preferences in Constraints and Configurators

In this paper we follow the same approach as in [5], in the sense that we use constraint-based technology and we provide explanation and conflict-resolution help via AC enforcing. However, we do this in the more general context of preference-based configurators. In fact, the configurator scenario we have described in the previous section assumes that both the compatibility constraints and the user constraints are hard, that is, they are either satisfied or violated.

In many real-life applications, however, both these kinds of constraints may instead be just preferences, thus allowing for more than two degrees of satisfaction. For example, in a car configuration problem, a user may prefer red cars, but may also not want to completely rule out other colors. Thus red will get a higher preference with respect to the other colors. In a PC configuration problem, some combinations of mother-boards and processors may be preferable to others.

Soft constraints. Preferences, or *soft constraints* as we may call them, can easily be modeled as classical constraints where each combination of values for the variables is associated to an element, which is usually interpreted as the preference for that combination, or its importance level [2]. Note that variable domains can be seen as unary constraints, thus it is also possible to associate preferences to single variable values. The elements which are used as preference levels have to be ordered, in order to know which elements represent higher (or lower) preferences.

Given a set of soft constraints, a complete variable assignment inherits its preference level by combining the preference levels it selects in the constraints. Solutions are therefore ranked by their preference level, and thus soft constraint problems are constraint optimization problems, where one usually looks for an optimal solution, that is, a solution which has the highest preference level (according to the ordering of the preference levels).

For example, a typical soft constraint scenario, that we will use extensively in this paper, is the fuzzy constraint framework [11,4], where preferences are real numbers between 0 and 1, with a higher number denoting a higher preference, and are combined via the minimum operator. Thus the preference value of a solution is the minimum among the preference values selected by this solution on the constraints, and the best solutions are those with maximum preference value.

In another scenario, one may want to minimize the sum of the costs (which are usually real numbers). Note that even classical hard constraints could be seen as soft ones, where there are just two levels of preference (allowed and not-allowed), with allowed better than not-allowed, and with logical AND as the combination operator.

More complex structures and orderings can be used in this context, to provide a more sophisticated way of comparing solutions. For example, one may want to both maximize the minimum preference and also to minimize the sum of the costs. In this way, solutions will be partially ordered according to these two criteria. See [2] for more details about soft constraints.

Search and AC enforcing in soft constraint problems. While searching for a solution can be done via a backtracking search, interleaved with constraint propagation, searching for an optimal solution requires a branch-and-bound approach[1], which again can be interleaved with constraint propagation. In fact, the presence of soft constraints, in some cases like the fuzzy one, does not preclude the use of constraint propagation algorithms (see [2] for a formal proof of a sufficient condition for the safe application of soft constraint propagation).

However, there is a fundamental difference between constraint propagation for hard constraints and for soft ones. Let us consider AC enforcing to make it simple, and also because we will use AC in this paper. In the hard case AC enforcing performs the following change over the domain D_i of variable x_i: $D_i := \{d \in D_i$ such that, for any other variable x_j, there exists $d_j \in D_j$ with $\langle d_i, d_j \rangle$ satisfying the constraint c_{ij} between x_i and $x_j\}$. That is, values for variable x_i that are not compatible with any other variable are deleted from the domain of x_i. It is easy to see that value elimination can be seen as a form of preference decrease: from allowed to not-allowed.

Instead, soft AC enforcing over fuzzy constraints performs the following: for each $d_i \in D_i$, consider its preference value $p(d_i)$; then $p(d_i)$ is set to $max_{x_j \in V, d_j \in D_j}\ min(p(d_i), p(d_j), p(\langle d_i, d_j \rangle))\}$. That is, values are not deleted, but the preference level of each tuple is adjusted by considering other constraints. It is important to notice that running soft AC over soft constraint problems is polynomial in the size of the problem (as is AC over hard constraint problems). More precisely, in a soft constraint problem with n variables, d elements in each variable domain, and l preference values, soft AC takes at worst $n^3 \times d^3 \times l$ steps.

Explanations and preferences. The concept of "best" explanation is not uniquely determined, because it depends on the domain model one works with. The most used criterion for choosing how to compute explanations is that the best explanation is the minimal one, with respect to its cardinality [5,17,7]. Another way to solve this problem is to put some preferences into the problem, so that explanations are ordered and one best explanation exists.

In [15] the main idea is that there could be many domain models that compete as a basis for an explanation (but to compute an explanation, the author uses

[1] An alternative strategy is local search, but we will not consider it in this paper.

only a part of the model, called "view"). Therefore, the explainer has to choose the appropriate view that answers the question in the right context. The selection criteria are called "preferences" and are functions that compare the possible views of the models to order them and to give to the user the most appropriate explanation.

We use preferences in a different way: in our system, preferences are given to the constraints of the problem by a user, and they represent the importance of the constraints in the problem. Then, explanations are sequences of constraints, which have been added by the user during the interaction with the system. We then compute explanations describing why the preferences for some values decrease, and suggesting at the same time which assignment has to be retracted, in order to maximize the evaluation of a solution.

Desired features for preference-based configurators. It is now clear that configurators with soft constraints should help users not only to avoid conflicts or to make the next choice so that a smaller number of later choices are eliminated, but also to get to an optimal (or good enough) solution. More precisely, when the user is about to make a new choice for a component type, the configurator should show the consequences of such a choice in terms of conflicts generated, elimination of subsequent choices, and also quality of the solutions. In this way, the user can make a choice which leads to no conflict, and which presents a good compromise between choice elimination and solution quality.

As for the hard case, all this can be done via the help of the soft AC enforcing algorithm, which is able to remember what caused a certain preference decrease. Thus, when a user wants to know why a certain value has a certain preference, which is lower than the initial one, the configurator is able to justify that, and the justification is just a sequence of previous user choices.

Note that the configurator must remember a sequence of previous choices, and not just one, as in the hard case, because here preferences can be lowered at several stages. For example, suppose a preference changes from 1 to 0.7 because of use choice C1, then from 0.7 to 0.4 because of another choice C2. Then, when a user wants to know why the current preference of 0.4, the system must give him/her both C1 and C2 (in the correct sequence) as the reason. In fact, this is the whole history of the causes that produced the current preference. It is crucial to record the whole history, and not just the last preference change, since we must also be able to retract some steps and go back to previous states.

Summarizing, the questions that a preference-based configurator must be able to answer include the following: Why has this configuration (or this component) such a preference level? Why did this sequence of choices lead to a conflict? What should I do to remove this conflict? What effect will this choice have on the future choices and on the quality of the solutions? In particular, are there choices that will lead to a conflict?

While the first two questions refer to a situation in which a user tries to understand the consequences of what has been done already, the last one tries to prevent undesired consequences by guiding the choices of the user towards good solutions and no conflict.

4 A Case-Study

We developed a system which shows the above mentioned features for preference-based configurators in the context of a variant of the n-queens problem where fuzzy preferences have been added. The system is available at the URL `http://www.math.unipd.it/~frossi/soft-config.html`.

N-queens with preferences. In the n-queens problem, n queens must be placed on a n by n board in such a way that no queen can attack each other. Recall that a queen attacks any other queen which is on the same row, column, or diagonal.

This problem is only apparently far from the configuration scenario we have described above. In fact, each queen can be seen as a component type, and the concrete components for a type are the positions for the queen. Moreover, the attack rules for the queens are the compatibility constraints, and positioning a queen in a certain cell means choosing a certain concrete component for a component type. A solution to the n-queens problem is therefore a complete configuration where, for each component type, we have chosen a concrete component, in such a way that the compatibility constraints are satisfied.

Notice that, in this system, all the component types (that is, the rows) are known at the beginning. Moreover, the user choices (that is, the queen positionings) are just unary constraints.

In this paper we consider a variant of the classical n-queens problem to model preference-based configurators. In particular, we associate a preference to each cell of the board, and also to each pair of positions within an attack constraint. We use preference levels between 0 and 1, and we aim at maximizing the minimum preference, as in the fuzzy framework [11].

As usual, we have one variable for each row, with the n cells of the row as possible values, and thus constraints hold between two positions on the same column and on the same diagonal. Notice that this implies that there is no constraint between two positions on the same row.

Preference setting. In our system, the user can select any cell and he/she can assign to it the desired preference level (the default one is 1). For the constraints, we allow the user to give to each constraint a level of importance (always between 0 and 1, the default value is 1), from which we automatically obtain the preferences to be assigned to all the pairs of values (recall that constraints are all binary in the n-queens problem). More precisely, the preference level of a pair will depend both on the level of importance of the constraint (the higher the importance level is, the smaller the preference is for pairs under attack), and on the distance between the elements of the pair (elements under attack which are farther apart are tolerated more by giving them a higher preference value). Figure 1 shows the graphical interface of our system, with the list of possible preferences for cell (4,5), and, on the right, the buttons for setting the importance level of the constraints ("Diagonal ur-bl", which stands for the constraint over the diagonal from upper right to bottom left, "Diagonal br-ul", and "Vertical").

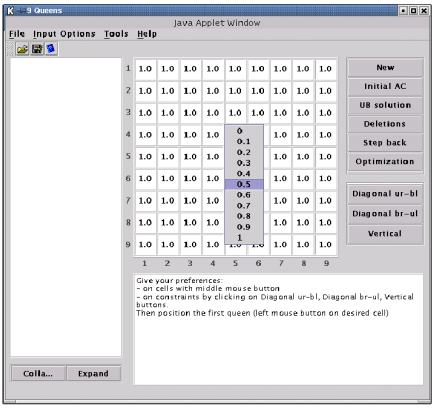


Fig. 1. Graphical interface of our system, with the preferences for cell (4,5).

After the preference setting phase, the user can run an initial soft AC enforcing algorithm via the "initial AC" button, which, as mentioned above, may possibly decrease the preferences of the cells. We recall that enforcing soft AC is polynomial in the size of the given soft constraint problem.

Queen positioning. To position a queen on a cell, we just have to click on the cell with the left mouse button. This will put a queen in that cell, and will also automatically run AC again after the positioning. Thus the content of the cells, i.e., their preference level, may be lowered after the positioning of the new queen. In particular, if the level of preference of a cell gets to 0 (the lowest value, which indicates a complete conflict), the cell is grayed out and an x is shown in the cell (see Figure 4).

Since a queen positioning triggers soft AC, the complexity of this operation is $O(n^3 \times d^3 \times l)$.

Once a queen has been placed on the board, its position can be changed. If we want to move a queen to another cell of the same row, we can just click with the left mouse button on the new position. We can also retract the position of the latest queen that has been put on the board, via the "step back" button on the right. By clicking on this button, the last queen disappears and the last run of AC enforcing is retracted. This means that we go back to the state before the queen positioning.

Some queens can also be positioned by the system, without any input from the user. This happens where there is no other way to position a queen without leading to a conflict. For the cells that have been automatically filled with a queen by the system, the explanation is the set of all other cells of the same row, which have all preference level 0, and which in turn have their explanation lists.

5 Explanations

Our system computes several kinds of information during its interaction with a user. In particular, it is able to explain why a cell has a certain current preference level, showing the causes for this level. Moreover, it is also able to maintain the correctness of this information between state changes which involve new queen positionings or retractions or previously positioned queens.

Explanations for the cell preferences. At any moment, the cells of the board show their current preference level. Figure 2 shows a state of the system, where a queen has already been positioned in cell (4,5). In this state, the vertical constraints preference level has been set to 0.5, the diagonal up-right to bottom-left constraints preference has been set to 1.0, the other diagonal constraints preference has been set to 0.9. The value shown in each cell is its preference level. For example, cell (1,2), which has been highlighted via a circle and an arrow, has been given a preference level of 0.3.

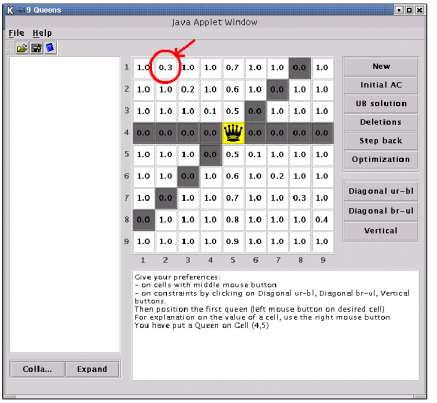


Fig. 2. Each cell shows its current preference level.

To ask for explanations about the current preference of a certain cell, the user can click on the cell with the right mouse button, and a sequence of past choices (that is, queen positions) will appear in the lower panel of the graphical interface. For example, in Figure 3 we have asked for the explanation of the preference of cell (6,5) (after three queen positionings), and we have got the following list of choices: `Queen(4,5) --> 0.6, Queen(3,2) --> 0.3` (see lower panel of the graphical interface). This means that the current preference for cell (6,5), which is 0.3, has been caused by the positioning of a queen in cell (4,5), which lowered the preference from 1 to 0.6, and by a later positioning of a queen in cell (3,2), which has lowered the preference from 0.6 to 0.3. In the same style as in [5], we can also show the list of explanations as a tree, in the left panel, as shown in Figure 3.

In this example, the "explanation" contains more than what is strictly sufficient. This is because in some cases, the entire sequence of choices may be required to explain the current preference. As yet, our system does not distinguish these cases, but instead gives an historical rather than a strictly (or minimal) logical explanation.

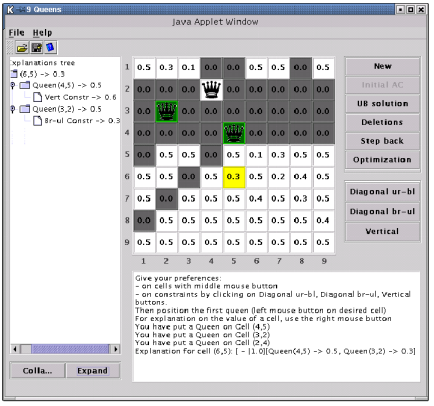


Fig. 3. Explanation: list (lower panel) and tree (left panel).

This representation of the explanation for the preference level of a cell may help the user in both understanding why we are in a certain situation, and also in possibly retracting some previous choices, which are shown to be the causes for a certain low preference in some cells.

From the implementation point of view, to keep track of the explanations for each cell, we associate a list of queen positions and preference levels to each variable value (that is, a cell). At the beginning, such lists are empty. Whenever AC is enforced, if a cell, say C, gets a new lower preference, say p, because of a queen positioning, say Qij, the list for C is augmented with a new element, which contains the pair (Qij, p). Thus, at any moment, the list for cell C contains the sequence of queen positionings which are the causes for the preference changes, from the initial level to the current one.

This bookeeping is interleaved with the soft AC enforcing, since at each AC step we may need to add an element to the list associated to a certain cell.

Notice that the explanation generated for a cell does not depend on the way AC is performed (that is, the order in which AC considers the variables and their values), but only on the sequence of queen positionings. Moreover, the preference value of a cell depends only on the set of queen positionings, i.e. it does not depend on the order in which the queens are put on the board.

However, the explanation does depend on the order of previous queen positionings. In fact, the cause for the current level of preference of a cell, in general, is not just the last queen positioning which changed that level of preference,

but includes also earlier positionings. This is proved by the existence of cases where performing a certain queen positioning, say P, at the beginning, or after some other queen positionings, say $P_1, \ldots, P_n$, leads to different preference values. This means that, in the second case, P is not the only cause for the current preference value. Thus the explanation for the current preference value of a cell should include all queen positionings which have decreased the level of preference of that cell.

Changing state. In our system, a state of the computation is the current state of the board (queen positions and cells preferences), plus the current explanation for each cell, which, as explained above, is a list of previous choices and preference levels. This state can be changed in several ways and for several reasons.

First, we can change the state by positioning a new queen. This means going forward towards a complete solution. Or, we may also go back, because we are not satisfied with the current state. In fact, we may not like the current state because it is a conflict state, that is, some row without a queen has 0 preference values in all its cells. Or we may want to change the state because we don't like the preference values of the cells in the current state. In both cases, we may want to change the state by either retracting the last step, or some previous step.

It is easy to see that, in a state change, the most expensive and complex operation is the update of the explanation lists. It is then easy to imagine that, if we want to retract the last choice, the state change is not so difficult nor expensive, since it just involves the deletion of some last elements from some lists. Thus we just need to go through all the lists (as many as the cells, or, in general, the number of domain values in all domains) and check whether the last element is the queen positioning we want to retract. Thus, if a problem has n variables and d elements in each variable domain, the time complexity of this operation is $O(n \times d)$.

However, if we need to change the position of a queen which is not the last one to have been positioned, then the lists of explanations require careful handling. In fact, such a queen may appear in the middle of an explanation list, and thus its deletion requires the re-computation of everything that is after it in the lists. In principle, to compute the new state after a queen move, we can truncate each list up to the element containing this queen, if any, and redo the queen positionings which are contained in the lists in successive positions. In this way, we have the state obtained after all the previous choices, and in the same sequence, except the choice related with the queen that we have moved. After this, we put the queen in the new position.

Thus we need to go through all the elements of all lists (and there are $O(n \times d \times n)$ of them), to truncate them, and then we need to redo at most n positionings, which take $O(n^3 \times d^3 \times l)$ each. Thus the time complexity of this kind of state change is $O(n^2 \times d + n^4 \times d^3 \times l) = O(n^4 \times d^3 \times l)$.

Note however that is not what is actually done in the implementation, since there are several ways to recompute the new lists more efficiently. However, this may give an idea of the added complexity given by the introduction of the preferences. In fact, in the hard case, there is no need to keep a list of causes

for the preference changes, since each cell can be subject to just one change: from allowed to not-allowed. Thus in the hard case we just need to remember at most one queen positioning as the cause for the not-allowed state of the cell. In the soft case, instead, we need to remember all the preference changes, and the queen causing each of them.

6 Implications

Each user-system interaction, that is, a queen positioning or retraction, has some consequence on the current and future states of the system. In fact, cell preferences could be changed because of such events, and this can also affect the global preference of complete assignments. Knowing the consequences of the next move over the future evolutions of the entire system can help the user in choosing such a move, in away that leads to an optimal solution quickly.

Our system shows two kinds of implications of a possible next move: the number of cells that would be deleted (that is, that would get a 0 preference level) if the move is performed, and an upper bound of the preference of a complete solution. This data are however (at least for now) not used by the system to suggest a best next move; this decision, based on heuristics, is left to the user.

Deletions. As mentioned above, knowing the consequences of all future choices may be helpful in understanding which is the most convenient choice to take next. One such consequence is the number of cells where it will not be possible to put any queen if we make a certain choice now. These are the cells that will get a preference value 0 if we position a queen in a certain position.

More precisely, for every cell without a queen, say C, we can compute the number of other cells which will be deleted if we put a queen in cell C. This is done by making a copy of the current state, by adding a queen in cell C to such a copy of the state, by running soft AC, and by counting the number of cells that, in the resulting state, get a 0 preference.

The complexity of the overall computation is therefore $O((n \times d) \times (n^3 \times d^3 \times l + n \times d))$, that is, $O(n^4 \times d^4 \times l + n^2 \times d^2) = O(n^4 \times d^4 \times l)$.

By clicking on the button "deletions" on the right, each cell shows the number of cells that will be deleted if we put a queen in that cell. Figure 4 shows the board after we have clicked on the deletion button, and considering the same setting as in Figure 2. For example, we can see that cell (3,3) contains the number 10. This means that, by putting a queen in cell (3,3), ten new cells would not be allowed any longer.

This information can guide the user in making the next choice. There are different possible heuristics that can be good to solve the problem. For example, the user may want to choose the cell which generates the smallest number of deleted cells, hoping to avoid a conflict. Or, like in the first-fail heuristics, the user may prefer the cell which generates the greatest number of deleted cells.

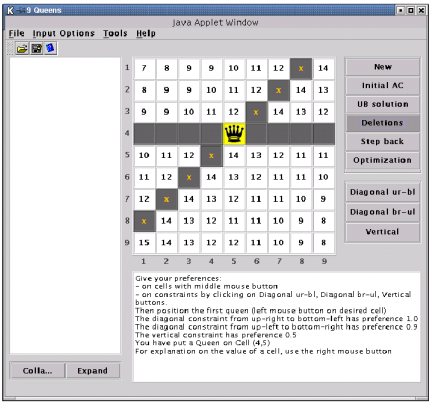


Fig. 4. Deletions: each cell shows the number of cells that would be deleted if a queen is positioned in this cell.

Upper bounds for solutions. When we have preferences, solutions are ranked by their preference level. Thus the aim is to obtain an optimal, or good enough, solution. The configurator should therefore guide the user towards good solutions.

This can be done by estimating the quality of the solutions which contain a certain queen. More precisely, for each cell, say C, we can compute an upper bound of the preference level of all solutions that contain a queen in cell C (and all the queens already positioned). This can be done, for example, by making a copy of the current state, positioning a queen in cell C, enforcing soft AC, and then taking the minimum value among all preferences in the resulting state. Since, by making more choices, the preference values can only decrease, and since the preference of a solution is the minimal preference among all those in the final state, this value is certainly an upper bound to the value of all solutions that contain a queen in cell C (and all the previously positioned queens).

The complexity of this operation, for all cells, is therefore $O(((n \times d) \times (n^3 \times d^3 \times l)) + (n \times d))$, that is, $O(n^4 \times d^4 \times l)$.

Figure 5 shows the upper bounds computed for each cell, which can be seen by clicking on the "UB solution" button on the right. For example, cell (8,9) shows a value of 0.4. This means that, by putting a queen in this cell, we cannot hope to obtain solutions with preference values higher than 0.4.

Thus users will tend to choose cells with a higher upper bound, hoping to get better solutions. However, it is expected that the user will sometimes have to consider a trade-off between the possibility of better solutions and the risk of conflict (which is partly described by the number of deletions for that cell).

7 Getting an Optimal Solution

Once the user has positioned all the queens, he/she has obtained a complete configuration with a certain preference level (given, we recall, by the minimum

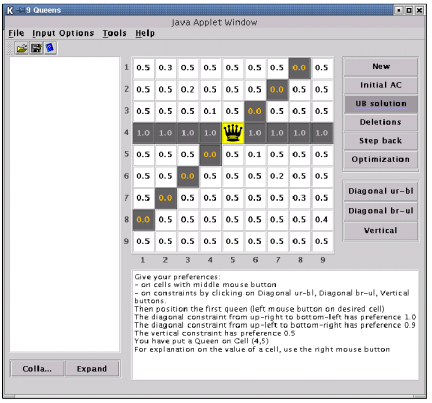


Fig. 5. Solution upper bounds: each cell shows an upper bound to the quality of the solutions which are compatible with a queen in this cell.

of all the preferences in all constraints and domains), which is shown in the lower panel. Figure 6 shows a complete solution with preference level 0.5.

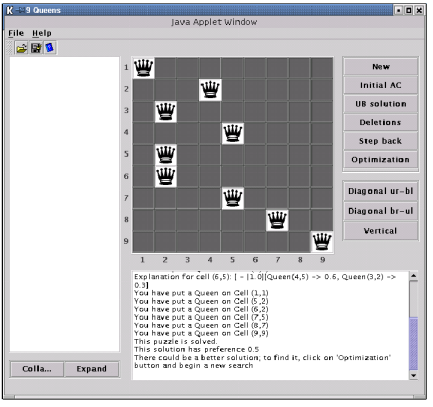


Fig. 6. A complete solution, with preference level 0.5.

But this configuration is not necessarily an optimal one. If the user is satisfied with the preference level of the obtained solution, he may stop the interaction with the configurator. However, the configurator should help him/her also in the case he/she wants to get a better solution. In our system, this is done as follows:

– the configurator checks (in a possibly incomplete way) whether the obtained solution is optimal or not;

– if the configurator cannot conclude that we have an optimal solution, it suggests that the user start again with an empty board and the initial preferences; however, it first lowers to 0 all those preferences which are lower than or equal to the preference of the last solution found. In this way, if a new solution will be found, it will certainly be better than the last one.

For example, in our system, the configurator checks the optimality of an obtained solution in an incomplete way as follows: it creates a copy of the initial state where all the preferences lower than, or equal to, the solution preference are set to 0, and it checks whether there is a variable whose domain elements all have preference 0. If so, it communicates to the user that the solution is an optimal one. Otherwise, it says that there could be better solutions. In Figure 6, we see a solution which is not optimal, while in Figure 7 we have an optimal solution.

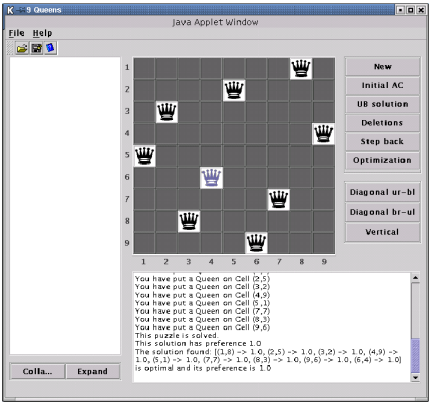


Fig. 7. An optimal solution, with preference level 1.

8 Conclusions and Future Work

We have investigated the scenario of configuration problems with preferences rather than just hard constraints, and we have shown that, even in this very general context, it is possible to build explanations and implications automatically by keeping track of the inferences that are made during constraint propagation. However, the process of recording the necessary information is more complex and expensive than in the hard case, since in general a sequence of previous choices has to be recorded for each possible choice, and not just one as in the hard case.

Many extensions of this work are possible. For example, in our system for now we can handle only fuzzy constraints. However, our theoretical setting can

allow for many generalizations, provided that some conditions are met for the correct application of constraint propagation algorithms [2].

Another extension involves the use of preferences not only to model the relation among the parts of a product (that is, the initial state of our system), but also to express the user desires during his/her interaction with the system. In fact, after some moves, a user may want to express some preferences that were not clear at the beginning of the interaction. This extension would allow also for the addition of new constraints. This would allow us to have an incremental model of the product to be configured, as suggested by [12,9].

Acknowledgments. This work was supported in part by Science Foundation Ireland under Grant 00/PI.1/C075. The first author is supported by a Principal Investigator Award from Science Foundation Ireland. Part of this work was done while Manuela Moretti was visiting the Cork Constraint Computation Centre, Cork, Ireland. The italian MIUR project NAPOLI and the ASI project ARISCOM have partially supported this work.

References

1. J. Amilhastre, H. Fargier, P. Marquis. Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artificial Intelligence 135 (1–2): 199–234*, 2000.
2. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
3. J. Bowen. Using dependency records to generate design coordination advice in a constraint-based approach to concurrent engineering. *Computers in Industry*, 33:191–199, 1997.
4. D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, pages 1131–1136. IEEE, 1993.
5. E. C. Freuder, C. Likitvivatanavong, R. J. Wallace. Explanation and implication for configuration problems. *IJCAI 2001 workshop on configuration*, pages 31–37, 2001.
6. F. Frayman. User-interaction requirements and its implications for efficient implementations of interactive constraint satisfaction systems. In *Proc. CP 2001 workshop on user-interaction in constraint satisfaction, Paphos, Cyprus*, 2001.
7. N. Jussien, V. Barichard. The palm system: explanation-based constraint programming. In *Proc. CP 2000 workshop on techniques for implementing constraint programming systems*, 2000.
8. S. Mittal, F. Frayman. Towards a generic model of configuration tasks. *Proc. 11th IJCAI*, 1989.
9. D. Sabin, E. C. Freuder. Configuration as Composite Constraint Satisfaction. *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*, AAAI Press, 1996.
10. D. Sabin, R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems and their Applications. Special issue on configuration*, pages 42–49, 1998.
11. T. Schiex. Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In *Proc. 8th Conf. of Uncertainty in AI*, pages 269–275, 1992.

12. T. Soininen, E. Gelle. Dynamic Constraint Satisfaction in Configuration. *Configuration Papers from the AAAI Workshop*, pp. 95–100, AAAI Technical Report WS-99-05, AAAI Press, 1999.
13. T. Soininen, J. Tiihonen, T. Mannisto, R. Sulonen. Towards a general ontology of configuration. *Artificial Intelligence for Engineering, Design and Manifacturing, 12:357–372*, 1998.
14. M. Stumptner. An overview of knowledge-based configuration. *AI Communication, 10 (2)*, 1997.
15. G. Suthers. Preferences for Model Selection in explanation. In *Proc. of IJCAI95*, Vol. 2, 1993
16. Edward P. K. Tsang. Foundations of Constraint Satisfaction. *Academic Press*, 1993.
17. R. J. Wallace, E. C. Freuder. Explanations for whom? In *Proc. CP 2001 Workshop on User-Interaction in Constraint Satisfaction*, 2001.

Constraint Processing Offers Improved Expressiveness and Inference for Interactive Expert Systems

James Bowen

Department of Computer Science
UCC, Cork, Ireland
j.bowen@cs.ucc.ie

Abstract. Expert systems constitute one of the most successful application areas for Artificial Intelligence techniques; they have been deployed in many areas of industry and commerce. If-then rules are the core knowledge representation technology in currently deployed systems. However, if we replace rules by constraints, we get improved expressiveness in knowledge representation and richer inference.

1 Introduction

An expert system [5] is a program which mimics human problem-solvers, in several senses: it contains an explicit representation of the knowledge which is used by humans who are expert at solving tasks in some problem domain; its reasoning process mimics that of the human experts; it elicits data from its users in a fashion similar to that used by a human expert who questions his client during a consultation; it can explain its answers to its users in the same way as a human expert can explain his conclusions to his clients. Expert systems constitute one of the most successful application areas for Artificial Intelligence (AI) techniques - they have been assimilated into the mainstream where they are widely deployed, frequently in concert with non-AI software technologies [8,10,11,12].

In currently deployed expert systems, knowledge about the problem domain is represented in the form of if-then rules. There are two kinds of rules [5]: imperative rules, where the consequent of a rule specifies some operation(s) to be performed if the antecedent is satisfied; and declarative rules, where the consequent of a rule specifies some fact which is implied by the truth of the antecedent.

In this paper, it is argued that, if constraints are used instead of declarative rules, improved functionality is achieved: constraints provide a richer expressive medium than rules; constraints support a richer form of inference than rules.

The rest of this paper is organised as follows. First, a review of constraint-based reasoning is given, with particular emphasis on interactive processing. Then the notion of using constraints to build interactive expert systems is explored, with particular emphasis on mixed-initiative information acquisition during interactive processing. Following this, constraint programming is related, in

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 93–108, 2003.

a model-theoretic fashion, to Predicate Calculus, with particular emphasis on a treatment of constraint satisfaction as model completion. The approach is illustrated by means of an example expert system for selecting a laptop to meet a user's needs; in this discussion, emphasis is placed on the expressiveness of constraint-based knowledge representation. Then, it is explained that constraint propagation provides a richer form of inference than that supported by rule-based systems, because it supports *modus tollens* as well as *modus ponens*.

2 Constraint-Based Reasoning

In the literature, several different definitions are given for constraint networks, with varying degrees of formality. However, they may all be regarded as variations of the following theme:

Definition 1, Constraint Network:
A constraint network is a triple $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$. $\mathbf{D}$ is a finite set of $p > 0$ domains, the union of whose members forms a universe of discourse, $\mathcal{U}$. $\mathbf{X}$ is a finite tuple of $q > 0$ non-recurring parameters. $\mathbf{C}$ is a finite set of $r \geq q$ constraints. In each constraint $C_k(T_k) \in \mathbf{C}$, T_k is a sub-tuple of $\mathbf{X}$, of arity a_k; $C_k(T_k)$ is a relation, a subset of the a_k-ary Cartesian product $\mathcal{U}^{a_k}$. In $\mathbf{C}$ there are q unary constraints of the form $C_k(X_j) = D_i$, one for each parameter X_j in $\mathbf{X}$, restricting it to range over some domain $D_i \in \mathbf{D}$ which is called the *domain of* X_j.

The overall network constitutes an intensional specification of the simultaneous value assignments that can be assumed by the parameters. In other words, the network constitutes an intensional specification of a q-ary relation on $\mathcal{U}^q$, in which each q-tuple provides an ordered group of values, each value being an assignment for the corresponding parameter in $\mathbf{X}$. This implicitly specified relation is called the *intent* of the network:

Definition 2, The Intent of a Constraint Network:
The intent of a constraint network $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$ is
$$\mathit{II}^{\mathbf{D},\mathbf{X},\mathbf{C}} = \overline{C_1}(\mathbf{X}) \cap ... \cap \overline{C_r}(\mathbf{X}),$$
where, for each constraint $C_k(T_k) \in \mathbf{C}$, $\overline{C_k}(\mathbf{X})$ is its cylindrical extension [4] in $\mathcal{U}^q$.

Note that the definitions given above admit infinite domains, implying that the universe of discourse $\mathcal{U}$ may be infinite and that the constraint relations may be infinite, thereby making it possible that the intent of a network may be an infinite relation. In finite-domain networks, the domains and constraints are usually specified extensionally; in networks which contain infinite domains and relations, these domains and relations must, necessarily, be specified intensionally.

Many different forms of constraint satisfaction problem (CSP) have been distinguished. The forms of CSP encountered most frequently in the literature can be defined in terms of a network intent as follows:

Definition 3, The Decision CSP:
Given a network $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$, decide whether $\Pi^{\mathbf{D},\mathbf{X},\mathbf{C}}$ is non-empty.
Definition 4, The Exemplification CSP:
Given a network $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$: return nil if $\Pi^{\mathbf{D},\mathbf{X},\mathbf{C}}$ is empty; otherwise, return some tuple from $\Pi^{\mathbf{D},\mathbf{X},\mathbf{C}}$.
Definition 5, The Enumeration CSP:
Given a network $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$, return $\Pi^{\mathbf{D},\mathbf{X},\mathbf{C}}$.

The Exemplification CSP is the one most commonly addressed by algorithm researchers. Solving the Exemplification CSP is frequently used as a surrogate for solving the Decision CSP. The Enumeration CSP is rarely addressed; it is sometimes solved analytically (in the case of infinite-domain problems) or, in the case of finite-domain problems, by finding all solutions to the Exemplification CSP.

2.1 Interactivity

Most research on constraint processing has focus on autonomous problem-solving by machines. Many real-world applications, however, require interactive decision support rather than automated problem-solving. Consequently, recent constraints research has involved interactive processing. In [3] , for example, Exemplification CSPs are solved by an interactive version of the MAC algorithm, in which search moves are made by a user assigning values to network parameters, while the machine performs constraint propagation (arc-consistency) after each move by the user.

In [2], a more general form of interactive processing was introduced. This was based on a new form of CSP called the Specialization CSP:

Definition 6, The Specialization CSP:
Given a network $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$: return nil if $\Pi^{\mathbf{D},\mathbf{X},\mathbf{C}}$ is empty; otherwise, return (i) a set $\mathbf{A}$ of additional constraints such that $\Pi^{\mathbf{D},\mathbf{X},\mathbf{C}\cup\mathbf{A}}$ contains exactly one tuple and (ii) this tuple.

If the given network has an empty intent, the task is, as in the Exemplification CSP, to identify that fact. If the given network has a non-empty intent, the task includes, as in the Exemplification CSP, finding a group of consistent assignments for the network parameters. However, whereas solving the Exemplification CSP involves searching through the space of possible parameter assignments, solving the Specialization CSP involves searching through the space of possible constraints. When the Exemplification CSP is being solved interactively, the user directs the search by inputting parameter assignments (which, of course, are constraints of a restricted form). By contrast, when the Specialization CSP is solved interactively, the user can input arbitrary constraints.

In fact, applications based on the Specialization CSP have been around for quite a few years [13, 14]. Related application work includes [15, 16, 17].

3 Constraint-Based Interactive Expert Systems

While not all expert systems interact with human users (process control expert systems, for example, interact with process sensors and actuators), most do. The general scenario is that the expert system should advise the user by determining the value for certain crucial characteristics of a situation affecting the user. This form of decision-support can be accomodated in constraint-based reasoning if we define a new form of CSP, as follows.

Definition 7, The Targeted Specialization CSP:
Given a network $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$ and a sub-tuple $\mathbf{T}$ of $\mathbf{X}$: return nil if $\Pi^{\mathbf{D},\mathbf{X},\mathbf{C}}$ is empty; otherwise return (i) a set $\mathbf{A}$ of additional constraints such that $\Pi_{\mathbf{T}}^{\mathbf{D},\mathbf{X},\mathbf{C}\cup\mathbf{A}}$ contains exactly one tuple and (ii) this tuple.

An expression of the form $\Pi_{\mathbf{T}}^{\mathbf{D},\mathbf{X},\mathbf{C}}$ denotes the *projection* of a network intent onto the sub-tuple of the network parameters that are in $\mathbf{T}$. This projection is a relation in which each tuple provides an ordered group of values which, if they are assumed by the corresponding parameters in $\mathbf{T}$, will satisfy the constraints. Thus, given a network with a non-empty intent, the task posed by the Targeted Specialization CSP is to find a group of consistent values for the targeted parameters. As in the case of the Specialization CSP, the task of interactively solving the Targeted Specialization CSP involves receiving information from the user and propagating it throughout the network. As in the Specialization CSP, the information from the user may be arbitrary constraints.

3.1 Knowledge Representation in a Constraint-Based Expert System

Generic Domain Knowledge. In a constraint-based expert system, the domain knowledge embedded in the system consists of the constraints $\mathbf{C}$ in a network $\langle \mathbf{D}, \mathbf{X}, \mathbf{C} \rangle$. The intent of this network will be non-empty – otherwise, the constraints would be just an inconsistent set of assertions, rather than a coherent body of expertise about a class of situations that affect the users of the expert system.

Instance-specific Knowledge. Information about the problem instance affecting a particular user is represented by a set, $\mathbf{A}$, of additional constraints which are received from the user. A complete description of a problem instance consists, therefore, of $\mathbf{C} \cup \mathbf{A}$ – all constraints, both those representing the generic domain knowledge, $\mathbf{C}$, and those representing the information about the specific problem instance, $\mathbf{A}$, must be satisfied.

The advice from the expert system to the user consists of the values for the target parameters, $\mathbf{T}$, that are admitted by the constraints in $\mathbf{C} \cup \mathbf{A}$ – that is, the advice consists of the single tuple in $\Pi_{\mathbf{T}}^{\mathbf{D},\mathbf{X},\mathbf{C}\cup\mathbf{A}}$.

3.2 Mixed-Initiative Acquisition of Information

Mixed-initiative interaction is a desirable feature of intelligent programs. Achieving it is the subject of ongoing research – a workshop on the topic was organized at AAAI-99 [1].

In expert systems, supporting mixed-initiative interaction means enabling an appropriate balance between machine-generated questions and user-volunteered facts. Backward-chaining through declarative rules involves machine-driven acquisition of information about a specific problem instance: the system's hypotheses comprise the roots of a set of interlaced trees whose leaves represent possible data points; when considering a hypostheis, the system backward-chains from the root to its leaves, asking questions about these; when a root is found, all of whose leaf nodes receive positive answers from the user, *modus ponens* inference causes the expert system to derive the truth of the hypothesis corresponding to the root. Forward-chaining through declarative rules involves matching user-volunteered facts with leaf nodes and using *modus ponens* to derive a conclusion corresponding to some root. Mixed-initiative interaction with expert systems based on declarative rules involves some mixture of backward- and forward-chaining.

The interactive constraint-based systems which have been reported in the literature involve users volunteering information – in [3], for example, the user volunteers parameter assignments. This, and the fact that inference in constraint-based systems is called constraint propagation, may seem to imply that interaction in constraint-based systems must be user-driven. However, this is not the case. It is possible to use backward-chaining in constraint networks – given a target parameter whose value it must determine, the system can question the user about the immediately neighbouring parameters, or about their neighbours, and so on. However, the very richness of interconnectivity in constraint networks (including, for example, cyclical interdependence of parameters) means that it is much more difficult to decide which parameters to ask the user about – optimal question-generation in interactive constraint-based expert systems is the subject of ongoing research by the author of this paper [18]. While optimal question-generation is still an open research topic, backward-chaining is already being done – a system which questions the user about the parameters neighbouring a target parameter will be illustrated below. Given that both backward- and forward-chaining is possible in interactive constraint-based systems, mixed-initiative interaction simply involves some mixture of the two – the system illustrated below supports mixed-initiative interaction.

4 Constraints and First-Order Predicate Calculus

Depending on the nature of the symbols that can appear in their antecedents and consequents, declarative if-then rules are implication statements in either propositional or predicate calculus. Constraints offer a much richer expressiveness than rules because constraints can provide the expressive power of the full first-order Predicate Calculus (PC).

When a first-order language, $\mathcal{L}$, is used to discuss some universe of discourse, $\mathcal{U}$, the constant symbols of $\mathcal{L}$ denote elements in $\mathcal{U}$, the function and relation symbols of $\mathcal{L}$ denote functions and relations over $\mathcal{U}$, while the logic variables are quantified over $\mathcal{U}$. Note that a universe of discourse, being a set of entities, may be either a finite or an infinite set. If a universe of discourse is finite, the Predicate Calculus is not essential – the Propositional Calculus provides sufficient reasoning power.

However, there are other reasons for choosing a notation than just the power of its associated reasoning system – otherwise, there would have been no need for programming language developers to progress beyond machine code. In expert systems, facility of user-input and perspicuity of system-output are just as important as speed of system-internal inference. Thus, even in applications having finite universes of discourse, Predicate Calculus may be preferred to Propositional Calculus, simply because the greater expressive flexibility offered by the Predicate Calculus enables knowledge to be represented in a more naturalistic fashion – just as the father of a large family finds it easier to say that ""all my children have left school" than to say that "Al, Bob, Cait, Dora, ..., Xavier, Yuri and Zeev have left school".

The discussion that follows will illustrate how, in applications having finite universes of discourse, constraint-based technology offers both programmers and users of expert systems the expressive flexibility of the full Predicate Calculus. It will also show how, in applications having infinite universes of discourse, constraint-based technology offers almost the same flexibility – the only limitation being that quantifiers must be relativized to finite subsets of an infinite universe of discourse[1].

4.1 Constraint Satisfaction as Model Completion

In [9], the current author first discussed the relationship between constraint satisfaction and finding models in Predicate Calculus – Mackworth, in [7], discussed a similar notion, the relationship between finite constraint satisfaction and finding models in the Propositional Calculus.

The predicate calculus approach can be described as follows. Consider some application domain for an expert system. The entities in this application domain comprise a universe of discourse $\mathcal{U}$. To discuss $\mathcal{U}$, a first-order language, $\mathcal{L}$, is defined, with an associated partial model, $\mathcal{M}_p$, which contains an interpretation, in terms of $\mathcal{U}$, for all function and predicate symbols of $\mathcal{L}$ and most, but not all, of its constant symbols.

This language $\mathcal{L}$ can then be used to specify constraint networks, networks in which the set of parameters will be the set of un-interpreted constant symbols of $\mathcal{L}$ and in which the set of constraints will be a set $\mathcal{S}$ of sentences of $\mathcal{L}$. Any sentence of $\mathcal{L}$, provided it references at least one of the un-interpreted

[1] Theoretically, this limitation is not necessary – quantifiers can range of over infinite universes of discourse. In practice, however, the constraint processing algorithms that have been developed so far can process only relativized quantifiers.

constant symbols of $\mathcal{L}$, can be a constraint – even one containing arbitrarily nested quantification[2]. From this perspective, it can be seen that CSPs become model-completion problems. For example, the Exemplification CSP becomes the task of computing, if one exists, a total model $\mathcal{M}$ of $\mathcal{L}$ such that $\mathcal{M} \supset \mathcal{M}_p$ and such that all the sentences in $\mathcal{S}$ are true under $\mathcal{M}$, or, if no such model exists, of returning the information that this is so. Similarly, if any model of $\mathcal{L}$ exists which subsumes $\mathcal{M}_p$ and entails $\mathcal{S}$, the Targeted Specialization CSP becomes the task of computing a set $\mathcal{S}_a$ of additional sentences such that every model $\mathcal{M}$, $\mathcal{M} \supset \mathcal{M}_p \wedge \mathcal{M} \models (\mathcal{S} \cup \mathcal{S}_a)$, has the same set of interpretations for the constant symbols in **T**.

This approach to relating constraints and predicate calculus is quite different from the CLP paradigm [6]. In CLP, a constraint network is treated as a goal (a theorem to be proven); the clauses in a CLP program are not part of the network – they define the semantics of application-specific relation symbols in the query. Because a CLP network is treated as a query, network parameters correspond to existentially quantified logic variables while constraints are restricted to a very limited subset of PC utterances - arbitrary nesting of quantification, for example, is prohibited in CLP. A key advantage of the model-completion approach over CLP is that, since network parameters correspond to constant symbols (albeit symbols in search of an interpretation) rather than logic variables, the parameters can be referenced in multiple sentences; this is because usage of a constant symbol, unlike usage of a logic variable, is not restricted to the lexical scope of a single quantifier symbol within a single sentence. Thus, a parameter which is referenced in sentences that form part of a generic network for an application domain, can also be referenced in sentences input by the user to provide information about his specific problem instance. Thus, this approach is better than CLP at supporting user-interaction.

The greater expressiveness of the model-completion approach (the fact that, subject to the caveats given earlier, arbitrary sentences of $\mathcal{L}$ can be used as constraints) simply reinforces the benefits which are derived from the treatment of parameters as constant symbols. It does, however, mean that, provided the model completion approach is used, constraint-based reasoning offers greater expressiveness for knowledge representation than rule-based reasoning.

4.2 Example

To illustrate the approach, consider a very simple expert system for selecting, from a range of laptop computers, one which will best meet the needs of a user.

Language and Universe of Discourse. The universe of discourse comprises the range of available laptops and the real numbers, the latter being present

[2] Of course, constraint processing algorithms, in particular the type of consistency processing algorithm that is discussed later in this paper, will be more likely to achieve inferential completeness if either the universe of discourse is finite or all quantifiers are relativized to finite subsets of the universe.

because the user's requirements (regarding CPU speed, RAM and disk space, and machine weight), as well as the corresponding laptop characteristics, and their prices, are numbers. Inspired by a laptop range manufactured by a certain well-known firm, our universe of discourse $\mathcal{U}$ is $\Re \cup \{c810, c410, c210, c110\}$, where $\Re$ contains the real numbers while c810 etc. are laptops. Since $\mathcal{U}$ subsumes $\Re$, our language $\mathcal{L}$ contains symbols to discuss the members of $\Re$ - that is, $\mathcal{L}$ contains numerals (constant symbols interpreted to denote members of $\Re$) as well as predicate and function symbols, such as `>=`, `*`, etc., which are interpreted to denote standard relations and functions over $\Re$. Since $\mathcal{U}$ contains the laptops, $\mathcal{L}$ must also include some symbols to discuss these: a unary predicate symbol, `laptop`, whose extension is the set of laptops; some unary function symbols, `speed`, `ramCapacity`, `diskCapacity`, `weight` and `price`, which map from the laptops onto the numbers that are the obvious laptop characteristics; and constant symbols, such as `c110`, `c210`, etc., which are interpreted to denote the obvious laptops. The symbols listed above have their interpretations defined in the partial model $\mathcal{M}_p$ of $\mathcal{L}$. The symbols which are not interpreted in $\mathcal{M}_p$ are the constant symbols `minSpeed`, `minRAM`, `minDisk` and `maxWeight` (which, when they are finally interpreted, should denote the obvious user requirements), and `chosenModel` (which should denote the appropriate laptop).

Generic Domain Knowledge. The generic domain knowledge for our expert system can now be expressed as a set of sentence in $\mathcal{L}$. For example, the need for the chosen laptop to satisfy the user's computing requirements could be expressed as a set of three ground atomic sentences

```
speed(chosenModel) >= minSpeed.
ramCapacity(chosenModel) >= minRAM.
diskCapacity(chosenModel) >= minDisk.
```

while the weight requirement could be expressed as

```
weight(chosenModel) =< maxWeight.
```

Suppose we wish to specify that the cheapest laptop which meets the user's computing requirements must be chosen. Given that the only money-related symbol we have is the function `price`, how do we represent the notion of "cheapest"? A laptop is "cheapest" if there is no other laptop whose price is less. Thus, our specification above can be expressed in $\mathcal{L}$ as:

```
not exists X :
  ( laptop(X) and
    price(X) =< price(chosenModel) and
    speed(X)>=minSpeed and
    ramCapacity(X)>=minRAM and
    diskCapacity(X)>=minDisk ).
```

This PC sentence can be paraphrased in English as "there should not exist any laptop which is cheaper than the chosen one and which also satisfies the computing requirements".

Note that the universe of discourse for this application is infinite – it subsumes $\Re$. However, the quantification in the constraint just given is relativized to a finite subset of the universe – to the set of laptops. This relativization means that the sentence is equivalent to a ground sentence, one in which all the available laptops are referenced by name. While this equivalance is what makes the sentence tractable[3] as a constraint, the freedom to use a quantifier in expressing the constraint makes for easier knowledge management – just like the father mentioned earlier, who often finds it easier to refer to his large family collectively instead of listing them individually by name.

In this simple application domain, a constraint using heavily nested quantification would be very contrived. However, if we extended the scope of the application to include, say, power supply systems around the world, then we might need a certain degree of quantifier nesting – in a constraint referring to worldwide rechargability, for example, one quantifier might range over the laptops while another ranged over the power supply systems used in various parts of the world.

Partial Model. An implementation of the model-completion approach to constraint-based reasoning has been built by the author of this paper. An expert system for laptop selection built on top of this implementation consists of the above five sentences plus some statements which specify (a) the application-specific language $\mathcal{L}$ to which the sentences belong and (b) a partial model $\mathcal{M}_p$ for $\mathcal{L}$. It is assumed that every universe of discourse subsumes $\Re$, so the implementation provides a set of pre-defined symbols for discussing $\Re$ and a pre-defined interpretation for all these symbols. Thus, in specifying a universe, a language and a partial model for an application, all that needs to be done is to specify the application-specific material. In an expert system for laptop selection, the statements needed to provide the application-specific detail for $\mathcal{L}$ and $\mathcal{M}_p$ are as follows:

```
domain laptop =::= {c810,c410,c210,c110}.
function speed(laptop) -> number
    =::= { c810->1.2, c410->0.9, c210->0.6, c110->0.57 }.
function ramCapacity(laptop) -> integer
    =::= { c810->1024, c410->1024, c210->256, c110->256 }.
function diskCapacity(laptop) -> integer
    =::= { c810->68, c410->40, c210->20, c110->15 }.
function weight(laptop) -> number
    =::= { c810->3.2, c410->2.9, c210->3.1, c110->2.7 }.
function price(laptop) -> integer
    =::= { c810->3299, c410->2599, c210->1799, c110->1439 }.
chosenModel : laptop.
minSpeed : positive number.
minRAM : positive number.
```

[3] To currently developed algorithms, at least.

```
minDisk : positive number.
maxWeight : positive number.
```

These statements can be explained as follows. An application-specific universe of discourse consists of the union of $\Re$ with the application-specific domains. The example expert system needs only one such domain, containing the laptops; it is declared in the first statement above. This statement also introduces several symbols into $\mathcal{L}$: a unary predicate symbol, `laptop`, and four constant symbols, `c810`, `c410`, `c210` and `c110`. The statement implicitly defines the interpretation of these symbols: the constant symbols are interpreted as the obvious members of $\mathcal{U}$ while the unary predicate symbol is interpreted as denoting the set of laptops. Each of the subsequent five statements introduces an application-specific unary function symbol into $\mathcal{L}$ and defines its interpretation. Each of the final five statements introduces a constant symbol into $\mathcal{L}$. These symbols are not interpreted, although the space of possible interpretations for each symbol is restricted to a subset of $\mathcal{U}$; `chosenModel`, for example, must denote a laptop while `minSpeed` must denote a member of $\Re^+$.

Network. The constraint network defined by the foregoing contains five constraints (one for each of the five sentences of generic domain knowledge specified above) and five parameters (one for each constant symbol that is un-interpreted in $\mathcal{M}_p$).

The parameters are: `chosenModel`, `minSpeed`, `minRAM`, `minDisk` and `maxWeight`. The domain of the parameter `chosenModel` is {c810, c410, c210, c110}, while each of the other parameters has the domain $\{X|X > 0\}$.

Four of the constraints are binary – those corresponding to the sentences of generic domain knowledge which specify that the chosen model must satisfy the user's computing and weight requirements. The fifth constraint, that corresponding to the sentence of generic domain knowledge which specifies that the cheapest satisfactory laptop must be chosen, involves five parameters.

The constraint relations corresponding to the sentences of generic domain knowledge can be computed from the interpretations, in $\mathcal{M}_p$, for the interpreted symbols in each sentence. For example, consider the sentence

```
speed(chosenModel) >= minSpeed.
```

This is a binary constraint – it contains two constant symbols, `chosenModel` and `minSpeed`, and, since neither of them is interpreted in $\mathcal{M}_p$, they are both parameters in the constraint network. The constraint relation which corresponds to this sentence can be computed from the pre-defined interpretation for the standard predicate symbol `>=` and the interpretation of the application-specific function symbol `speed`. Remember that the interpretation of the function symbol `speed` was defined above as follows:

```
function speed(laptop) -> number
=::= { c810->1.2, c410->0.9, c210->0.6, c110->0.57 }.
```

Thus, the sentence `speed(chosenModel) >= minSpeed` means that if `chosenModel` had the value c810, then 1.2 should be greater than or equal to the value of `minSpeed`; similarly, if `chosenModel` had the value c410, then 0.9 should be greater than or equal to the value of `minSpeed`; and so on. Thus, the sentence `speed(chosenModel) >= minSpeed` corresponds to the following constraint relation on the parameter pair ⟨`chosenModel`, `minSpeed`⟩:

$$\{\langle X, Y\rangle | (X = c810 \wedge Y \leq 1.20) \vee (X = c410 \wedge Y \leq 0.90) \vee (X = c210 \wedge Y \leq 0.60) \vee (X = c110 \wedge Y \leq 0.57)\}$$

How are quantified sentences handled? Instead of considering the quantified sentence which specifies that the cheapest satisfactory laptop must be chosen, we will consider the following shorter sentence, which specifies that the maximum possible amount of RAM is needed:

```
not exists X : (laptop(X) and ramCapacity(X)>minRAM).
```

Its meaning is computed as follows. First, the negation is moved inside the quantifier, so that the sentence becomes

```
all X : (laptop(X) implies ramCapacity(X) =< minRAM).
```

Then, because the `laptop` domain is finite, the effect of this sentence can be achieved by iterating over the domain – in essence, treating the sentence as if it were

```
ramCapacity(c810) =< minRAM and ramCapacity(c410) =< minRAM and
ramCapacity(c210) =< minRAM and ramCapacity(c110) =< minRAM.
```

Remember that the function symbol `ramCapacity` was defined earlier as follows:

```
function ramCapacity(laptop) -> number
    =::= { c810->1024, c410->1024, c210->256, c110->256 }.
```

This means that the constraint relation corresponding to the above sentence is the following unary constraint on the parameter `minRAM`: $\{X | X \geq 1024\}$.

At this stage, it may be appropriate to consider quantification over infinite domains. Consider the sentence:

```
not exists X : X > minRAM.
```

The system being described here can accept this sentence, although it will not be able to do anything useful with it. Recognizing that the quantifier is not relativized to a finite subset of the universe of discourse, it will not attempt to use iteration; instead, it will treat the above sentence as the following unary constraint on the parameter `minRAM`: $\{X | \neg\exists Y(Y > X)\}$. As we shall see below, the basic operation of the system involves arc consistency processing, using an approach in which term re-writing is used to process to infinite constraint relations. The basic difficulty with the above sentence lies in the fact that nothing in the set of term-rewriting rules used (so far) can do anything useful with the

above intensional formula – therefore, the system cannot guarantee to achieve total arc consistency in a network containing such a constraint relation[4]. Possibly, however, one could imagine an augmented set of rules which could do more. Nevertheless, it will always be possible, given any set of re-writing rules, to devise a set of constraints which are beyond the inferential competence of the rules.

Constraint Processing. The basic operation of the run-time system consists of applying (hyper-)arc consistency to the constraints in the network, constraints whose relations may be either finite (and represented extensionally) or infinite (and represented intensionally). Finite domains are pruned by removal of inconsistent values. An infinite domain is pruned in two stages: first, a conjunct is added to the intensional formula that specified the domain before pruning; then, term rewriting is used to simplify the extended intensional formula.

Consider, for example, the following constraint relation on the parameter pair ⟨`chosenModel`, `minSpeed`⟩:

$$\{\langle X, Y\rangle | (X = c810 \wedge Y \leq 1.20) \vee (X = c410 \wedge Y \leq 0.90) \vee (X = c210 \wedge Y \leq 0.60) \vee (X = c110 \wedge Y \leq 0.57)\}$$

Initially, the domain of `chosenModel` is {c810, c410, c210, c110} and that of `minSpeed` is $\{X | X > 0\}$. When we use arc-consistency on the arcs of the above constraint, the domain of `chosenModel` is unchanged, but the intensional formula for the domain of `minSpeed` undergoes the following changes. First it becomes

$$\{X | X > 0 \wedge (X \leq 1.2 \vee X \leq 0.9 \vee X \leq 0.6 \vee X \leq 0.57)\}$$

in which the intensional formula has been expanded, to include a conjoined expression which captures the impact of the constraint on the set of possible values for `minSpeed`. Term rewriting then changes this formula, initially reducing it to

$$\{X | X > 0 \wedge (X \leq 1.2)\}$$

and then rewriting it further to

$$\{X | 0 < X \leq 1.2\}.$$

When the constraints representing the generic domain knowledge have all been processed (as they would be before any user input were accepted), the domains of `minRAM`, `minDisk` and `maxWeight` would have been reduced to $\{X | 0 < X =< 1024\}$, $\{X | 0 < X =< 68\}$ and $\{X | X >= 2.7\}$, respectively.

[4] Strictly speaking, of course, the system can achieve "'total" arc consistency, by propagating this intensional formula through all relevant parts of the network. In practice, of course, *useful* arc consistency involves producing simplified intensional formulae – for example, if a formula involves a contradiction, useful arc consistency would involve detecting this and inferring that the parameter whose domain is described by the formula is the empty set.

Mixed-initiative Interaction. The approach supports mixed-initiative interaction. A user interacting with this expert system can volunteer instance-specific data by asserting appropriate sentences in $\mathcal{L}$ – each such sentence must, of course, refer to at least one of the un-interpreted symbols. For example, if the user knows that he needs a CPU speed of at least 0.7 gigahertz, he could assert

```
minSpeed = 0.7.
```

Similarly, if he wants the largest possible amount of RAM, he could assert

```
not exists X : (laptop(X) and ramCapacity(X)>minRAM).
```

Each such sentence extends the constraint network defined in the expert system, by adding a new constraint. After each new constraint is asserted by the user, its arcs are entered into the queue of arcs maintained by the hyper-arc consistency algorithm. As usual in arc consistency algorithms, if the domain of any parameter is reduced by any arc of this new constraint, the other constraint arcs referencing this parameter are appended to the queue. The algorithm reaches quiescence when the queue becomes empty again, at which point the user can volunteer another assertion or retract one of his previous assertions – dependency records are maintained to facilitate such retractions. (The dependency records are also used to generate explanations when requested by the user.)

Suppose, for example, the user asserted `minSpeed=0.7`. Constraint propagation would result in the domain of `minSpeed` being restricted to $\{0.7\}$. Since this parameter is also referenced in `speed(chosenModel) >= minSpeed`, the arcs of this constraint would be appended to the queue. The effect of this is that the domain of `chosenModel` is reduced to $\{c810, c410\}$, because the other laptops are not fast enough. This change would then activate the other constraints which reference this parameter. By the time that quiescence is reached, the domain for `maxWeight` would have been reduced from $\{X|X >= 2.7\}$ to $\{X|X >= 2.9\}$. If the user were then to assert `not exists X : ( laptop(X) and ramCapacity(X) > minRAM )`, that would reduce the domain of `minRAM` from $\{X|0 < X =< 1024\}$ to $\{1024\}$ but would not reduce further the domain of any other parameter.

Instead of volunteering instance-specific data, the user could hand the initiative over to the machine by specifying that he wishes it to help him determine an appropriate value for `chosenModel`. In doing so, he would be creating a Targeted Specialization CSP in which **T**, the set of targeted parameters, would be {`chosenModel`}. The machine would then ask the user for information about the other parameters in the network, each reply being treated as another constraint to be added to the network: a user's reply need not specify a value for the parameter which was the subject of the machine's question; it could, instead, be an arbitrary sentence involving the parameter – for example, a user asked about the required disk space could reply that he wants ten times as much disk space as RAM, by entering the sentence `minDisk = 10 * (minRAM/1000)`, division by 1000 being involved because RAM is specified in megabytes while disk space is specified in gigabytes.

5 Improved Inference

It has been shown how the use of constraints supports richer knowledge representation than that available to rule-based expert systems – constraints extend both the range of domain expertise that can be expressed and the range of instance-specific data that users of an interactive expert system can provide. However, it should also be pointed out that constraints also support richer inference.

Inference in rule-based expert systems is based on *modus ponens*. That is, we can have inferences of the form

$$(A \Rightarrow B, A) \quad \vdash \quad B.$$

Constraint propagation, however, subsumes both *modus ponens* and *modus tollens*. That is, it can also make inferences of the form

$$(A \Rightarrow B, \neg B) \quad \vdash \quad \neg A.$$

Suppose, for example, that we have an expert system in which there are two parameters, `length` and `width`, that must assume values from $\Re^+$. Suppose that one piece of domain knowledge is `length >= 1000 implies width >= 500`. Suppose that, while backward-chaining through some other piece of domain knowledge, the expert system asks the user for the value of `width` and receives the response that `width = 400`. A rule-based expert system would not be able to deduce from this reply, and from the domain knowledge just given, that `length` must be less than `1000`.

Now suppose that the expert system is constraint-based rather than rule-based. Initially, the domains of `length` and `width` would both be $\{X|X > 0\}$. According to the standard semantics of material implication in logic, the constraint relation corresponding to `length >= 1000 implies width >= 500` is the union of two infinite sets of tuples, represented by the two disjuncts in this intensional formula:

$$\{\langle X, Y\rangle | X < 1000 \vee (X \geq 1000 \wedge Y \geq 500)\}.$$

The first of these two sets corresponds to the case where the antecedent of the implication is not satisfied, the second to the case where it is. When the user replies that `width = 400`, this is treated as a unary constraint whose relation is $\{400\}$. Propagating this constraint would, first, reduce the domain of `width` from $\{X|X > 0\}$ to $\{400\}$ and then activate the constraint `length >= 1000 implies width >= 500`. The fact that the domain of `width` is now $\{400\}$ means that the second set in the union comprising the semantics of the constraint is irrelevant. Thus, arc consistency means that the first set can be projected onto the domain of `length`. Thus, the domain of `length` is reduced from $\{X|X > 0\}$ to $\{X|0 < X < 1000\}$. In other words, applying arc consistency to the constraint relation $\{\langle X, Y\rangle | X < 1000 \vee (X \geq 1000 \wedge Y \geq 500)\}$ on the parameters $\langle$ `length`, `width` $\rangle$, in a context where the domain of `width` is $\{400\}$, achieves the same effect as applying *modus tollens* to the following premises: `length >= 1000 implies width >= 500` and `width=400`.

6 Conclusions

The standard way of relating constraint-based reasoning to the Predicate Calculus (PC) is Constraint Logic Programming (CLP), in which constraints are integrated into a proof-theoretic approach to logic. This paper advocates an alternative, model-theoretic approach. In this approach, the task of solving a constraint satisfaction problem becomes that of completing a partial model for a first-order PC language. It has been shown that, in this context, constraint propagation algorithms provide an inference capability which subsumes the effects of both *modus ponens* and *modus tollens*. Thus, if constraints replace declarative rules in the construction of expert systems, two benefits follow: more expressive knowledge representation and more powerful inference.

References

1. AAAI, 1999, *Proc. AAAI-99 Workshop on Mixed-Initiative Intelligence.*
2. Bowen J, 2001, "The (Minimal) Specialization CSP: A basis for Generalized Interactive Constraint Processing", *Proc. CP-2001 Workshop on User-Interaction in Constraint Processing.*
3. Freuder E, Likitvivatanavong C and Wallace R, 2000, "A Case Study in Explanation and Implication", *Proc. CP-2000 Workshop on Analysis and Vizualization of Constraint Programs and Solvers.*
4. Friedman G and Leondes C, 1969, "Constraint Theory, Part I: Fundamentals", *IEEE Transactions on Systems Science and Cybernetics*, ssc-5, 1, 48–56.
5. Jackson P, 1999, *Introduction to Expert Systems*, 3 rd. Edition, Addison Wesley Longman.
6. Jaffar J and Lassez J, 1987, "Constraint Logic Programming", *Proc. POPL-87.*
7. Mackworth A, 1992, "The Logic of Constraint Satisfaction", *Artificial Intelligence*, 58, 3–20.
8. Rasmus D, 2000, "Knowledge Management Trends: The Role of Knowledge in E-Business", *PCAI Magazine*, 14(4), Special issue on Knowledge Management, Expert Systems and E-Business.
9. Bowen J and Bahler D, 1991, "Conditional Existence of Variables in Generalized Constraint Networks", *Proc. AAAI-91.*
10. Amin R and Bramer M, 2002, "Intelligent Data Analysis for Conservation: Experiments with Rhino Horn Fingerprint Identification", *Applications and Innovations in Intelligent Systems X, Proceedings of the 22nd SGAI International Conference on Knowledge Based Systems and Applied Artificial Intelligence.*
11. Dolsak B, 2002, "Finite Element Mesh Design Expert System", *Knowledge-Based Systems*, 15, 315–322.
12. Hossack J, 2002, "A Multi-Agent Intelligent Interpretation System for Power Distrubance Diagnosis", *Applications and Innovations in Intelligent Systems X, Proceedings of the 22nd SGAI International Conference on Knowledge Based Systems and Applied Artificial Intelligence.*
13. Bowen J and Bahler D, 1992, "Frames, Quantification, Perspectives and Negotiation in Constraint Networks for Life-Cycle Engineering", *International Journal of AI in Engineering*, 7, 199–226.

14. Bowen J, 2001, "Constraint-Based Co-operative Problem-Solving: A Case Study from Concurrent Engineering", *Proceedings CP-2001 Workshop on Cooperative Solvers in Constraint Programming*.
15. O'Sullivan B, 2001, "Constraint-Aided Conceptual Design", Professional Engineering Publishing.
16. Freuder E and O'Sullivan B, "Generating Tradeoffs for Interactive Constraint-Based Configuration", *Proceedings CP-2001*, 590–594.
17. Faltings B and Freuder E, 1998, (co-editors), *IEEE Intelligent Systems*, Special Issue on Configuration, 13, 4.
18. Bowen J and Likitvivtanavong C, 2002, "Question-Generation in Constraint-Based Expert Systems", Technical Report, CS Dept, UCC, Cork, Ireland.

A Note on Redundant Rules in Rule-Based Constraint Programming

Sebastian Brand

CWI, P.O. Box 94079, 1090 GB, Amsterdam, The Netherlands

Abstract. Constraint propagation can sometimes be described conveniently in a rule-based way. Propagation is then fixpoint computation with rules. In the typical case when no specific strategy guides the fixpoint computation, it is preferable to have a minimal set of rules. We propose a natural criterion for redundancy of a rule, and describe a test for a class of rules. Its relevance is demonstrated by applying it to several rule sets from two important approaches to automatic rule generation.

1 Introduction

Constraint propagation is an essential ingredient for successful solving of constraint satisfaction problems. A declarative approach to capturing the desired propagation is to use a rule-based language such as `CHR` and AKL, also ELAN, or the indexical-based clp(FD)/GNU-Prolog [9,6,12,10,7]. The rules are applied exhaustively in a fixpoint computation, and in this way constraint propagation takes place. The fixpoint represents a locally consistent state of the store.

A difficulty lies in obtaining suitable propagation rules. In the recent years a series of articles has dealt with this issue [5,11,2,3]. These approaches take constraint definitions, extensionally by stating all valid tuples or intensionally by a constraint logic program, and generate automatically a set of propagation rules.

Example. For the constraint $and(x, y, z)$ that describes $x \wedge y = z$ for variables $x, y, z \in \{0, 1\}$ we may generate the rules $x = 0 \rightarrow z = 0$; $y = 1, z = 0 \rightarrow x = 0$; etc. Applying them in any order to $and(x, 0, z)$ until no rule is applicable anymore leads to the fixpoint $and(x, 0, 0)$, which is generalized arc consistent.

Typically there are many valid rules for a constraint, but not all are useful in a constraint solver, or necessary for the desired local consistency. For example, a rule that is less general than another one can be discarded since its effect is subsumed. Rule generation therefore needs a concept of redundancy. The above-mentioned approaches deal explicitly with redundancy by one other rule, e. g., as in the case of $x = 0 \rightarrow z = 0$ subsuming $x = 0, y = 0 \rightarrow z = 0$; but only partly and informally with redundancy by several other rules. We argue here for considering redundancy rigorously with respect to a *set* of rules and its fixpoints.

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 109–120, 2003.

Example. For one constraint (`and11`), the rule generation algorithm of [5] computes a set of 4656 constraint propagation rules. It does not contain any subsumed rules. Removing from this set all rules that are redundant by the remaining set yields 393 rules. This minimal rule set (8% of the original size) enforces the same local consistency.

We proceed by introducing a simple and natural definition of redundancy in the general framework of fixpoint computation over functions [4]. This is then instantiated to the case of functions in rule form, which subsequently allows a connection between a few requirements on rules and a simple, effective redundancy test. We establish the usefulness of this by considering two important cases of automatic rule generation where automatic redundancy detection is essential: membership rules [5] and a class of `CHR` propagation rules (RuleMiner) [2]. Both fit in our framework, and hence our redundancy criterion can lead to reduced sets of rules.

2 Redundancy with Respect to Fixpoints

2.1 Preliminaries

We begin by recalling, concisely, chaotic iteration of [4], a general framework for constraint propagation. In section 4 we discuss a specific case.

Definition 1. *Let $(D, \sqsubseteq)$ be a partial order. Let F be a set of functions $\{f_1, \ldots, f_n\}$ on D. A function f is called*

- inflationary *if* $\quad d \sqsubseteq f(d) \quad$ *for all* d,
- monotonic *if* $\quad d \sqsubseteq e \;\Rightarrow\; f(d) \sqsubseteq f(e) \quad$ *for all* d, e. □

The elements of the partial order are used to model (equivalent) CSP's, the functions model propagation steps/narrowing operators/etc.

We are interested in fixpoints of the functions. By a chaotic iteration of F we mean an infinite sequence $d_0, d_1, \ldots$ such that $d_{i+1} = f(d_i)$ for some $f \in F$, and in which each $f \in F$ is applied infinitely often. An iteration stabilises at some e if an index k exists such that for all $i \geq k$ we have $d_i = e$. A simple consequence of the Stabilization Lemma in [4] is the following.

Lemma 2. *If all functions in F are inflationary and monotonic and the corresponding partial order $(D, \sqsubseteq)$ is finite then every chaotic iteration of F starting in d stabilises at the least fixpoint that is greater than or equal to d.* □

The least fixpoint greater than or equal to a given point can thus be effectively computed by repeatedly applying the functions in any order until stabilisation is obtained. For CSP's, constraint propagation steps taking place in any order should result in the same locally consistent state while retaining the solutions.

2.2 Redundant Functions

The cost of a generic fixpoint computation depends on the number of functions involved, in particular in absence of a good strategy to select functions. It is therefore useful to identify functions unnecessary for this computation. We define redundancy accordingly.

Definition 3. *A function f is redundant with respect to a set F of functions if the sets of fixpoints of F and $F \cup \{f\}$ are equal.* □

An equivalent formulation is: f is redundant if every fixpoint of F is a fixpoint of f as well.

3 Rules

The type of functions we are interested in are rules of the form $b \to g$, where b is a condition and the conclusion g a function. The condition is evaluated on the elements of D, we write $\mathit{Holds}(b, d)$ if b is true on d. The application of rules is defined by

$$(b \to g)(d) := \begin{cases} g(d) & \text{if } \mathit{Holds}(b, d) \\ d & \text{otherwise} \end{cases}$$

We are now interested in rules with particular properties.

Definition 4. *A condition b is*

- monotonic *if*

$$\mathit{Holds}(b, d) \ \wedge \ d \sqsubseteq e \quad \Rightarrow \quad \mathit{Holds}(b, e)$$

  for all d, e.
- precise *if a point w exists that satisfies $\mathit{Holds}(b, w)$, and also*

$$\mathit{Holds}(b, d) \quad \Rightarrow \quad w \sqsubseteq d$$

  for each d. We call w, which is unique, the precision witness *of b.*

A function g is stable *if*

$$g(d) \sqsubseteq e \quad \Rightarrow \quad g(e) = e$$

holds for all d, e □

The following connections between properties of a rule and properties of its body function are easy to prove.

Lemma 5. *Consider a rule $b \to g$. If g is inflationary then so is $b \to g$. If b and g are monotonic then so is $b \to g$.* □

Note also that if g is a function such that d and $g(d)$ are comparable for all d, then stability of g implies inflationarity.

3.1 Redundant Rules

The above properties are useful as they allow a simple test for redundancy.

Theorem 6. *Consider a rule $r = b \to g$ such that its condition b is monotonic and precise with witness w, and its body function g is stable. Let e be the least fixpoint of the rule set R greater than or equal to w. The rule r is redundant with respect to R if $g(e) = e$ holds.*

Proof. We show that $g(e) = e$ implies that an arbitrary fixpoint d of R is a fixpoint of r, by a case distinction on the condition.

b **holds at** d**:** We have $w \sqsubseteq d$ from the precision of b. Also, $w \sqsubseteq e \sqsubseteq d$ since e is the least fixpoint greater than or equal to w. From $e \sqsubseteq d$, $g(e) = e$, and stability of g we conclude $g(d) = d$ and finally $r(d) = (b \to g)(d) = d$.

b **does not hold at** d**:** Here $r(d) = (b \to g)(d) = d$ holds immediately. □

This test is interesting as it needs to compute only one fixpoint of R instead of all. It is effective if

- the precision witness is accessible,
- $g(e) = e$ can be decided, and
- fixpoint computations are effective (cf. Lemma 2).

Definition 7. *A set of rules R is said to be* minimal with respect to redundancy *(or just minimal), if no $r \in R$ is redundant with respect to $R - \{r\}$.* □

Minimal sets of rules can of course be obtained by a simple bounded loop: choose an untested rule, test whether it is redundant and remove it from the current set if it is. In general, however, the obtained minimal set depends on the order of testing; see the example later. This poses the questions for

- a total preference between two rule sets, and/or
- a rule selection strategy approximating such a preference.

For our experiments we used a selection strategy that prefers to retain rules that are deemed to be computationally less costly.

Note 8. What does it mean to remove a rule r from a minimal rule set R? Then $R - \{r\}$ must have more fixpoints than R. If R contains constraint propagation operators, then $R - \{r\}$ is propagates less. Minimal rule sets cannot be reduced further without relaxing the local consistency notion.

Partial redundancy. We point out that a rule $r = b \to g$ with a conclusion $g = g_1, \ldots, g_n$ (describing a composition) such that any two different g_i, g_j commute can be understood as the collection $b \to g_1, \ldots, b \to g_n$ of rules, and vice versa. The respective fixpoints, and rule properties, are the same. We regard here the different incarnations as equivalent. If a rule as a whole is not redundant it might be so partially. That is, some part of its conclusion is redundant, or in other words, some sub-rules of its decomposition are.

3.2 Redundancy by One Rule

We highlight a specific case involving only two rules, one 'stronger' than the other: subsumption.

Corollary 9. *Consider a rule $r = b \to g$ in a rule set R, and a rule $r' = b' \to g'$ not in R that satisfies the requirements mentioned in Theorem 6, and moreover is such that g' is inflationary. Assume that*

$$\mathit{Holds}(b', d) \;\Rightarrow\; \mathit{Holds}(b, d) \qquad \text{and} \qquad g'(d) \;\sqsubseteq\; g(d)$$

holds for all d. Then r' is redundant with respect to R.

Proof. Let e be the least fixpoint of R greater than or equal to the witness w' of b'. We show that $g'(e) = e$, which entails the desired result by Theorem 6.

We have $\mathit{Holds}(b', w')$, so by monotonicity of b' also $\mathit{Holds}(b', e)$. The first requirement above implies $\mathit{Holds}(b, e)$. We know for the fixpoint e that $e = r(e)$, and with $\mathit{Holds}(b, e)$ also $e = g(e)$. By the second requirement we conclude $g'(e) \sqsubseteq e = g(e)$, but g' is also inflationary: $e \sqsubseteq g'(e)$. Hence, $g'(e) = e$. □

4 Membership Rules

In [5] a specific type of rules, called membership rules, is presented. These rules are the output of a generation algorithm whose input is an extensionally defined constraint. The associated local consistency is generalized arc consistency. The generated rule set contains only *minimal rules* in the sense that no rule 'extends' or is subsumed by another. As it turns out, the redundancy notion proposed here can be applied to reduce the generated sets of membership rules.

We instantiate now the elements of rule-based chaotic iteration. First we introduce the elements of the partial ordering, CSP's.

Constraint satisfaction problems. Consider a sequence $X = x_1, \ldots, x_n$ of variables with associated domains $D_1, \ldots, D_n$. By a constraint C on X we mean a subset of $D_1 \times \ldots \times D_n$. Given an element $d = d_1, \ldots, d_n$ of $D_1 \times \ldots \times D_n$ and a subsequence $Y = x_{i(1)}, \ldots, x_{i(\ell)}$ of X we denote by $d[Y]$ the sequence $d_{i(1)}, \ldots, d_{i(\ell)}$. A constraint satisfaction problem (CSP), consists of a finite sequence of variables X with respective domains $\mathcal{D}$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write it as $\langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, where $X = x_1, \ldots, x_n$ and $\mathcal{D} = D_1, \ldots, D_n$. By a solution to $\langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, we mean an element $d \in D_1 \times \ldots \times D_n$ such that for each constraint $C \in \mathcal{C}$ on a sequence of variables X we have $d[X] \in C$.

Partially ordering equivalent CSP's. With a CSP $\langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ we associate now a specific partial order: the Cartesian product of the partial orders $(\mathcal{P}(D_i), \supseteq)$ Hence this order is of the form

$$(\mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_n), \supseteq)$$

where $\supseteq$ is the Cartesian product of the reversed subset order. The elements of this partial order are sequences $(E_1, .., E_n)$ of respective subsets of $(D_1, \ldots, D_n)$

The membership rules of [5] acceptable for a specific CSP over variables $X = x_1, \ldots, x_n$ with finite domains have the form

$$y_1 \in S_1, \ldots, y_k \in S_k \ \rightarrow \ z_1 \not\models a_1, \ldots, z_m \not\models a_m$$

where $y_1, \ldots, y_k$ are pairwise different variables from X, and $S_1, \ldots, S_k$ are subsets of the respective variable domains. Also the $z_1, \ldots, z_m$ are pairwise different variables from X, while $a_1, \ldots, a_m$ are elements of the respective variable domains. If each S_i is a singleton then a membership rule is called *equality rule*.

The computational interpretation of a membership rule is: if the current domain of the variable y_i is included in the set S_i for all $i \in [1..k]$, then remove the element a_j from the domain of z_j for all $j \in [1..m]$.

Applying membership rules. We define the atomic case

$$\mathit{Holds}(x_i \in S, E) \ := \ E_i \subseteq S$$

where $E = (E_1, \ldots, E_n)$ is an element of the partial order for a CSP with variables $x_1, \ldots, x_n$. A sequence of conditions holds if each atomic conditions does. Applying a rule body $z_1 \not\models a_1, \ldots, z_m \not\models a_m$ takes place stepwise with

$$(x_i \not\models a)(E) \ := \ E_i \setminus \{a\} \ .$$

Rule properties. Membership rules possess the relevant properties. To see this, consider a CSP $\langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ and an associated rule

$$b \rightarrow g \qquad \equiv \qquad y_1 \in S_1, \ldots, y_k \in S_k \ \rightarrow \ z_1 \not\models a_1, \ldots, z_m \not\models a_m \ .$$

Precise condition: We define

$$S'_i = \begin{cases} S_j & \text{if } x_i \text{ occurs in } b \text{ as } y_j \\ D_i & \text{if } x_i \text{ does not occur in } b. \end{cases}$$

Consider now the element $E = (S'_1, \ldots, S'_n)$ of the partial order. Observe that condition b holds at E. Moreover, it holds only for elements $E' \subseteq E$. Hence, E is a precision witness, and b is precise.

Monotonic condition: It is easy to see that this holds.

Monotonicity, inflationarity, stability of body: All true, by trivial proofs.

The redundancy test of Theorem 6 is therefore applicable to sets of membership rules. Note in particular that the required precision witness can be extracted immediately from the rule and the CSP at hand.

Table 1. Definition and membership rules for the constraint c

c	x	y	z	u
	0	1	0	1
	1	0	0	1
	1	1	1	0

$$
\begin{array}{lcll}
c(x,y,z,0) & \rightarrow & x \neq 0, y \neq 0, z \neq 0 & (1)\\
c(x,y,1,u) & \rightarrow & u \neq 1, \underline{x \neq 0, y \neq 0} & (2)\\
c(0,y,z,u) & \rightarrow & u \neq 0, y \neq 0, \underline{z \neq 1} & (3)\\
c(x,0,z,u) & \rightarrow & u \neq 0, x \neq 0, \underline{z \neq 1} & (4)\\
c(x,y,z,1) & \rightarrow & z \neq 1 & (5)\\
c(x,y,0,u) & \rightarrow & u \neq 0 & (6)\\
c(1,1,z,u) & \rightarrow & u \neq 1, \underline{z \neq 0} & (7)\\
c(x,1,0,u) & \rightarrow & x \neq 1 & (8)\\
c(x,1,z,1) & \rightarrow & \underline{x \neq 1} & (9)\\
c(1,y,0,u) & \rightarrow & y \neq 1 & (10)\\
c(1,y,z,1) & \rightarrow & \underline{y \neq 1} & (11)
\end{array}
$$

4.1 An Example

We illustrate a number of issues by way of example. Suppose the constraint $c(x, y, z, u)$ is defined by the three solution tuples of Table 1. The underlying domain for all its variables is $\{0, 1\}$, hence the induced corresponding partial order is $(\{\{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\}, \ldots, \emptyset \times \emptyset \times \emptyset \times \emptyset\}, \supseteq)$. The algorithm of [5] produces a set of 11 rules.

The presence of rule 11, $c(1, y, z, 1) \rightarrow y \not= 1$, states that if$c(x, y, z, u)$ then it is correct to conclude from $x = 1$ and $u = 1$ that $y \not= 1$ (validity), and furthermore that neither $x = 1$ nor $u = 1$ suffices for this conclusion (minimality).

Let us examine a fixpoint of these rules: we are interested in the smallest fixpoint greater than or equal to $E_1 = \{1\} \times \{0, 1\} \times \{0, 1\} \times \{1\}$ which corresponds to the CSP in the preceding paragraph. Suppose rule 11 is considered. Its application yields $E_2 = \{1\} \times \{0\} \times \{0, 1\} \times \{1\}$ from where rule 4 leads to $E_3 = \{1\} \times \{0\} \times \{0\} \times \{1\}$. This is indeed a fixpoint since each rule either does not apply or its application results again in E_3.

A redundant rule. A second possible iteration from E_1 that stabilises in E_3 is by rule 5 followed by rule 10. Rule 11 can be applied at this point but its body effects no change on E_3; E_3 is a fixpoint of all rules including rule 11. We conclude that rule 11 is redundant — we just performed the test of Theorem 6.

The process of identifying redundant rules can then be continued for the rule set $\{1, \ldots, 10\}$. A possible outcome is depicted in Table 1, where redundant parts of rules are underlined. From the 20 initial atomic conclusions 13 remain, thus we find here a reduction ratio of 65%.

Nondeterminism in rule removal. Consider the justification for the redundancy of rule 11, and observe that rule 11 has no effect since rule 10, which has the same body, was applied before. Suppose now that the process of redundancy identification is started with rule 10 instead of rule 11. This results in rule 10 being shown redundant, with a relevant application of rule 11.

Note moreover, that one of the rules 10,11 must occur in *any* reduced rule set since their common body $y \not= 1$ occurs nowhere else. It is unclear whether there is a sensible criterion which of these two rules it should be.

4.2 Membership Rule Generation

The algorithm of [5] that generates the set of rules from a constraint definition can straightforwardly be abstracted to:

1. collect all (syntactically and semantically) valid membership rules
2. discard those that *extend* another present rule

The membership rule $b' \to g'$ *extends* the rule $b \to g$ if $g' = g$ and (using our terminology) $\mathit{Holds}(b', d) \to \mathit{Holds}(b, d)$. If in a given rule set a rule does not extend another one then it is called *minimal*.

The minimality requirement is weaker than that of Corollary 9. A non-minimal rule is therefore a special case of the rule being redundant. We conclude that a generated set of membership rules has the same set of fixpoints as the set of all valid membership rules, and point out that the generated set can be expected not to be minimal, even though it contains only minimal rules.

5 Propagation Rules

The concept of a CSP in the preceding section emphasises domains as a channel through which constraint propagation takes place. We now take a different viewpoint: we understand a CSP as just a set of constraints. Domains are unary constraints, and propagation means posting implied constraints.

5.1 Abstract Propagation Rules

A propagation rule in this framework describes that some constraints should be added to the problem if some other constraints are present. We show in the following that this model as well fits into the partial order framework. The major goal we pursue is to make a connection between the partial order framework, and propagation rules as used in the `CHR` language and in the work [2] on automatic rule generation.

Constraints and CSP's. A constraint is an atomic formula on variables. The language $\mathcal{L}$ is defined as consisting of all constraints over a finite set of variables and a finite set of constraint symbols. A CSP is a set of such constraints. The corresponding partial order on these CSP's is simply $(\mathcal{P}(\mathcal{L}), \subseteq)$.

Rules and applications. A propagation rule has the form $C_L \to C_R$ where C_L and C_R are sets of constraints of $\mathcal{L}$. Suppose $E \subseteq \mathcal{L}$ is a set of constraints. The evaluation of the condition C_L on E is defined by $\mathit{Holds}(C_L, E) := C_L \subseteq E$ while the application of the body C_R is simply $C_R(E) := E \cup C_R$. This explains completely the rule application $(C_L \to C_R)(E)$.

Properties. The relevant rule properties hold for propagation rules. The proof obligations for an arbitrary $C_L \to C_R$ follow (the proofs themselves are trivial).

Precise condition: Take the witness C_L. To show: $C_L \subseteq E \quad \Rightarrow \quad C_L \subseteq E$
Monotonic condition: $E_1 \subseteq E_2 \quad \Rightarrow \quad E_1 \cup C_R \subseteq E_2 \cup C_R$
Monotonic body: $C_L \subseteq E_1 \wedge E_1 \subseteq E_2 \quad \Rightarrow \quad C_L \subseteq E_2$
Stability of body: $C_L \cup E_1 \subseteq E_2 \quad \Rightarrow \quad C_L \cup E_2 = E_2$
Inflationarity of body: $C_L \cup E_1 \subseteq E_2 \quad \Rightarrow \quad C_L \cup E_2 = E_2$

Abstract propagation rules are amenable to the redundancy test of Theorem 6.

5.2 A Class of CHR Propagation Rules

`CHR` [9] is a high-level rule-based language especially designed for writing constraint solvers. It defines propagation rules that can add constraints, and simplification rules that replace constraints by simpler ones. Both types can have an extra guard on the variables. The class of guard-free `CHR` propagation rules in which the body does not introduce new variables and is interpreted as a set of constraints, however, corresponds to abstract propagation rules. An important contrast point is that `CHR` interprets a rule as a template of which a copy with fresh variables is matched against the constraint store.

RuleMiner rules. The work [2] describes the generation by an algorithm called "RuleMiner" of constraint propagation rules that in particular can have multiple constraints in its condition. The generated rules correspond to guard-free `CHR` propagation rules. The requirement that no new variables be introduced in the conclusion of a rule is guaranteed by the generation algorithm.

Generation. Several criteria to discard a rule are used in RuleMiner. The single most important one is called *lhs-cover*. A rule $C'_L \to C'_R$ is *lhs-covered* by $C_L \to C_R$ if $C'_L \supseteq C_L$ and $C'_R \subseteq C_R$. This coincides with the condition of Corollary 9. The notion of *lhs-cover* is a special case of general redundancy.

6 Experiments

We implemented in ECL^{i}PS^e [8] the computation of minimal sets for membership rules and RuleMiner rules. The results for some benchmark rule sets are listed in Tables 2 and 3.

Table 2. Membership rules

	$\texttt{and11}_M$	$\texttt{and11}_E$	$\texttt{and3}_M$	$\texttt{equ3}_M$	$\texttt{fula2}_E$	$\texttt{fork}_E$	$\texttt{fork}_M$
total	4656	153	18	26	52	12	24
redundant (partially)	4263 (2)	0 (6)	5 (0)	8 (0)	24 (0)	0 (9)	6 (6)
redundancy ratio	81%	4%	30%	26%	35%	35%	40%

Membership rules. For each constraint, the set of minimal membership or equality rules (index M/E) was computed by the rule generation algorithm of [5]. Table 2 shows the size of the rule set, the number of fully and, in parentheses, partially redundant rules. The redundancy ratio for the entire rule set shows the percentage of atomic disequalities that were removed from the rule conclusions.

The constraints are the `and` logic gate for a number of logics (the numeric suffix states the domain size), the binary `fulladder` (arity 5), the equivalence relation for a three-valued logic, and the `fork` constraint from the Waltz language for the analysis of polyhedral scenes.

Computation times are negligible in so far as they are considerably less than the corresponding rule generation time.

RuleMiner rules. The authors of [2] kindly provided us with some generated rule sets for the constraints `and`, `or`, `xor` in a 6-valued logic. Rules embody propagation from single constraints, and propagation from pairs of constraints. In both cases additional equality constraints between two variables, or a variable and a constant, can occur in a rule condition. The conclusion of a rule consists of equalities and disequalities. Two following two rules are examples:

$$\begin{aligned} and(x,x,z) &\rightarrow x \not\models \overline{\mathsf{d}}, x \not\models \mathsf{d}, x = z \\ and(x,y,z), or(z,y,1) &\rightarrow z \not\models \overline{\mathsf{d}}, z \not\models \mathsf{d}, x = z, y = 1 \end{aligned}$$

They are rewritten so as to fit the format of abstract propagation rules, introducing new variables and equalities in the heads. Correspondingly we assume appropriate rules for handling such equality constraints. They are not tested for redundancy, but are otherwise proper part of the rule set.

The first three table rows in Table 3 describe the results for rule sets corresponding to the single constraints. The three center rows contain the results for rule sets for pairs. Finally, the last three rows correspond to the union of the rule sets for a pair of constraints and its respective individual constraints. This would be the appropriate use configuration.

The tested RuleMiner rule sets did not contain any fully redundant rules. This can be attributed to the additional redundancy criteria other than lhs-cover. The rule sets did contain partial redundancies, however.

Table 3. RuleMiner rules

	`and`	`or`	`xor`	`andor`	`andxor`	`orxor`	`andor+`	`andxor+`	`orxor+`
total	19	19	28	138	207	199	176	254	246
partially redundant	7	7	1	83	82	77	135	192	184
redundancy ratio	24%	24%	3%	38%	21%	21%	61%	54%	54%

7 Final Remarks

Closely related recent work can be found in [1]. The issue of redundancy is there examined for full `CHR` rules, using concepts derived from term rewriting theory. The class of `CHR` rules is more expressive than our rule class. One substantial difference is the presence of simplification rules, which remove constraints from the store, and are thus non-monotonic and non-inflationary. Consequently, the proposed redundancy test is less specific than ours, by appealing more abstractly to termination, confluence, and operational equivalence of original and reduced program. Here, we look at rules for which Lemma 2 implicitly guarantees the former two properties, while Theorem 6 constitutes a concrete test of operational equivalence. Benefitting from inflationarity and monotonicity, we can do with only one fixpoint computation per candidate rule, while two computations are needed in the method of [1]: with and without the candidate.

Completion. There is a link between completion of term rewriting systems and redundancy. Completion adds rules to a rule set so as to make it confluent, that is, to prevent the possibility that some state exists from which two iterations stabilise in different fixpoints. In this case a new rule is introduced that joins both iterations (in effect, removing one fixpoint). The new rule thus permits an alternative iteration leading to the same fixpoint. Redundancy removal, in contrast, tries to *prevent* alternative iterations, by removing a rule that occurs in one but not the other, while retaining the fixpoints.

Benefit. It seems obvious that discarding a larger number of redundant rules accelerates fixpoint computation. It is less clear, however, whether this is true when removing one single rule. For particular combinations of scheduler, rule set, and starting point of computation, the effect might indeed be adverse. This is more relevant still for the case of a partially redundant rule. We can not, therefore, say in general that reducing redundancy is always useful (although in our experiments that was the case). Observe, however, that partial redundancy can be easily reintroduced.

Future Issues. An open question is the relation between two minimal sets. How many different minimal sets are there if we define equivalence classes on

rules (sets) such that for instance rule 10 and 11 of subsection 4.1 belong to one? How is such an equivalence detected?

The partial orders for membership rules and abstract propagation rules could be integrated into one. The inclusion test of a membership rule could correspond to a unary constraint or a set of disequalities. A treatment of a wider class of `CHR` propagation rules as abstract propagation rules is possible. For instance, guards of rules could be seen as constraints.

Rule generation, specifically of membership rules, should ideally incorporate the redundancy test as a part rather than as a post-process. It is then useful to identify preconditions that prevent a rule from being redundant with respect to the partial rule set. One such is that the body of the rule must occur in another rule.

Acknowledgements. I am grateful for comments on this paper that I received from Krzysztof Apt, Eric Monfroy, and the anonymous referees.

References

1. Slim Abdennadher and Thom Frühwirth. Using program analysis for integration and optimization of rule-based constraint solvers. In *Onzièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'2002)*, May 2002.
2. Slim Abdennadher and Christophe Rigotti. Automatic generation of propagation rules for finite domains. In *Principles and Practice of Constraint Programming (CP 2000), Singapore*, 2000.
3. Slim Abdennadher and Christophe Rigotti. Towards inductive constraint solving. *Lecture Notes in Computer Science*, 2239, 2001.
4. Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, June 1999.
5. Krzysztof R. Apt and Eric Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 2001.
6. Björn Carlson, Mats Carlsson, and Sverker Janson. The implementation of AKL(FD). In *International Symposium on Logic Programming*, 1995.
7. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, June 1996.
8. IC-Parc. *ECLiPSe*. `http://www.icparc.ic.ac.uk/eclipse/`.
9. Th. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, pages 95–138, October 1998.
10. Claude Kirchner and Christophe Ringeissen. Rule-based constraint programming. *Fundamenta Informaticae*, 34(3):225–262, 1998.
11. C. Ringeissen and E. Monfroy. Generating propagation rules for finite domains via unification in finite algebras. In *New Trends in Constraints*, 2000.
12. Vijay Anand Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

A Study of Encodings of Constraint Satisfaction Problems with 0/1 Variables*

Patrick Prosser and Evgeny Selensky

Department of Computing Science, University of Glasgow, Scotland.
pat/evgeny@dcs.gla.ac.uk

Abstract. Many constraint satisfaction problems (csp's) are formulated with 0/1 variables. Sometimes this is a natural encoding, sometimes it is as a result of a reformulation of the problem, other times 0/1 variables make up only a part of the problem. Frequently we have constraints that restrict the sum of the values of variables. This can be encoded as a simple summation of the variables. However, since variables can only take 0/1 values we can also use an occurrence constraint, e.g. the number of occurrences of 1 must be k. Would this make a difference? Similarly, problems may use channelling constraints and encode these as a biconditional such as $P \leftrightarrow Q$ (i.e. P if and only if Q). This can also be encoded in a number of ways. Might this make a difference as well? We attempt to answer these questions, using a variety of problems and two constraint programming toolkits. We show that even minor changes to the formulation of a constraint can have a profound effect on the run time of a constraint program and that these effects are not consistent across constraint programming toolkits. This leads us to a cautionary note for constraint programmers: take note of how you encode constraints, and don't assume computational behaviour is toolkit independent.

1 Introduction

A constraint satisfaction problem (csp) is composed of a set of variables, each with a domain of values. Constraints restrict combinations of variable assignments. The problem is to find an assignment of values to variables that satisfies the constraints, or show that none exists [10]. There are many real world instances of csp's, such as scheduling, timetabling, routing problems, frequency assignment, design problems, etc. Since many of these problems are commercially important, we now have toolkits that allow us to express these problems as csp's.

Even when we have decided upon a formulation of a csp, we are then faced with choice of how we implement the constraints using the toolkit provided. What we investigate here is how the implementation of the constraints can influence the execution time of our constraint programs. We limit our study to problems with variables that can only take the values zero or one, i.e. 0/1 variables, and to two constraints: a restriction on the sum of the variables, and

* This work was supported by EPSRC research grant GR/M90641 and ILOG SA.

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 121–131, 2003.

the biconditional constraint. We use three different problems as vehicles for this study. The first problem is the *independent set* of a hypergraph. The second problem is closely related, the *maximal independent set* of a hypergraph. Both of these problems have a natural encoding using 0/1 variables. The third problem is the balanced incomplete block design problem (bibd), and is again naturally formulated using 0/1 variables. For each of these problems we encode the constraints in a number of different ways and measure the run time to find the first solution. We use two constraint programming toolkits, Ilog Solver 5.0 [5] and Choco 1.07 [1].

In the next section we introduce the three problems, independent set, maximal independent set, and balanced incomplete block design. We also present a proof that our various encodings achieve the same level of consistency. Section 3 gives the results of our empirical study. We then imagine a study based on the encodings we have studied and show how this can lead us to a contradiction. We then present an explanation of the sensitivity of performance with respect to the implementation of our constraints. Section 6 concludes this paper.

2 Three Problems

We now present the three problems we will investigate, and their various encodings. The first problem is *independent set* and we use this as a vehicle to examine the implementation of a constraint that restricts the sum of variables. The second problem is *maximal independent set* and we use this to explore how we can implement the biconditional. The third problem, balanced incomplete block designs uses both constraints, summation and biconditional.

2.1 Independent Set

Given a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, an independent set I is a subset of V such that no two vertices in I share an edge in E. Independent set is one of the first NP-complete problems. A variant of it, GT20 in Garey and Johnson [2], asks if there is an independent set of size k or larger. For a hypergraph $H = (V, E)$ each edge in E is a subset of the vertices in V, and an independent set I of H is then a subset of V such that no edge $e \in E$ is subsumed by I. For example we might have a hypergraph H of 9 vertices, v_1 to v_9, and 4 hyper edges $\{(v_1, v_2, v_3), (v_2, v_4, v_5), (v_4, v_6), (v_3, v_7, v_8, v_9)\}$. This is shown in Figure 1. An independent set of size 7 is then $I = \{v_1, v_2, v_5, v_6, v_7, v_8, v_9\}$.

We can formulate this problem as a csp, such that each vertex v_i corresponds to a 0/1 variable x_i, and if $x_i = 1$ then v_i is in the independent set I. The *independence* constraint restricts the sum of the variables/vertices in a hyperedge to be less than the arity of that hyperedge, where arity is the number of variables/vertices involved in that hyperedge. There is an independence constraint for each hyperedge. Finally we have the constraint that the sum of all the variables has to be greater than or equal to k, i.e. the size of the independent set.

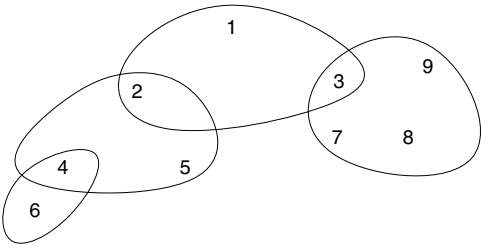


Fig. 1. Our example hypergraph, $H = (V, E)$ where $E = \{(v_1, v_2, v_3), (v_2, v_4, v_5), (v_4, v_6), (v_3, v_7, v_8, v_9)\}$. An independent set of size 7 is then $I = \{v_1, v_2, v_5, v_6, v_7, v_8, v_9\}$

The problem of finding an independent set of a given size is of particular interest. This is because any constraint satisfaction problem, with discrete and finite domains, can be formulated as the problem of finding an independent set [12]. Assume that we have a csp P with n variables x_1 to x_n, each with a domain d_i of size m_i. In its hypergraph representation H we have k variables each with a domain $\{0, 1\}$, where $k = \sum_{i=1}^{n} m_i$. Variable $z_{i,a} = 1$ corresponds to the instantiation $x_i = a$ in P, and $z_{i,a} = 0$ corresponds to $x_i \; /\!\!= a$ in P. The domain of each variable x_i in P is represented as a clique K_m in H, such that there are edges $(z_{i,a}, z_{i,b})$ for all $a, b \in d_i$. The constraints in the original problem P are represented as hyperedges in H. For example, a nogood $(x_i = a, x_j = b, ..., x_r = j)$ in P has the corresponding hyperedge $(z_{i,a}, z_{j,b}, ..., z_{r,j})$ in H. Finding a solution to P corresponds to finding an independent set of size n in H.

Therefore, given a hyperedge e, where $vert(e)$ is the set of vertices involved in e and $arity(e)$ is the number of vertices involved in e, we have the corresponding independence constraint $\sum_{x_i \in vars(c)} x_i < arity(c)$, where each of the variables x_i in constraint c corresponds to the vertices v_i in hyperedge e.

We consider encoding the independence constraint in two ways. First, and most obviously, we perform arithmetic on the values of the variables in the hyperedge and restrict them to be less than the arity. In Choco this would be done as $sumVars(x) <= length(x)$ where x is a list of integer variables (and in Ilog Solver $IloSum(x) <= x.getSize()$, where x is an array of integer variables). Alternatively, since variables are constrained to take 0/1 values, we can state that the number of occurrences of the value 1 must be less than or equal to k. Again in Choco, we express this as $occur(1, x) <= length(x)$, and in Solver we use the $IloDistribute$ function. We call the first encoding $ind1$ and the second $ind2$. We now prove that these two encodings achieve the same level of consistency [1].

[1] This proof is due to Francois Laburthe.

Theorem 1. *Generalised arc consistency (GAC) on the occur constraint in ind*2 *achieves the same level of consistency as bounds consistency (BC) on the sum constraint in ind*1*.*

Proof. We need to prove that $(BC(a \leq sum(X) \leq b) \Leftrightarrow GAC(a \leq occur(1, X) \leq b)$, where X is a set of 0/1 variables $\{x_1, ..., x_n\}$. Since both constraints have the same solution set, and GAC is stronger than BC we know that $GAC(occur(1, X))$ is at least as strong as $BC(sum(X))$ i.e. any value removed by $BC(sum(X))$ is also removed by $GAC(occur(1, X))$. We now prove that any value removed by $GAC(occur(1, Xi))$ is also removed by $BC(sum(X))$. Let nb_1 be the number of variables instantiated to 1, nb_0 be the number of variables instantiated to 0, and $nb_{0/1}$ be the number of uninstantiated variables. Suppose the value $x_i = 0$ is removed by $GAC(occur(1, X))$. We consider two cases:

(i) All other variables are instantiated. Consequently both constraints reduce to unary constraints, and $x_i = 0$ is also removed by $BC(sum(X))$.

(ii) Variables $x_{j1}, ..., x_{jk}$ are not instantiated, and the tuple $(x_i = 0, x_{j1} = 1, ..., x_{jk} = 1)$ along with the instantiated variables is infeasible. Consequently $nb_1 + nb_{0/1} - 1$ is not in the interval $[a, b]$ Similarly, the tuple $(x_i = 0, x_{j1} = 0, ..., x_{jk} = 0)$ along with the instantiated variables is infeasible. Consequently nb_1 is not in $[a, b]$. Therefore the intervals $[nb_1, nb_1 + nb_{0/1} - 1]$ and $[a, b]$ are disjoint. Either (a) or (b) hold
   (a) $nb_1 + nb_{0/1} - 1 < a$. So, $sum(X - \{x_i\}) < a$, where all uninstantiated variables in $X - \{x_i\}$ contribute the value 1 to the sum. $BC(sum(X))$ then removes the value $x_i = 0$
   (b) $nb_1 > b$. So $sum(X - \{x_i\}) > b$, where all uninstantiated variables in $X - \{x_i\}$ contribute the value 0 to the sum. Again, $BC(sum(X))$ removes the value $x_i = 0$.

   Therefore, when $GAC(occur(1, X))$ removes a value so does $BC(sum(X))$.

Since GAC is stronger than BC, and whenever $GAC(occur(1, X))$ removes a value so does $BC(sum(X))$, the two representations *ind*1 and *ind*2 achieve the same level of consistency. *QED*

2.2 Maximal Independent Set

Given a hypergraph $H = (V, E)$, where V is the set of vertices and E is the set of edges, a *maximal* independent set M is a set such that there is no independent set M' that subsumes M, i.e. we cannot add any vertex to M without losing the independence property. The problem is then, given some integer $k < |V|$, is there a maximal independent set of size k [8]?

Using Figure 1 as an example, when $k = 5$ there are 3 maximal independent sets, one of these being $\{v_2, v_3, v_4, v_8, v_9\}$. When $k = 6$ there are 11 maximal independent sets, one of these being $\{v_2, v_3, v_5, v_6, v_8, v_9\}$. When $k = 7$ there is a single maximal independent set $\{v_1, v_2, v_5, v_6, v_7, v_8, v_9\}$ and this is the largest, and is therefore the maximum independent set. There are no maximal independent sets for any other values of k.

We need a constraint to specify when a vertex is in the maximal independent set, and when it is not in the maximal independent set. Looking again at Figure 1 we can see that vertex v_2 is not in the maximal independent set M if vertices v_4 and v_5 are in M or v_1 and v_3 are in M. We can express this with the following constraint:

$$(v_4 + v_5 = 2) \vee (v_1 + v_3 = 2) \leftrightarrow v_2 = 0$$

where $\leftrightarrow$ is the biconditional *if and only if*. Alternatively we can express when v_2 is in M. Therefore we might alternatively have the constraint

$$(v_4 + v_5 < 2) \wedge (v_1 + v_3 < 2) \leftrightarrow v_2 = 1$$

The biconditional $p \leftrightarrow q$ is logically equivalent to $(p \wedge q) \vee (\neg p \wedge \neg q)$ and to $(p \rightarrow q) \wedge (q \rightarrow p)$. Therefore using the definitions below for (1) p, (2) $\neg p$, (3) q and (4) $\neg q$ we can describe the maximality constraint equivalently as $p \leftrightarrow q$, or as $(p \rightarrow q) \wedge (q \rightarrow p)$, or as $(p \wedge q) \vee (\neg p \wedge \neg q)$

$$p : \quad v_i = 1 \tag{1}$$

$$\neg p : \quad v_i = 0 \tag{2}$$

$$q : \bigwedge_{e \in E(v_i)} \sum_{v_j \in V(e) - v_i} v_j < arity(e) - 1 \tag{3}$$

$$\neg q : \bigvee_{e \in E(v_i)} \sum_{v_j \in V(e) - v_i} v_j = arity(e) - 1 \tag{4}$$

where $E(v_i)$ is the set of edges involving variable/vertex v_i and $V(e) - v_i$ is the set of variables/vertices in the edge e excluding vertex v_i.

As stated above, the maximum independent set is also a maximal independent set. Consequently when reformulating a csp P of n variables as a problem of finding an independent set of size n, we could also incorporate the redundant maximality constraints. Therefore one of the questions we investigate is, does the redundant maximality constraint improve search performance?

2.3 Balanced Incomplete Block Design

A balanced incomplete block design (bibd) is an arrangement of v objects into b blocks, each of size k. Each element of v occurs in r blocks and every possible pair of objects occurs together in λ blocks [6] Therefore, a bibd can be defined by the quintuple $\langle v, b, r, k, \lambda \rangle$, and visualised as a v by b matrix of 0/1 values. There are r 1's in each row, k 1's in each column, and the scalar product of any two rows is equal to λ. Tabulated below is a matrix for the bibd $\langle 6, 10, 5, 3, 2 \rangle$

We encode this in the most naive way [2]. We have $v \times b$ 0/1 variables, $v_{i,j}$. There are v sum constraints for each row and b sum constraints for each column. For every pair of rows, (i, j), we generate b additional variables $l_{i,j,1}$ to $l_{i,j,b}$, and b constraints of the form $v_{i,k} = 1 \wedge v_{j,k} = 1 \leftrightarrow l_{i,j,k} = 1$. Finally, for the pair of rows we have the $b-$ary constraint $\sum_{k=1}^{b} l_{i,j,k} = \lambda$.

The biconditional can again be encoded in three ways, as described in the previous subsection, and the summation constraints can be encoded as occurrence constraints.

[2] A more efficient encoding is proposed in [7]

Table 1. An instance of bibd $\langle 6, 10, 5, 3, 2\rangle$

0	0	0	0	0	1	1	1	1	1
0	0	1	1	1	0	0	0	1	1
0	1	0	1	1	0	1	1	0	0
1	0	1	0	1	1	0	1	0	0
1	1	0	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0	1	0

3 The Empirical Study

The experiments were run on two different machines. The Choco experiments were run on a Pentium III 755 MHz processor with 256MB of ram, and the Solver experiments on a Pentium III 933 MHz processor with 1GB of ram. We use bibd's as data sets for our study of (maximal) independent sets. The bibd can be viewed as a regular hypergraph, with each vertex of degree r and each hyperedge of arity k. We use three such hypergraphs which we denote A, B, and C. Hypergraphs A and B both correspond to non-isomorphic instances of the bibd $\langle 25, 50, 8, 4, 1\rangle$ and C corresponds to an instance of $\langle 40, 130, 13, 4, 1\rangle$.

Table 2. Search effort (nodes) and CPU time (milliseconds) to find the first independent set of size k for hypergraphs A, B, and C.

	k	Sol	Nodes	ind1S	ind2S	ind1C	ind2C
A	14	yes	170	125	93	290	20
	15	no	36901	781	2812	70350	3530
	16	no	17652	406	1359	35190	1760
	17	no	7585	218	625	16420	800
	18	no	3150	125	297	7220	350
B	14	yes	16	62	62	20	0
	15	yes	16	62	62	20	0
	16	no	17516	422	1344	36280	1740
	17	no	7503	218	625	16960	790
	18	no	3058	125	281	7050	340
C	21	yes	19219	609	2640	78280	2760
	22	yes	101217	2875	13172	-	14320
	23	no	8237508	225703	1059660	-	1147860
	24	no	4136599	114734	512015	-	580200

In Table 2 we have the results of searching for the first independent set of size k for the three hypergraphs. The column *Sol* is "yes" if an independent set

of size k was found. Note that k is increasing as we move down the table, and the last "yes" entry corresponds to the largest independent set. Results are given for Ilog Solver 5.0 and Choco 1.07 implementations, where $ind1$ uses summation of variables and $ind2$ uses the occurrence constraint, $ind1S$ and $ind2S$ are the Solver implementations and $ind1C$ and $ind2C$ are the Choco implementations. Both implementations explore the same number of nodes, as expected. Run time is given in milliseconds, and where there is a $-$ entry, the process was terminated after more than 16 hours.

The difference between the Solver implementations is large, with the summation constraint ($ind1S$) significantly faster than the occurrence constraint ($ind2S$). The difference is always in $ind1S$'s favour and is typically about a factor of three. In Choco the difference is typically a factor of about twenty, but this time in the favour of $ind2C$. That is, the occurrence implementation dominates the summation implementation in Choco by a factor of twenty, whereas in Solver the summation dominates the occurrence constraint by a factor of three.

Table 3. Search effort (nodes) and CPU time (milliseconds) to find the first maximal independent set of size k for hypergraphs A and B.

	k	Nodes	mis1S	mis2S	mis3S	mis1C	mis2C	mis3C
A	14	62	78	94	78	40	60	40
	15	25093	6781	9781	3375	18450	31090	35620
	16	15862	3469	5016	1969	13210	18630	22560
	17	7585	1516	2172	969	5530	7380	9200
	18	3150	641	891	437	1840	2630	3580
B	14	87	94	109	78	50	90	20
	15	16	62	62	62	10	10	1860
	16	15744	3453	4969	1937	14080	17980	22210
	17	7485	1484	2156	937	4480	7500	9300
	18	3058	609	875	422	2590	3610	3560

Table 3 gives the results of our study of the maximal independent set problems, i.e. a study of our different encodings of the biconditional constraint $p \leftrightarrow q$. We use the two hypergraphs A and B, and search for the first maximal independent set of size k. Again, experiments were performed in Ilog Solver and in Choco. The column $mis1S$ gives the Solver run time (milliseconds) for the encoding of the biconditional $p \leftrightarrow q$, and $mis1C$ is the Choco equivalent. Columns $mis2S$ and $mis2C$ report on the encoding of the biconditional in Solver and Choco as $(p \rightarrow q) \wedge (q \rightarrow p)$, and columns $mis3S$ and $mis3C$ as $(p \wedge q) \vee (\neg p \wedge \neg q)$. All these encodings use the summation constraint, rather than the occurrence constraint. From our results we see that in Solver $(p \wedge q) \vee (\neg p \wedge \neg q)$ (i.e. column $mis3S$) is the fastest implementation of the biconditional, typically 50% faster

than $mis1S$ and twice as fast as $mis2S$. In Choco $p \leftrightarrow q$ (i.e. $mis1C$) is the fastest implementation of the biconditional.

Our final experiments are on first solution search for bibd using Choco. We use the same problems studied by Prestwich [9].

Table 4. CPU time (milliseconds) to find the first bibd that satisfies the given parameters

Parameters	bibd0	bibd1	bibd2	bibd3	bibd4	bibd5
$\langle 7,7,3,3,1\rangle$	30	10	10	10	20	10
$\langle 6,10,5,3,2\rangle$	70	30	40	30	40	3670
$\langle 7,14,6,3,2\rangle$	400	150	220	170	360	20
$\langle 9,12,4,3,1\rangle$	260	90	150	120	560	40
$\langle 8,14,7,4,3\rangle$	1000	340	500	410	820	–

Six different encodings are used. *bibd*0 uses the summation constraint, whereas all others (*bibd*1 to *bibd*5) use the occurrence constraint. *bibd*0 and *bibd*1 encode the biconditional as $p \leftrightarrow q$, whereas *bibd*2 uses $(p \rightarrow q) \wedge (q \rightarrow p)$. Encoding *bibd*3 uses multiplication, i.e. the constraint $(X = 1 \wedge Y = 1) \rightarrow Z = 1$ can be replaced with $X \times Y = Z$, because $X, Y, Z \in \{0, 1\}$. Encoding *bibd*4 uses the Choco constraint $and(ifThen(p, q), ifThen(q, p))$, and this behaves as two lazy implications. Encoding *bibd*5 uses $(p \wedge q) \vee (\neg p \wedge \neg q)$. These six encodings are all logically equivalent, and the search processes all visit the same number of nodes.

Comparing *bibd*0 with *bibd*1 we see again that in Choco the occurrence constraint is more efficient than the summation constraint, although not as significantly as in independent set (Table 2). Table 4 shows that the direct encoding of the biconditional, *bibd*1, gives the best performance. Multiplication, *bibd*3 is the next best thing, whereas the encoding $(p \wedge q) \vee (\neg p \wedge \neg q)$ is again the worst, failing to terminate in reasonable time on $\langle 8, 14, 7, 4, 3\rangle$.

4 Different Model, Different Conclusion

The notion of maximality can be encoded as a redundant constraint when we are searching for an independent set of size k if and only if we know that k is the size of the largest independent set. In particular, we can use this constraint when we reformulate a csp of n variables into the problem of finding an independent set of size n in the corresponding hypergraph. Would this redundant constraint pay off? We can compare some of the results from Table 2 with those in Table 3 to allow us to imagine how such an investigation might proceed.

The largest tabulated value of k for which we find a maximal independent set also corresponds to the size of the largest independent set. Therefore we can

compare Tables 2 and 3 for hypergraph A with $k \geq 14$, and B with $k \geq 15$ (i.e. B's independent set of size 14 might not be maximal, therefore we can only consider B problems with $k \geq 15$).

In our imaginary (Choco) experiments we encode our problem using the summation constraint and search for an independent set of size k. This corresponds to column $ind1C$ in Table 2. We then encode our problem again, this time using the redundant maximality constraint. Assume this uses the encoding of the biconditional corresponding to $mis1C$ in Table 3, i.e. using Choco's $ifOnlyIf$ constraint. This is re-tabulated in Table 5, comparing only $mis1$ with $ind1$ in both Choco and Solver. The redundant constraint gives us a speed up of a factor of about three. This suggests that our reformulation is an improvement, and we might recommend it. In fact, we get an improvement regardless of our encodings of the biconditional.

Table 5. CPU time (milliseconds) to find the largest independent set, with (mis) and without (ind) the redundant maximality constraint. Does maximality help? It depends on the toolkit.

	k	ind1S	mis1S	ind1C	mis1C
A	14	125	78	290	40
	15	781	6781	70350	18450
	16	406	3469	35190	13210
	17	218	1516	16420	5530
	18	125	641	7220	1840
B	15	62	62	20	10
	16	422	3453	36280	14080
	17	218	1484	16960	4480
	18	125	609	7050	2590

We can now imagine our experiments, but this time using Solver. We compare column $ind1S$ in Table 2 with columns $mis1S$, $mis2S$, and $mis3S$ in Table 3. Looking at Table 5 we see that the maximality constraint degrades performance, sometimes by a factor of ten. Our Solver experiments would suggest that the maximality constraint should be avoided, at all costs! Therefore, we can replicate our empirical study, changing only the implementation toolkit and reach an entirely different conclusion.

5 Why Was There a Difference in the Choco Encodings?

Why should there have been such a dramatic difference in the performance of the $sumVars$ and $occur$ constraints in Choco? The following is an explanation due to Francois Laburthe, one of the authors of Choco:

> The constraint propagation phase in Choco is based on a dual event queueing mechanism. The filtering algorithm of a constraint can be implemented in two ways, either as one general revision procedure reaching consistency from any state or as parametrised procedure that reach consistency after a given domain reduction on a given variable. The first behaviour corresponds to the general description of arc consistency algorithms for constraints specified by list of feasible tuples; the second corresponds to specialised propagation patterns inspired from rule-based systems. Choco features two pools of pending events: domain updates and constraint revisions. The first pool is of higher priority, i.e. whenever all immediate propagation after the domain updates have been done, the pending constraints are awoken.
> The implementation used in this study propagates linear constraints as *delayed constraints* (generic revision procedure) This was motivated by the fact that one linear pass is enough to reach bounds consistency, no matter the number of variables whose domain have been reduced since the former AC state. This turns out to be too slow on the examples above for the following reasons: (a) time is wasted checking whether there are pending variable event before popping each constraint event (b) it seems that this leads to more constraint checks in infeasible situations. Choco has now been modified in the light of these results. The change consisted in altering the management of linear constraints: up to a certain number of variables (set by default to 8), they are propagated through the *variable event mechanism*. When they involve more, they keep being propagated in the *constraint revision event mechanism*.

With these changes in place, Choco's performance is now no more than 6 times worse than Solver over the data sets presented here. We can see in some cases it is a close competitor. For example in Table 2, if we compare columns $ind2S$ and $ind2C$ we see that both toolkits run in roughly the same time, even though Choco is on a 755MHz machine and Solver is on a 933Hz machine.

6 Conclusion

We have examined two constraints, the biconditional and a restriction on the sum of 0/1 variables. The various implementations presented are logically equivalent and result in the same exploration of the search space. Yet the run times are often very different. For example, one encoding was reliably twenty times faster than another. Unfortunately, this did not translate across programming toolkits. We saw that summation was faster than counting occurrences in Ilog Solver, yet it was the other way round in Choco. Even more notable was that a good encoding of the biconditional in Solver corresponded to the worst in Choco!

Why are there differences in run times when we make these subtle reformulations? They have different types of constraints and different numbers of constraints. The performance of arc consistency can be affected by the order that constraints are processed (see for example [3] or [11]). This is one explanation. The other is most obviously down to the way the toolkit providers have implemented the various constraints in the first place (for example, section 5).

What lessons can we learn? First, what we learn with one toolkit might not carry over to another. Therefore, we need to exercise caution when picking up a new tool. Second, when we have an implementation we should not let it rest there; we should investigate other logically equivalent implementations. That is, we should experiment. We often explore different ways of formulating a problem, maybe reaching a conclusion that one formulation is better than the other, measured as run time. Before we do this, we should make sure that we have explored the finest level of formulation, the actual implementation. And finally, as in other branches of science we should be encouraged to replicate results, just as we have done here using two toolkits. Obviously there is value in doing this.

Acknowledgements. We would like to thank our reviewers, ILOG (our collaborators on this project), the OCRE team for giving us Choco, Alice Miller for introducing us to these problems, and Francois Laburthe. Francois gave us the proof in subsection 2.1 and the explanation in 5. We would also like to thank our friends and colleagues in the APES research group. Our work here might be considered as a continuation of our cautionary tales of the empirical analysis of algorithms, i.e. *How not to do it* [4].

References

1. CHOCO. http://www.choco-constraints.net/ home of the Choco Constraint Programming System.
2. Michael. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
3. Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Paul Shaw, and Toby Walsh. The constrainedness of arc consistency. In *Principles and Practices of Constraint Programming*, pages 327–340, 1997.
4. I.P. Gent and T. Walsh. How Not To Do It. *APES Technical Report*. 1995. http://www.dcs.st-and.ac.uk/∼apes/1995.html
5. ILOG. http://www.ilog.com.
6. R. Mathon and A. Rosa. Tables of parameters of bibds with r $\leq$ 41 including existence, enumeration, and resolvability results. *Annals of Discrete Mathematics*, 26:275–308, 1985.
7. P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129:133–163, 2001.
8. Nina Mishra and Leonard Pitt. Generating all maximal independent sets of bounded-degree hypergraphs. In *Tenth Annual Conference on Computational Learning Theory*, pages 211–217, 1997.
9. S. D. Prestwich. Balanced incomplete block design as satisfiability. In *12th Irish Conference on Artificial Intelligence and Cognitive Science*, 2001.
10. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
11. Rick Wallace. Why ac3 is almost always better than ac4 for establishing arc consistency in csp's. In *IJCAI-93*, pages 239–245, 1993.
12. R. Weigel and C. Bliek. On reformulation of constraint satisfaction problems. In *Proceedings of ECAI-98*, pages 254–258, 1998.

A Local Search Algorithm for Balanced Incomplete Block Designs

Steven Prestwich

Cork Constraint Computation Centre
Department of Computer Science
University College, Cork, Ireland
s.prestwich@cs.ucc.ie

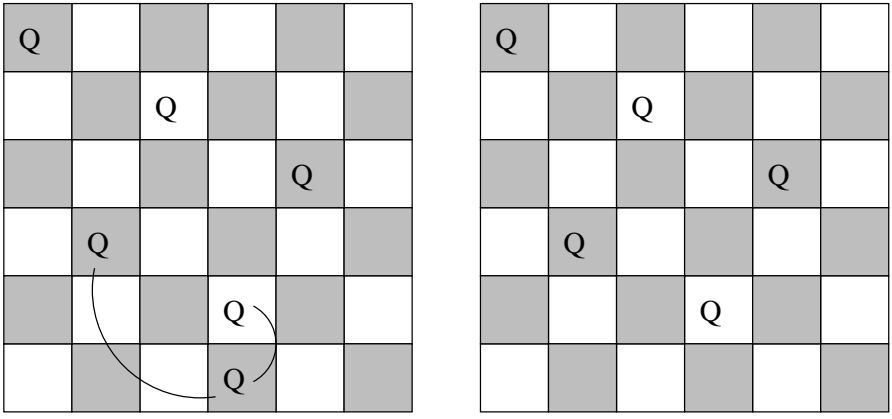
Abstract. Local search is often able to solve larger problems than systematic backtracking. To apply it to a constraint satisfaction problem, the problem is often treated as an optimization problem in which the search space is the set of total assignments, and the number of constraint violations is to be minimized to zero. Though often successful, this approach is sometimes unsuitable for structured problems with few solutions. An alternative is to explore the set of consistent partial assignments, minimizing the number of unassigned variables to zero. A local search algorithm of this type violates no constraints and can exploit cost and propagation techniques. This paper describes such an algorithm for balanced incomplete block design generation. On a large set of instances it out-performs several backtrackers and a neural network with simulated annealing.

1 Introduction

Local search has proved very successful on many combinatorial problems. It is incomplete and (usually) does not exploit cost and constraint reasoning, but its good scalability sometimes makes it indispensable. The natural way to apply local search to optimization problems is to explore the legal solutions and minimize the given cost function. This is impractical for constraint satisfaction problems in which legal solutions are hard to find. For such problems local search is therefore usually applied to the space of total assignments, and the objective function to be minimized is the number of constraint violations; if this becomes zero then a solution has been found.

As an example of a pure constraint satisfaction problem, consider the N-queens problem with a popular model: N variables each taking a value from the integer domain $\{1, \ldots, N\}$, each variable corresponding to a queen and row, and each value to a column. The constraints for this problem are that no two queens may attack each other (a queen attacks another if they lie on the same row, column or diagonal). The usual local search approach is to explore a space of total assignments, that is with all N queens placed somewhere on the board. Figure 1(i) shows a total assignment containing two constraint violations: the last queen can attack two others and vice-versa (attack is symmetrical). We may try to remove these violations, at the risk of introducing new ones, by repositioning a queen involved in a conflict — that is by reassigning a variable to another value. This is the idea behind most local search algorithms for constraint satisfaction, which has been

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 132–143, 2003.



(i) total assignment (ii) consistent partial assignment

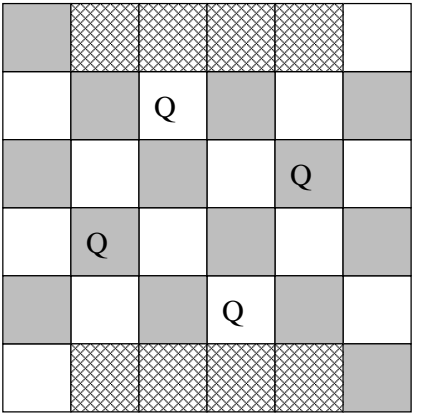
Fig. 1. Local search spaces for N-queens

highly successful on many problems; for example Min-Conflicts Hill Climbing [6] for binary CSPs, GSAT [17] for Boolean satisfiability, and many more recent algorithms.

However, a non-computer scientist (or at least a non-constraint programmer) might design a very different form of local search for N-queens: begin placing queens randomly in non-attacked positions; when a queen cannot be placed, randomly remove one or more placed queens; continue until all queens are placed. The states of this algorithm correspond to consistent partial assignments in our model, as shown in Figure 1(ii). We may add and remove queens randomly, or bias the choice using heuristics. This algorithm explores a different space but is still local search. We may view the number of unassigned variables (unplaced queens) as the cost function to be minimized. No queens can be added to the state in Figure 1(ii), which is therefore a local minimum under this function. This approach is taken manually by dispatch schedulers [4]. Another usage is the IMPASSE class of graph colouring algorithms [7], where a consistent partial assignment is called a *coloration neighborhood*.

An advantage of this type of local search is that constraints are never violated, avoiding the need for weighted sums of cost and infeasibility functions: local search can concentrate its efforts on feasibility. For example in [14] local search was performed on partial assignments consistent under cost constraints obtained from a relaxation of an optimization problem. Upper and lower cost bounds were maintained in exactly the same way as in branch-and-bound but scalability was improved. Furthermore, it found optimal solutions while local search on the legal solutions (minimizing the given cost function) failed to do so.

Another advantage is that this form of local search can be integrated with (at least some forms of) constraint propagation in the tightest possible way: propagation prunes states from the search space, as in many backtracking algorithms. In [10] coloration neighborhoods were restricted to those consistent under forward checking, with im-



(i) FC-consistent partial assignment

0	2	2	2	2	0
1	1	0	3	1	1
1	3	2	2	0	1
1	0	2	2	3	1
1	1	3	0	1	1
0	2	2	2	2	0

(ii) corresponding conflict counts

Fig. 2. Forward checking in local search

proved results over IMPASSE colouring. Notice that though the state in Figure 1(ii) is consistent, under constraint propagation it is inconsistent: the domain of the variable corresponding to the last row is empty. Local search enhanced with constraint propagation prunes such states. Removing the queen on row 1 makes it FC-consistent (consistent under forward checking) as shown in Figure 2(i); squares on free rows that are under attack by at least one queen are shaded, and both free rows contain empty squares. However, this state cannot be extended while maintaining FC-consistency (placing a queen in column 1 or 6 on row 1 or 6 causes all squares to be under attack on the other row) so it is a local minimum.

To update variable domains while applying local search, we maintain a *conflict count* for each variable-value pair. The conflict counts for the state in Figure 2(i) are shown in Figure 2(ii). A conflict count denotes the number of constraints that would be violated if the corresponding assignment were made. (This is distinct from local search algorithms in which conflicts *do* occur and are counted.) They are computed incrementally on [un]assigning a variable. Notice that the number of attacking queens *on other rows* are counted (each potential attack corresponds to a binary constraint that would be violated), and that the queens are placed on squares with zero conflict counts (corresponding to values that have not been deleted from domains by forward checking). Conflict counts can be generalized to non-binary constraints: they were used for unit propagation in a local search algorithm for SAT in [14], generalized to propagation on 0/1 variables with linear constraints in [11], and used to maintain arc-consistency in local search in [12].

Standard variable ordering heuristics can be used, for example selecting the variable with smallest domain. An additional benefit of conflict counts is that we can obtain a domain size for assigned variables as well as unassigned ones. In the example of Figure 2 the variables for rows 2–5 each have domain size 1, but in general these may be different. This can be used to guide the selection of variables for unassignment, for example the

variable with *greatest* domain size. These techniques are explored in [9], where such an algorithm is shown to require even fewer search steps than Min-Conflicts Hill-Climbing to solve N-queens problems.

Because this approach performs local search in a constrained space, we call it Constrained Local Search (CLS). Alternatively it can be viewed as a non-systematic form of backtracking, which we call Incomplete Dynamic Backtracking. The relationships between CLS and other search algorithms are discussed in other papers [9,10,14]. In the next section we describe a CLS implementation for BIBD generation.

2 Constrained Local Search for BIBD Generation

BIBD generation is a standard combinatorial problem, originally used in the statistical design of experiments but since finding other applications such as cryptography. A BIBD is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. Another way of defining a BIBD is in terms of its *incidence matrix*, which is a binary matrix with v rows, b columns, r ones per row, k ones per column, and scalar product λ between any pair of distinct rows. A BIBD is therefore specified by its parameters (v, b, r, k, λ). An example is shown in Figure 3. It can be proved that for a BIBD to exist its parameters must satisfy the conditions $rv = bk$, $\lambda(v-1) = r(k-1)$ and $b \geq v$, but these are not sufficient conditions. Constructive methods can be used to design BIBDs of special forms, but the general case is very challenging and there are surprisingly small open problems, the smallest being (22,33,12,8,4). One source of intractability is the large number of symmetries: given any solution, any two rows or columns may be exchanged to obtain another solution. A survey of known results is given in [2] and some references and instances are given in CSPLib (problem 28).[1]

$$\begin{bmatrix} 1&0&1&1&1&0&0&0&0&1 \\ 0&0&1&1&0&1&1&0&1&0 \\ 1&1&0&1&0&0&0&1&1&0 \\ 0&0&0&0&1&0&1&1&1&1 \\ 0&1&1&0&0&1&0&1&0&1 \\ 1&1&0&0&1&1&1&0&0&0 \end{bmatrix}$$

Fig. 3. A solution to the BIBD instance $(6, 10, 5, 3, 2)$

The most direct CSP model for BIBD generation represents each matrix element by a binary variable $m_{ij} \in \{0, 1\}$. There are three types of constraint: (i) v b-ary constraints for the r ones per row, (ii) b v-ary constraints for the k ones per column, and (iii) $v(v-1)/2$ $2b$-ary constraints for the λ matching ones in each pair of rows. This is the

[1] http://www.csplib.org

constraint model we use for BIBDs, but instead of expressing the constraints directly we express them via the integer variables defined below, giving a description that is closer to the implementation:

– for each row there are two variables

$$\rho_i^0 = card\{j \in \{1 \dots b\} |\ m_{ij} = 0\} \quad (1 \leq i \leq v)$$
$$\rho_i^1 = card\{j \in \{1 \dots b\} |\ m_{ij} = 1\} \quad (1 \leq i \leq v)$$

– similarly for each column there are two variables

$$\chi_j^0 = card\{i \in \{1 \dots v\} |\ m_{ij} = 0\} \quad (1 \leq j \leq b)$$
$$\chi_j^1 = card\{i \in \{1 \dots v\} |\ m_{ij} = 1\} \quad (1 \leq j \leq b)$$

– to handle the scalar products there are two variables for each pair of rows

$$\pi_{ii'}^1 = card\{j \in \{1 \dots b\} |\ m_{ij} = 1 \ \wedge \ m_{i'j} = 1\} \quad (1 \leq i < i' \leq v)$$
$$\pi_{ii'}^0 = card\{j \in \{1 \dots b\} |\ m_{ij} = 0 \ \vee \ m_{i'j} = 0\} \quad (1 \leq i < i' \leq v)$$

The constraints are expressed on these counts:

$$\begin{array}{llll} \rho_i^1 \leq r & \rho_i^0 \leq b - r & (1 \leq i \leq v) \\ \chi_j^1 \leq k & \chi_j^0 \leq v - k & (1 \leq j \leq b) \\ \pi_{ii'}^1 \leq \lambda & \pi_{ii'}^0 \leq b - \lambda & (1 \leq i < i' \leq v) \end{array}$$

Initially all the m_{ij} are unassigned so all counts are zero. It is clear that a total variable assignment satisfying these constraints corresponds to a BIBD. Thus we have a local search algorithm for BIBDs that performs back-checking on all constraints. We now add forward checking, but only on the row and column constraints (it may be added to the scalar product constraints in future work). For each matrix entry m_{ij} and domain value $b \in \{0, 1\}$ a conflict count κ_{ijb} is maintained, which counts the number of constraints that would be violated under the assignment $m_{ij} = b$.

All the counts are updated incrementally. For example on assigning $m_{ij} = 0$, ρ_i^0 and χ_j^0 are incremented. If $\rho_i^0 = b - r$ then $\kappa_{ij'0}$ is incremented for all $m_{ij'}$ (where $j \not= j'$) not currently assigned to 0; and if $\chi_i^0 = v - k$ then $\kappa_{i'j0}$ is incremented for all $m_{i'j}$ (where $i \not= i'$) not currently assigned to 0. On incrementing a conflict count from 0 to 1, the corresponding value is deleted from its domain, and its domain size updated. Counts are similarly incremented on assigning $m_{ij} = 1$ and decremented on unassignment. Note that if incrementing any count would violate a row or column constraint, or if a domain size becomes zero, then the assignment is disallowed and its effects are removed.

By maintaining this information during search we can unassign any assigned variable at any point (or assign any unassigned variable, subject to constraints), allowing very flexible search strategies. It remains only to choose heuristics to guide the search, and we use heuristics similar to those in previous papers on CLS. Variables are selected for smallest domain size, breaking ties randomly. If the selected variable can be assigned a value then assign it, otherwise backtrack. To backtrack we unassign one or more assigned variables, in fact $B \geq 1$ variables where B is an integer *noise parameter* specified by the

		FC-CBJ-DG			FC-CBJ-SVP-VM		NN-SA	CLS	
parameters	vb	solns	nodes	ms	nodes	ms	solved	back	ms
7 7 3 3 1	49	50	21	1.4	21	4	yes	61	0
6 10 5 3 2	60	50	60	3.6	30	6	yes	141	1
7 14 6 3 2	98	50	2152	130	44	12	yes	293	3
9 12 4 3 1	108	50	40	1.8	48	10	yes	386	7
6 20 10 3 4	120	18	435	3700	62	35	yes	284	3
7 21 9 3 3	147	16	2877	4300	69	44	yes	364	6
6 30 15 3 6	180	6	196	9900	95	140	yes	430	9
7 28 12 3 4	196	11	195	7600	86	120	yes	502	13
9 24 8 3 2	216	44	763	1200	77	84	yes	536	23
6 40 20 3 8	240	3	156	1700	126	390	yes	457	214
7 35 15 3 5	245	6	230	1500	109	270	yes	444	18
7 42 18 3 6	294	6	141	1900	133	480	yes	619	32
10 30 9 3 2	300	38	181	3400	123	200	yes	1518	55
6 50 25 3 10	300	1	1057	31000	156	830	yes	517	32
9 36 12 3 3	324	29	478	6800	173	380	yes	1623	47
13 26 6 3 1	338	50	1076	350	145	170	yes	4548	154
7 49 21 3 7	343	2	151	33000	163	790	yes	578	38
6 60 30 3 12	360	2	139	46000	185	1500	yes	487	29
7 56 24 3 8	392	1	36401	46000	173	1200	yes	1604	50
6 70 35 3 14	420	0	0	54000	217	2300	yes	1602	56
9 48 16 3 4	432	19	685	16000	152	730	yes	1631	81
7 63 27 3 9	441	0	0	6000	193	1700	yes	1606	63
8 56 21 3 6	448	5	285	37000	323	2000	yes	1645	80
6 80 40 3 16	480	0	0	72000	246	3600	yes	1569	69

Fig. 4. Results with $vb < 1400$ and $k = 3$ (1 of 2)

user at runtime. It is called a noise parameter because it plays the same role as noise in more standard forms of local search: to escape from local minima. For unassignment the reverse heuristic is used: select variables for greatest domain size, breaking ties randomly. Values are chosen with a preference for the last value used (initialized randomly), but to avoid stagnation a small random factor is added.

In the next section we evaluate this algorithm on a large number of BIBD instances, and compare its performance with published results.

3 Experimental Results

As noted in [1] most combinatorics work on BIBDs focuses on unsolved problems and few papers collect a large set of computational results, though several authors give results on a small number of instances. We compare results for CLS with the only extensive published results we know of for BIBDs. From [5] we obtain results for: (i) forward checking with conflict-directed backjumping and variable ordering based on domain size and degree (FC-CBJ-DG) [15]; (ii) the same algorithm enhanced with heuristics

		FC-CBJ-DG			FC-CBJ-SVP-VM		NN-SA	CLS	
parameters	vb	solns	nodes	ms	nodes	ms	solved	back	ms
7 70 30 3 10	490	1	235	67000	216	2400	no	1621	81
15 35 7 3 1	525	48	395	980	193	390	yes	12586	548
12 44 11 3 2	528	41	591	5100	204	670	yes	6639	285
7 77 33 3 11	539	0	0	93000	240	3300	no	2593	106
9 60 20 3 5	540	12	386	29000	238	1600	yes	2603	133
7 84 36 3 12	588	1	1027	92000	254	4200	yes	2585	112
10 60 18 3 4	600	12	613	26000	188	1500	no	4594	204
11 55 15 3 3	605	33	680	12000	229	1300	yes	5559	234
7 91 39 3 13	637	0	0	130000	277	5400	no	3564	165
9 72 24 3 6	648	8	671	42000	309	3000	no	4628	201
13 52 12 3 2	676	43	298	4600	1008	3400	yes	8541	317
9 84 28 3 7	756	8	2054	54000	257	4200	no	4594	229
9 96 32 3 8	864	9	3997	66000	295	6400	no	5549	326
10 90 27 3 6	900	8	3131	56000	286	5300	no	8419	502
9 108 36 3 9	972	3	1193	96000	341	40000	no	8507	501
13 78 18 3 3	1014	37	1392	16000	280	3500		18454	1099
15 70 14 3 2	1050	36	1647	23000	1058	5700		25472	1501
12 88 22 3 4	1056	33	1271	28000	296	5000		15486	945
9 120 40 3 10	1080	6	10429	110000	461	14000		11471	658
19 57 9 3 1	1083	46	778	4800	745	6900		64528	3912
10 120 36 3 8	1200	4	9927	110000	377	13000		13581	896
11 110 30 3 6	1210	24	2491	49000	366	36000		13440	914
16 80 15 3 2	1280	40	2275	23000	490	4700		39486	2812
13 104 24 3 4	1352	30	1076	49000	344	8500		24507	1815

Fig. 5. Results with $vb < 1400$ and $k = 3$ (2 of 2)

for exploiting problem symmetries (FC-CBJ-SVP-VM). Both were executed on a Sun Ultra 60 with 360MHz. From [1] we obtain results for a neural network with simulated annealing as a relaxation strategy, which we shall refer to as NN-SA. This solved more BIBD instances than two other relaxation strategies: simple hill climbing and a novel strategy called parallel mean search. It was executed on a Connection Machine CM-200 with 2048 processors, and because of the parallel platform no execution times were given. CLS was executed on a 300 MHz DEC Alphaserver 1000A 5/300 under Unix.

First we consider the solvable instances with $vb < 1400$ and $k = 3$. Results are shown in Figures 4 and 5, where mean CPU times are given in milliseconds, the mean number of search tree nodes is shown for the backtrackers, and the mean number of backtracks shown for CLS with noise level 2. All algorithms except NN-SA were executed 50 times per instance with different random seeds, and column *solns* shows in how many runs a solution was found in the time limit (hence the long execution times in cases where few or no solutions were found). FC-CBJ-SVP-VM solves almost all instances, failing only once on 3 instances, whereas NN-SA fails on several instances and FC-CBJ-DG fails more often. CLS is the most successful algorithm, solving all instances 50 times and

		NN-SA	CLS		
parameters	vb	solved	noise	backtracks	sec
8 14 7 4 3	112	yes	2	282	0.0
11 11 5 5 2	121	yes	2	2544	0.04
10 15 6 4 2	150	yes	2	2852	0.04
9 18 8 4 3	162	yes	2	2714	0.05
13 13 4 4 1	169	yes	2	674	0.01
10 18 9 5 4	180	yes	2	5172	0.12
8 28 14 4 6	224	yes	2	2408	0.06
15 15 7 7 3	225	yes	2	30488	0.61
11 22 10 5 4	242	yes	2	47868	1.2
16 16 6 6 2	256	yes	2	25662	0.54
12 22 11 6 5	264	no	2	48516	1.23
10 30 12 4 4	300	yes	2	3632	0.12
16 20 5 4 1	320	yes	3	6771	0.16
9 36 16 4 6	324	yes	2	7316	0.21
8 42 21 4 9	336	no	2	8004	0.25
13 26 8 4 2	338	yes	2	13552	0.5
13 26 12 6 5	338	no	2	283418	7.93
10 36 18 5 8	360	no	2	30432	1.08
19 19 9 9 4	361	no	1	320158	9.73
11 33 15 5 6	363	no	2	51228	1.79
14 26 13 7 6	364	no	—	no	—
16 24 9 6 3	384	no	1	320052	9.8
12 33 11 4 3	396	yes	3	11706	0.33
21 21 5 5 1	441	yes	3	9576	0.22
8 56 28 4 12	448	no	2	3072	0.12
10 45 18 4 6	450	no	2	2314	0.08
15 30 14 7 6	450	no	—	no	—
16 30 15 8 7	480	no	—	no	—
11 44 20 5 8	484	no	2	31076	1.13

Fig. 6. Results with $vb < 1000$ and $k \neq 3$ (1 of 3)

taking a fraction of the time of the enhanced backtracker (assuming comparable speeds for the two machines).

It is interesting to note that on some instances CLS takes many more search steps than the backtrackers to find a solution but is much faster. Though it exploits backtrack-style techniques such as forward checking, as a local search algorithm it can recover from mistakes quickly (at least in principle); a backtracker may take an exponential time to recover from a poor decision high in the search tree, so it must do more work at each step to ensure good choices. CLS can therefore afford to use simpler techniques than a backtracker. In this case CLS does not perform symmetry breaking, which probably accounts for the different in speed. The use of symmetry breaking could be added to CLS but other work suggests that this will not improve the results. In previous papers [8,13] the addition of symmetry breaking constraints was found to increase the number of local search steps on several problems (including BIBD generation with a different

parameters	vb	NN-SA solved	CLS noise	backtracks	sec
9 54 24 4 9	486	no	3	5739	0.21
13 39 12 4 3	507	no	3	33426	1.22
13 39 15 5 5	507	no	2	257178	11.2
16 32 12 6 4	512	no	—	no	—
15 35 14 6 5	525	no	—	no	—
12 44 22 6 10	528	no	2	647678	27.9
23 23 11 11 5	529	no	—	no	—
10 54 27 5 12	540	no	2	24100	0.98
8 70 35 4 15	560	no	3	3750	0.13
17 34 16 8 7	578	no	—	no	—
10 60 24 4 8	600	no	2	14418	0.75
11 55 20 4 6	605	no	3	17787	0.8
11 55 25 5 10	605	no	2	86634	4.45
18 34 17 9 8	612	no	—	no	—
25 25 9 9 3	625	no	—	no	—
15 42 14 5 4	630	no	1	690050	42.9
21 30 10 7 3	630	no	—	no	—
16 40 10 4 2	640	no	2	89090	4.22
16 40 15 6 5	640	no	—	no	—
9 72 32 4 12	648	no	2	10262	0.54
15 45 21 7 9	675	no	—	no	—
13 52 16 4 4	676	no	3	89193	3.76
13 52 24 6 10	676	no	2	1133166	49.7
10 72 36 5 16	720	no	3	28620	1.33
19 38 18 9 8	722	no	—	no	—
11 66 30 5 12	726	no	2	28148	1.52
22 33 12 8 4	726	no	—	no	—
15 52 26 7 12	780	no	—	no	—

Fig. 7. Results with $vb < 1000$ and $k \neq 3$ (2 of 3)

model). The symmetry breaking approach of [5] does not add constraints to the model at the start of the search so these results may not apply, but they indicate that symmetries are not necessarily harmful for local search.

Next we consider the solvable instances with $vb < 1000$ and $k \not= 3$ Results are shown for CLS and NN-SA in Figures 6, 7 and 8. These problems are considerably harder, so CLS was executed three times per instance with noise levels 1, 2 and 3. The best result is reported in each case, "no" denoting that no solution was found after 2,000,000 backtracks. All instances solved by NN-SA were also solved by CLS. NN-SA solves 18/86 instances while CLS solves 54/86. Including the 39 instances with $k = 3$ and $vb < 1000$ from the earlier tables, NN-SA solves 54/125 instances (43.2%) while CLS solves 93/125 (74.4%).

In a previous paper [11] the experiments on the easier instances ($vb < 1400$, $k = 3$) were performed using a version of CLS for 0/1 integer programs. A 0/1 model of BIBD generation was used to encode the problems that introduced an auxiliary variable for

		NN-SA	CLS		
parameters	vb	solved	noise	backtracks	sec
27 27 13 13 6	729	no	—	no	—
21 35 15 9 6	735	no	—	no	—
10 75 30 4 10	750	no	3	21420	0.92
25 30 6 5 1	750	yes	2	361682	14.3
20 38 19 10 9	760	no	—	no	—
16 48 15 5 4	768	no	1	694455	43.2
16 48 18 6 6	768	no	—	no	—
12 66 22 4 6	792	no	1	22626	1.38
12 66 33 6 15	792	no	1	199475	12.8
9 90 40 4 15	810	no	2	12862	0.69
13 65 20 4 5	845	no	2	23760	1.4
11 77 35 5 14	847	no	1	97508	5.11
21 42 10 5 2	882	no	—	no	—
21 42 12 6 3	882	no	—	no	—
21 42 20 10 9	882	no	—	no	—
16 56 21 6 7	896	no	—	no	—
10 90 36 4 12	900	no	2	20348	0.99
15 60 28 7 12	900	no	—	no	—
18 51 17 6 5	918	no	—	no	—
22 42 21 11 10	924	no	—	no	—
15 63 21 5 6	945	no	1	423463	30.7
16 60 15 4 3	960	no	1	187646	6.31
16 60 30 8 14	960	no	—	no	—
31 31 6 6 1	961	yes	2	121008	2.04
31 31 10 10 3	961	no	—	no	—
31 31 15 15 7	961	no		no	
11 88 40 5 16	968	no	1	61386	4.07
22 44 14 7 4	968	no	—	no	—
25 40 16 10 6	1000	no	—	no	—

Fig. 8. Results with $vb < 1000$ and $k \neq 3$ (3 of 3)

each pair of matrix rows. This algorithm performed similarly to the enhanced backtracker, and considerably more slowly than the specialized implementation described here. Several other recent papers mention BIBD generation. In [13] five instances are considered: (7,7,3,3,1), (6,10,5,3,2), (7,14,6,3,2), (9,12,4,3,1) and (8,14,7,4,3). These are SAT-encoded and passed to three SAT algorithms using backtracking, local search and a hybrid (a SAT implementation of CLS). Computational results were unexpectedly poor, the fifth instance being unsolved after several hundred seconds by any SAT algorithm. In [16] the same five instances are considered, with better results. Using six different encodings in the Choco constraint system, all are solved in 10–3670 ms. In [3] a matrix model is used to break symmetries, and all non-symmetrical solutions to (8,14,7,4,3) are found in 238 seconds.

4 Conclusion

This paper described and tested a Constrained Local Search (CLS) algorithm for the problem of BIBD generation. Previous papers have argued that this form of local search is well-suited to large, structured combinatorial problems with few solutions. It combines the scalability of local search with the space-pruning ability (but not the completeness) of constraint programming. The BIBD results support this view: CLS outperformed existing backtrack-based approaches, and solved more instances than another local search algorithm (a neural network with simulated annealing).

Is a specialised algorithm necessary? In a previous paper [11] a CLS algorithm for linear pseudo-Boolean was applied to BIBD generation (among other problems) and gave results comparable to those of the best backtracker of [5]. The algorithm reported in this paper is much faster, showing that a specialised CLS algorithm can greatly outperform a general-purpose CLS algorithm on a given problem.

It would be nice to report the solution of at least one open problem. So far we have not achieved this (of course they may all be unsolvable) but will attempt to do so in future work. To improve the algorithm, forward checking could be applied to the scalar product constraints. Alternatively the linear pseudo-Boolean implementation of CLS could be extended to handle non-linear constraints, so that no auxiliary variables are necessary. However, both approaches would need $O(v^2b)$ memory whereas the algorithm in this paper needs only $O(vb)$, making it well-suited to very large instances.

Acknowledgments. This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

References

1. P. Bofill, C. Torras. Neural Cost Functions and Search Strategies for the Generation of Block Designs: an Experimental Evaluation. *International Journal of Neural Systems* vol. 11 no. 2, World Scientific Publishing Company, 2001, pp. 187–202.
2. C. J. Colbourn, J. H. Dinitz (eds.). The CRC Handbook of Combinatorial Designs. CRC Press, 1996.
3. P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Matrix Modelling. *Workshop on Modelling and Problem Formulation*, Cyprus, 2001.
4. M. Y. Loenko. A Non-Return Search Algorithm. *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, le Croisic, France, 2002, pp. 251–259.
5. P. Meseguer, C. Torras. Exploiting Symmetries Within Constraint Satisfaction Search. *Artificial Intelligence* vol. 129 no. 1–2, 2001, pp. 133–163.
6. S. Minton, M. D. Johnston, A. B. Philips, P. Laird. Minimizing Conflicts: A Heuristic Repair Method For Constraint Satisfaction and Scheduling Problems. *Constraint-Based Reasoning*, Freuder & Mackworth (Eds.), 1994.
7. C. Morgenstern. Distributed Coloration Neighborhood Search. D. S. Johnson and M. A. Trick (eds.), *Cliques, coloring and satisfiability: second DIMACS implementation challenge, DIMACS series in discrete mathematics and theoretical computer science* vol. 26, American Mathematical Society 1996, pp. 335–357.

8. S. Prestwich. Negative Effects of Modeling Techniques on Search Performance. *Annals of Operations Research* (to appear).
9. S. Prestwich. Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search. *Annals of Operations Research* (to appear).
10. S. Prestwich. Coloration Neighborhood Search With Forward Checking. *Annals of Mathematics and Artificial Intelligence* vol. 34 no. 4, 2002, Kluwer Academic Publishers, pp. 327–340.
11. S. Prestwich. Randomised Backtracking for Linear Pseudo-Boolean Constraint Problems. *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, le Croisic, France, 2002, pp. 7–19.
12. S. Prestwich. Maintaining Arc-Consistency in Stochastic Local Search. *Workshop on Techniques for Implementing Constraint Programming Systems*, Ithaca, NY, 2002.
13. S. Prestwich. First-Solution Search with Symmetry Breaking and Implied Constraints. *Workshop on Modelling and Problem Formulation*, Paphos, Cyprus, 2001.
14. S. Prestwich. A Hybrid Search Architecture Applied to Hard Random 3-SAT and Low-Autocorrelation Binary Sequences. *The Sixth International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science vol. 1894, Springer-Verlag 2000, pp. 337–352.
15. P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* vol. 9 no. 3, 1993, pp. 268–299.
16. P. Prosser, E. Selensky. On the Encoding of Constraint Satisfaction Problems with 0/1 Variables. *Workshop on Modelling and Problem Formulation*, Cyprus, 2001.
17. B. Selman, H. Levesque, D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, MIT Press 1992, pp. 440–446.

The Effect of Nogood Recording in DPLL-CBJ SAT Algorithms

Inês Lynce and João Marques-Silva

Technical University of Lisbon,
IST/INESC/CEL, Lisbon, Portugal
{ines,jpms}@sat.inesc.pt

Abstract. Propositional Satisfiability (SAT) solvers have been the subject of remarkable improvements in the last few years. Currently, the most successful SAT solvers share a number of similarities, being based on backtrack search, applying unit propagation, and incorporating a number of additional search pruning techniques. Most, if not all, of the search reduction techniques used by state-of-the-art SAT solvers have been imported from the Constraint Satisfaction Problem (CSP) domain and, most significantly, include forms of backjumping and of nogood recording. This paper proposes to investigate the actual usefulness of these CSP techniques in SAT solvers, with the objective of evaluating the actual role played by each individual technique.

1 Introduction

The areas of Constraint Satisfaction Problem (CSP) and Propositional Satisfiability (SAT) have been the subject of intensive research in recent years, with significant theoretical and practical contributions. In the area of SAT, several highly optimized solvers have been developed [3,11,16,17,26]. These state-of-the-art SAT solvers can now very easily solve very large, very hard real-world problem instances, which more traditional SAT solvers are totally incapable of. All of these highly effective SAT solvers are based on improvements made to the original Davis-Putnam-Logemann-Loveland (DPLL) backtrack search SAT algorithm [6]. Such improvements range from new search strategies, to new search pruning and reasoning techniques, and to new fast implementations.

Moreover, the relationship between SAT and CSP has become apparent due to an increasing number of mappings between SAT and CSP that have recently been proposed [10,24]. These different encodings have shed new insights on solving hard instances of CSP. Moreover, such results motivate a better understanding of the actual usefulness of the CSP techniques that have been utilized in successful SAT solvers.

Regarding different algorithmic solutions for SAT, and despite the potential theoretical and practical interest of all of them, we believe backtrack search to be the most robust approach for solving hard, structured, real-world instances of SAT. This belief has been amply supported by extensive experimental evidence obtained in recent years [3,11,16,17,26]. Moreover, the most effective algorithms

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 144–158, 2003.

are complete, and so able to prove what local search is not capable of, i.e. unsatisfiability. Indeed, this is often the objective in a large number of significant SAT-related real-world applications.

Most if not all backtrack search SAT algorithms also incorporate propagation techniques for consistency checking, by applying Boolean constraint propagation (BCP) [25]. (Observe that backtrack search with BCP, i.e. DPLL, is conceptually similar to the maintaining arc consistency algorithm (MAC) [21], and equivalent for suitable mappings [1,10,24].) Another strategy for reducing the number of searched nodes consists of performing back jumps in the search tree, skipping portions of the search space that can be shown not to contain a solution. In this context, and whenever a consistency check fails, conflict-directed backjumping (CBJ) [19] enables the search process to safely jump directly to the cause of the conflict.

In addition, state-of-the-art SAT solvers [3,11,16,17,26] effectively use learning techniques. In these solvers, whenever a conflict (*dead-end*) is reached, a new clause (*nogood*) is recorded to prevent the occurrence of the same conflict again during the subsequent search. Moreover, and from the first SAT solvers that incorporated non-chronological backtracking (NCB) [3,16], learning has always been a key component of the search algorithm, where recorded nogoods are used to determine the search point to backtrack to.

Hence, it is certainly relevant to conduct an unbiased evaluation of the isolated usefulness of DPLL-CBJ and of learning on a successful SAT solver. This work will allow us to find out whether a DPLL-CBJ SAT algorithm is enough *per se*, or the algorithm should definitely include nogood learning techniques. Therefore, the objectives of this paper are two-fold. First, to describe the organization of a DPLL-CBJ SAT algorithm. Second, to evaluate the effect of learning in this algorithm. For this purpose, we developed a general framework that implements a DPLL-CBJ SAT algorithm *with* and *without* nogood recording. Moreover, we evaluate the performance on a representative set of instances, obtained from different real-world problems. This evaluation allows us to confirm and extend the preliminary experimental results presented in [2].

The remainder of the paper is organized as follows. Next, we introduce definitions used throughout the other sections. Moreover, we provide a historical perspective regarding the evolution of both CSP and SAT. Afterwards, we briefly describe chronological backtrack (CB) algorithms for SAT. In Section 4 we overview non-chronological backtracking (NCB) SAT algorithms, and further relate them with DPLL-CBJ. Experimental results are given in Section 5, and finally Section 6 concludes the paper.

2 Background

This section introduces the notational framework used throughout the paper, for CSP and SAT. Moreover, we provide a historical perspective for both areas.

2.1 CSP

A CSP consists of a set of variables V and a set of constraints C. Each variable $v \in V$ has a domain of values M_v of size m_v. Each a-ary constraint $c \in C$ restricts a tuple of variables $\langle v_1, ..., v_a \rangle$ to an allowed combination of simultaneous values for the variables in the tuple. In a binary CSP, each constraint is a relation between two variables. Any binary CSP can be associated with a *constraint graph*, where the nodes represent variables and each edge links a pair of nodes if and only if there is a constraint on the corresponding variables. A CSP consists of deciding whether there exists a (*consistent*) assignment to the variables such that all the constraints are satisfied, i.e. no $c \in C$ is violated.

In tree search algorithms for CSP the variables are incrementally instantiated with values from their respective domains. In chronological backtrack (CB), when a value is assigned to a variable, *consistency checking* is performed backwards against the already instantiated variables. If a conflict (*dead-end*) is reached, the algorithm backtracks to the most recent (not *wiped-out*[1]) assigned variable, changes its assignment and continues from there.

Trying to improve the performance of backtrack search entails deciding how far to backtrack and also recording the reasons for the dead-end in the form of new constraints (*nogoods*). The idea of going back several levels in a dead-end situation, rather than going back to the chronologically most recent decision, was exploited independently in [9], where the term backjumping (BJ) was introduced, and in [23] as a part of the dependency-directed backtracking algorithm. Another example is Dechter's graph-based backjumping (GBJ) [8] that proposes to jump back to the source of the failure by using knowledge extracted from the constraint graph. In addition, conflict-directed backjumping (CBJ) [19] consists of keeping a *set* of past variables that failed consistency checks with each variable, based on dependencies from the constraints.

Arc-consistency (AC) [15] is a polynomial propagation algorithm commonly used in CSP. A state is arc-consistent if every variable has a value in its domain that is consistent with each of the constraints on that variable. Arc consistency can be achieved by successive deletion of values that are inconsistent with some constraint. As values are deleted, other values may become inconsistent because they relied on the deleted values. Arc consistency therefore exhibits a form of *constraint propagation*, as choices are gradually narrowed down. Furthermore, *maintaining arc consistency* (MAC) [21] is a solution procedure which incorporates and further maintains arc consistency during backtrack search. In addition, MAC can be improved by adding conflict-directed backjumping (CBJ), thus obtaining the algorithm MAC-CBJ [20].

2.2 SAT

In a SAT problem, propositional variables are denoted $x_1, \ldots, x_n$, and can be assigned truth values 0 (or F) or 1 (or T). The truth value assigned to a variable x

[1] A variable domain is wiped-out when *all* values have been tried for that variable without success.

is denoted by $\nu(x)$. (When clear from context we use $x = \nu_x$, where $\nu_x \in \{0, 1\}$). A literal l is either a variable x_i or its negation $\neg x_i$. A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses. A clause is said to be *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all of its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned truth value are said to be *free literals*. A formula is said to be *satisfied* if all its clauses are satisfied, and is *unsatisfied* if at least one clause is unsatisfied. A *truth assignment* A for a formula is a set of assigned variables and their corresponding truth values. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

Over the years a large number of algorithms has been proposed for SAT, from the original Davis-Putnam procedure (DP) [7], to recent backtrack search algorithms [3,12,16,17,26], to local search algorithms [22], among many others. Among the different algorithms, we believe backtrack search to be the most robust approach for solving hard, structured, real-world instances of SAT. Observe that only complete algorithms can establish unsatisfiability if given enough CPU time, which is often a requirement when considering real-world instances.

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland (DPLL) [6]. Moreover, non-chronological backtracking strategies (NCB) attempt to identify the variable assignments causing a conflict and backtrack directly to a point so that at least one of those variable assignments is modified. GRASP [16] and relsat [3] are examples of SAT solvers that successfully implement non-chronological backtracking.

Recent state-of-the-art SAT solvers are also characterized by using very efficient data structures, intended to reduce the CPU time required per each node in the search tree. Examples of efficient lazy data structures include the head/tail lists used in SATO [26] and the watched literals used in Chaff [17].

3 Chronological Backtrack SAT Algorithms

A backtrack search SAT algorithm is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the n problem variables. Each different truth assignment defines a *search path* within the search space. A *decision level* is associated with each variable selection and assignment. The first variable selection corresponds to decision level 1, and the decision level is incremented by 1 for each new variable decision assignment[2]. When relevant to the context, we use an assignment notation to indicate the decision level at which the variable assignment occurred. Thus, $x = \nu_x@d$ reads as "x is assigned ν_x at decision level d". In addition, and for each decision level, the *unit clause rule* [7] is applied. If a clause is unit, then the sole free literal must be assigned value 1 for the formula to be satisfied. In this case, the value of the literal and of

[2] Observe that all the assignments made before the first decision assignment correspond to decision level 0.

the associated variable are said to be *implied*. The iterated application of the unit clause rule is often referred to as Boolean Constraint Propagation (BCP) [25].

In chronological backtracking (CB), the search algorithm keeps track of which decision assignments have been toggled. Given an unsatisfied clause (i.e. a *conflict* or a *dead end*) at decision level d, the algorithm checks whether at the current decision level the corresponding decision variable x has already been toggled. If not, the algorithm erases the variable assignments which are implied by the assignment on x, including the assignment on x, assigns the opposite value to x, and marks decision variable x as toggled. In contrast, if the value of x has already been toggled, the search backtracks to decision level $d - 1$.

4 Non-chronological Backtrack SAT Algorithms

All of the most effective recent state-of-the-art SAT solvers [3,11,16,17,26] utilize different forms of non-chronological backtracking (NCB). Non-chronological backtracking backs up the search tree to one of the identified causes of failure, skipping over irrelevant variable assignments.

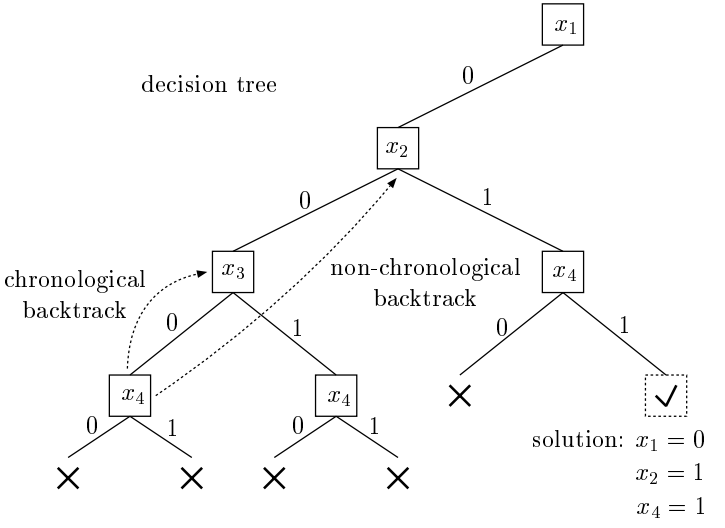


Fig. 1. Non-Chronological Backtracking

For example, let us consider Figure 1, that illustrates non-chronological backtracking for a given CNF formula. Once both x_1 and x_2 are assigned value 0, there are no possible assignments for the remaining variables x_3 and x_4 that can satisfy the formula. In this example, chronological backtracking wastes a potentially significant amount of time exploring a region of the search space without

solutions, only to discover, after potentially much effort, that the region does not contain any satisfying assignments.

The forms of non-chronological backtracking used in state-of-the-art SAT solvers are related to *dependency-directed backtracking* [23], since they are always associated with learning from conflicts. The incorporation of learning consists of the following: for each identified conflict, its causes are identified, and a new clause (called *nogood*) is created to explain and subsequently prevent the identified conflicting conditions.

In the next section we address conflict-directed backjumping (CBJ) [19], another form of non-chronological backtracking that does not incorporate learning. Afterwards, we describe the use of conflict-directed backjumping jointly with learning.

4.1 Conflict-Directed Backjumping

Conflict-directed backjumping (CBJ) [19] is the most accurate form of backjumping, and can be considered a combination of Gaschnig's backjumping (BJ) [9] and Dechter's graph-based backjumping (GBJ) [8].

BJ aims performing higher jumps in the search tree, rather than backtracking to the most recent yet untoggled decision variable. For each value of a variable v_j, Gaschnig's algorithm obtains the lowest level for which the considered assignment is inconsistent. In addition, BJ uses a marking technique that maintains, for each variable v_j, a reference to a variable v_i with the deepest level of the different levels with which any value of v_j was found to be inconsistent. Hence, a backjump from v_j is to v_i. Moreover, if the domain of v_i is wiped-out, then the search must chronologically backtrack to v_{i-1}. q

As an improvement, Dechter's GBJ extracts knowledge about dependencies from the constraint graph. CBJ builds upon this idea and, based on dependencies from the constraints, records the *set* of past variables that failed consistency checks with each variable v. This set (called *conflict set* in [8]) allows the algorithm to perform multiple jumps.

4.2 Learning and Conflict-Directed Backjumping

Learning can be combined with CBJ when each identified conflict is analyzed, its causes are identified, and a nogood is recorded to explain and prevent the identified conflicting conditions from occurring again during the subsequent search. Moreover, the newly recorded nogood is then used to compute the backtrack point as the *most recent* decision assignment from all the decision assignments represented in the recorded nogood.

For implementing learning techniques common to some of the most competitive backtrack search SAT algorithms, it is necessary to properly *explain* the truth assignments to the propositional variables that are implied by the clauses of the CNF formula. For example, let $x = v_x$ be a truth assignment implied by applying the unit clause rule to a unit clause ω. Then the explanation for this

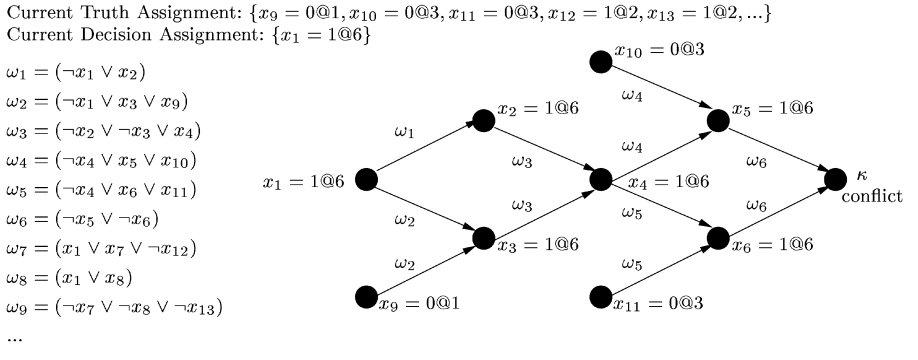


Fig. 2. Example of conflict diagnosis with nogood recording

assignment is the set of assignments associated with the remaining literals of ω, which are assigned value 0. For example, let $\omega = (x_1 \vee \neg x_2 \vee x_3)$ be a clause of a CNF formula φ, and assume the truth assignments $\{x_1 = 0, x_3 = 0\}$. For clause ω to be satisfied we must necessarily have $x_2 = 0$. Hence, we say that the *antecedent assignment* of x_2, denoted as $A(x_2)$, is defined as $A(x_2) = \{x_1 = 0, x_3 = 0\}$.

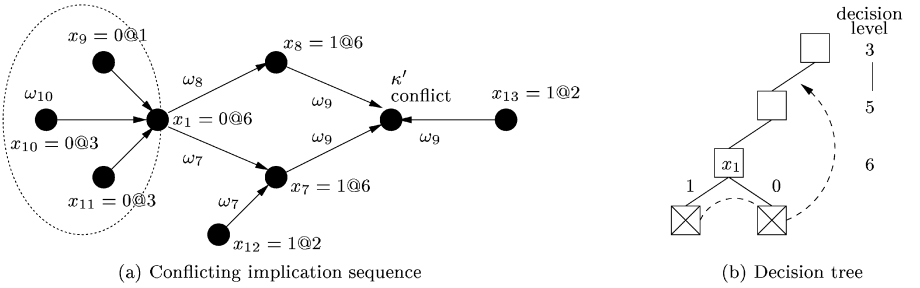
In addition, in order to explain other NCB-related concepts, we shall often analyze the directed acyclic *implication graph* created by the sequences of implied assignments generated by BCP. An implication graph I is defined as follows:

1. Each vertex in I corresponds to a variable assignment at a given decision level $x = \nu(x)@d$.
2. The predecessors of vertex $x = \nu(x)$ in I are the antecedent assignments $A(x)$ corresponding to the unit clause ω that caused the value of x to be implied. The directed edges from the vertices in $A(x)$ to vertex $x = \nu(x)$ are all labeled with ω. Vertices that have no predecessors correspond to decision assignments.
3. Special conflict vertices are added to I to indicate the occurrence of conflicts. The predecessors of a conflict vertex κ correspond to variable assignments that force a clause ω to become unsatisfied and are viewed as the antecedent assignment $A(\kappa)$. The directed edges from the vertices in $A(\kappa)$ to κ are all labeled with ω.

Next, we illustrate nogood recording with the example of Figure 2. A subset of the CNF formula is shown, and we assume that the current decision level is 6, corresponding to the decision assignment $x_1 = 1$. This assignment yields a conflict κ involving clause ω_6. By inspection of the implication graph, we can readily conclude that a *sufficient condition* for this conflict to be identified is,

$$(x_{10} = 0) \wedge (x_{11} = 0) \wedge (x_9 = 0) \wedge (x_1 = 1) \tag{1}$$

By creating clause $\omega_{10} = (x_{10} \vee x_{11} \vee x_9 \vee \neg x_1)$ we prevent the same set of assignments from occurring again during the subsequent search.



(a) Conflicting implication sequence (b) Decision tree

Fig. 3. Computing the backtrack decision level

In order to illustrate non-chronological backtracking based on nogood recording, let us now consider the example of Figure 3, which continues the example in Figure 2, after recording clause $\omega_{10} = (x_{10} \vee x_{11} \vee x_9 \vee \neg x_1)$. At this stage BCP implies the assignment $x_1 = 0$ because clause ω_{10} becomes unit at decision level 6. By inspection of the CNF formula (see Figure 2), we can conclude that clauses ω_7 and ω_8 imply the assignments shown, and so we obtain a conflict κ' involving clause ω_9. By creating clause $\omega_{11} = (\neg x_{13} \vee \neg x_{12} \vee x_{11} \vee x_{10} \vee x_9)$ we prevent the same conflicting conditions from occurring again. It is straightforward to conclude that even though the current decision level is 6, all assignments directly involved in the conflict are associated with variables assigned at decision levels less than 6, the highest of which being 3. Hence we can backtrack immediately to decision level 3.

4.3 Nogood Deletion Policy

Unrestricted nogood recording can in some cases be impractical. Recorded nogoods consume memory and repeated recording of nogoods can eventually lead to the exhaustion of the available memory. Observe that the number of recorded nogoods grows with the number of conflicts; in the worst case, such growth can be *exponential* in the number of variables. Furthermore, large recorded nogoods are known for not being particularly useful for search pruning purposes [16]. Adding larger nogoods leads to additional overhead for conducting the search process and, hence, it eventually costs more than what it saves in terms of backtracks.

As a result, there are three main solutions for guaranteeing the worst case growth of the recorded nogoods to be *polynomial* in the number of variables [14]:

1. We may consider *n-order learning*, that records only nogoods with n or fewer literals [8].
2. Nogoods can be temporarily recorded while they either imply variable assignments or are unit clauses, being discarded as soon as the number of unassigned literals is greater than an integer m. This technique is named *m-size relevance-based learning* [3].

3. Nogoods with a size less than a threshold k are kept during the subsequent search, whereas larger nogoods are discarded as soon as the number of unassigned literals is greater than one. We refer to this technique as *k-bounded learning* [16].

Observe that we can combine k-bounded learning with m-size relevance-based learning. The search algorithm is organized so that all recorded nogoods of size no greater than k are kept and larger nogoods are deleted only after m literals have become unassigned.

More recently, a heuristic nogood deletion policy has been introduced [11]. Basically, the decision whether a nogood should be deleted is based not only on the number of literals but also on its *activity* in contributing to conflict making and on the number of decisions taken since its creation.

5 Experimental Results

In this section we present the obtained experimental results. We start by describing the experimental setup that has been used for the different results. Then we analyze the results for the DPLL-CB SAT algorithm, the DPLL-CBJ SAT algorithm and the DPLL-CBJ SAT algorithm with nogood recording.

5.1 Experimental Setup

In order to experimentally evaluate the different algorithms, in a controlled experiment that ensures that only specific differences are evaluated, a dedicated SAT solving framework is needed. Consequently, we developed the JQUEST SAT framework, a Java implementation that can be used to conduct unbiased experimental evaluations of SAT algorithms and techniques.

This comparison was performed using the JQUEST SAT solver on instances selected from several classes of instances (see Table 1)[3]. In all cases, the problem instances chosen can be solved with several thousands of decisions by the most effective solvers, usually taking a few tens of seconds, thus being significantly hard. For this reason, different algorithms can result in significant variations on the time required for solving a given instance. In addition, we should also observe that the problem instances selected are intended to be *representative*, since each resembles, in terms of hardness for SAT solvers, the typical instance in each class of problem instances.

For the results shown a P-IV@1.7 GHz Linux machine with 1 GByte of physical memory was used. The Java Virtual Machine used was SUN's HotSpot JVM for JDK1.4. The CPU time was limited to 1500 seconds.

[3] All the instances are available from `http://www.lri.fr/~simon/satex/satex.php3` (Sat-Ex web site), with the exception of the *superscalar processor verification* instances.

Table 1. Example Instances

Application Domain	Selected Instance	# Variables	#Clauses	Satisfiable?
Circuit Testing (Dimacs)	bf0432-079	1044	3685	N
	ssa2670-141	4843	2315	N
Inductive Inference(Dimacs)	ii16b2	1076	16121	Y
	ii16e1	1245	14766	Y
Parity Learning(Dimacs)	par16-1-c	317	1264	Y
	par16-4	1015	3324	Y
Graph Colouring	flat200-39	600	2237	Y
	sw100-49	500	3100	Y
Quasigroup	qg3-08	512	10469	Y
	qg5-09	729	28540	N
Blocks World	2bitadd_12	708	1702	Y
	4blocksb	410	24758	Y
Planning-Sat	logistics.a	828	6718	Y
	bw_large.c	3016	50457	Y
Planning-Unsat	logistics.c	1027	9507	N
	bw_large.b	920	11491	N
Bounded Model Checking	barrel5	1407	5383	N
	queueinvar16	1168	6496	N
	longmult6	2848	8853	N
Superscalar Processor Verification	dlx2_aa	490	2804	N
	dlx2_cc_a_bug17	4847	39184	Y
	2dlx_cc_mc_ex_bp_f2_bug006	4824	48215	Y
	2dlx_cc_mc_ex_bp_f2_bug010	5754	60689	Y
Data Encryption Standard	cnf-r3-b2-k1.1	5679	17857	Y
	cnf-r3-b4-k1.2	2855	35963	Y

5.2 CBJ and Nogood Recording

The first table of results (Table 2) shows the CPU time required to solve each problem instance[4]. For the algorithms considered: **CB** denotes the chronological backtracking search SAT algorithm (corresponding to DPLL), **CBJ** denotes the DPLL-CBJ SAT algorithm and **CBJ+ng** denotes the CBJ SAT algorithm with nogood recording. Moreover, a variety of nogood deletion policies were considered, depending on the value of k, where k defines the *k-bounded learning* procedure used (see Section 4.3). For instance, `+ng10` means that recorded nogoods with size greater than `10` are deleted as soon as they become unresolved (i.e. not satisfied with more than one unassigned literal), whereas `+ngAll` means that all the recorded nogoods are kept.

Table 2 reveals interesting trends, and several conclusions can be drawn:

– Clearly, CB and CBJ have in general similar behavior (except for *bf0432-079* and *data encryption standard* instances, that only CBJ is able to solve).

[4] Instances that were not solved in the allowed CPU time are marked with —.

Table 2. CPU Time (in seconds)

Instance	CB	CBJ	CBJ+ng						
			+ng0	+ng5	+ng10	+ng20	+ng50	+ng100	+ngAll
bf0432-079	—	41.74	5.18	2.78	2.97	2.70	1.53	1.44	1.45
ssa2670-141	—	—	1.21	0.87	0.81	0.52	0.55	0.54	0.56
ii16b2	—	—	—	—	857.61	302.63	158.63	141.41	141.05
ii16e1	—	—	20.58	26.75	20.40	12.65	12.89	15.96	11.86
par16-1-c	65.92	77.06	19.91	14.75	16.07	16.78	18.19	18.08	18.13
par16-4	14.88	20.59	11.51	8.49	8.77	9.34	7.16	7.20	7.14
flat200-39	8.44	8.75	97.35	255.04	85.19	114.05	67.55	67.00	67.19
sw100-49	—	—	1.94	13.48	1.26	2.18	0.73	0.74	0.71
qg3-08	2.29	2.65	0.86	0.88	0.91	1.00	1.07	1.30	1.32
qg5-09	13.28	8.61	1.35	1.29	1.06	1.17	1.16	1.21	1.15
2bitadd_12	—	—	—	—	—	—	87.68	50.74	50.93
4blocksb	—	—	31.23	30.25	39.62	29.66	16.34	20.33	31.75
logistics.a	—	—	2.98	1.87	1.62	1.65	1.61	1.63	1.68
bw_large.c	—	—	76.03	55.13	36.41	38.37	43.71	38.03	38.06
logistics.c	—	—	26.88	7.24	4.60	15.40	10.22	10.23	10.25
bw_large.b	8.27	4.78	1.60	0.59	0.61	0.64	0.61	0.62	0.62
barrel5	99.49	132.37	171.94	279.31	36.35	19.80	23.43	24.00	21.99
queueinvar16	—	—	23.36	21.39	15.74	15.48	8.05	8.08	8.12
longmult6	—	—	27.66	23.63	26.20	29.16	32.32	31.74	32.02
dlx2_aa	—	—	54.74	36.21	37.96	9.80	6.43	6.62	6.60
dlx2_cc_a_bug17	—	—	430.31	—	—	500.25	220.06	6.48	6.54
2dlx_..._bug006	—	—	17.29	14.32	3.87	2.27	2.27	2.23	2.22
2dlx_..._bug010	—	—	3.32	4.13	3.83	5.31	4.55	2.03	1.93
cnf-r3-b2-k1.1	802.90	19.37	3.05	3.13	2.39	2.58	2.40	2.16	3.90
cnf-r3-b4-k1.2	405.98	12.62	5.42	5.39	5.48	5.12	4.89	4.45	3.91

- The CBJ+ng algorithms are in general clearly more efficient than the other algorithms. Indeed, for almost all the instances CBJ+ng achieves remarkable improvements, when compared with CB or with CBJ. Instances *flat200-39* and *barrel5* are the only exceptions. (For instance *barrel5*, this is only true for CBJ+ng with small values of k.)
- Some of the instances that are not solved by CBJ in the allowed CPU time (e.g *ii16b2* and *dlx2_cc_a_bug17*), also need a significant amount of time to be solved by *k-bounded learning* with a small value of k.
- For instance *flat200-39*, recorded nogoods result in an additional search effort to find a solution.
- From a practical perspective, unrestricted nogood recording is *not* necessarily a bad approach.

Table 3. Searched nodes

Instance	CB	CBJ	CBJ+ng						
			+ng0	+ng5	+ng10	+ng20	+ng50	+ng100	+ngAll
bf0432-079	—	98824	3939	1950	2010	1600	1168	1188	1188
ssa2670-141	—	—	2882	1988	1480	806	736	698	698
ii16b2	—	—	—	—	101451	32923	12188	10573	10573
ii16e1	—	—	20872	24714	17271	13869	8117	8442	7869
par16-1-c	52800	52800	11307	7249	7524	5255	5364	5364	5364
par16-4	10673	10246	4157	2676	2745	2467	1919	1919	1919
flat200-39	6286	5427	139264	287983	51308	40888	26428	25738	25738
sw100-49	—	—	4596	44247	2370	3748	1450	1450	1450
qg3-08	703	703	220	220	242	214	222	282	282
qg5-09	1578	1547	373	329	318	337	337	337	337
2bitadd_12	—	—	—	—	—	—	21244	11238	11238
4blocksb	—	—	5559	5007	6205	5009	2618	2491	3363
logistics.a	—	—	32872	16999	14899	15185	15185	15185	15185
bw_large.c	—	—	6137	5132	2763	2878	3000	2783	2783
logistics.c	—	—	55721	18839	14520	16444	15441	15441	15441
bw_large.b	1431	1112	293	128	195	195	195	195	195
barrel5	24727	24664	90115	141953	14684	8731	10396	12315	5985
queueinvar16	—	—	45053	41510	24842	19557	8460	8506	8083
longmult6	—	—	7407	5482	5666	5507	5019	4729	4725
dlx2_aa	—	—	319120	204995	208725	21036	10062	10035	10035
dlx2_cc_a_bug17	—	—	446626	—	—	212816	85713	3383	3383
2dlx_..._bug006	—	—	32297	25259	7775	3288	3227	3123	3123
2dlx_..._bug010	—	—	10086	19002	14358	13229	8533	3547	3522
cnf-r3-b2-k1.1	219037	6273	1168	1138	872	942	825	667	1221
cnf-r3-b4-k1.2	57843	2216	1011	1012	1010	943	910	776	729

It is interesting to observe that the *uselessness* of CBJ-related algorithms w.r.t. CB-related algorithms has been experienced in the past [4,18]. CBJ, when applied jointly with a domain filtering procedure (e.g. AC) and an accurate variable ordering heuristic, has been considered an expensive approach that almost always slows down the search, even when it saves a few constraint checks[5].

Table 3 gives the results for the number of searched nodes, for each instance and for the different configurations. It is plain from the results that CB and CBJ in general need to search more nodes to find a solution than the other algorithms. This can be explained by the effect of the recorded nogoods. Besides explaining an identified conflict, nogoods are often re-used, either for yielding conflicts or for implying variable assignments, introducing significant pruning in

[5] However, different conclusions have been obtained for specific classes of instances [5].

the search tree. Moreover, other conclusions can be established from the results on the searched nodes:

- For the *par* instances, CB and CBJ have the same or an approximate number of search nodes for these instances. This is explained by the fact that there are none or just a few backjumps during the search.
- For instances *logistics.a*, *bw_large.b* and *qg5-09* the search needs the same number of nodes for increasing values of k, since only small-size nogoods are recorded.
- Usually more recorded nogoods imply less searched nodes and less time needed to find a solution. (Even though the reduction in the number of nodes is more significant than the reduction in the amount of time, due to the overhead introduced by the management of nogoods.)

Overall, the effect of nogood recording is clear, and in general dramatic. The results clearly indicate that nogood recording is an essential component of current state-of-the-art SAT solvers. Nevertheless, the actual role played by the value of k is not clear, and subject of additional research.

As a final remark, we evaluated whether a different variable ordering heuristic could have affected the results[6]. The intuition was that a more elaborate heuristic could have improved the results obtained for CB and CBJ. Nevertheless, the experimental results presented in [13] for CB and CBJ, that include a more sophisticated heuristic, are still far from being competitive with non-chronological backtracking with nogood recording.

6 Conclusions and Future Directions

In this paper we address the use of DPLL-CBJ in SAT algorithms. In addition, we evaluate the effect of nogood recording in DPLL-CBJ SAT algorithms, and further analyze the effect of different nogood deletion policies. Given the experimental results, obtained for representative instances from several classes of problem instances, we conclude that nogood recording is crucial for competitive SAT algorithms. In addition, the results strongly suggest that backjumping techniques are not enough *per se* for state-of-the-art SAT solvers.

Moreover, we believe that CSP algorithms may also improve their performance by applying both jumping and learning. Interestingly, backjumping techniques and learning have their roots in Truth Maintenance Systems [23] but have been extensively studied in CSP [8,9,19]; nevertheless, constraint programming technology appears not to exploit it.

Future research work will extend the results of this paper by considering alternative approaches with the goal of optimizing SAT solvers. It is well-known that

[6] For the above results, we have applied the variable selection heuristic VSIDS (Variable State Independent Decaying Sum) [17]. It selects the literal that appears most frequently over all clauses, which means that the metrics only have to be updated when a new recorded clause is created.

state-of-the-art SAT solvers use nogood recording. On the other hand, DPLL-CBJ does not incorporate learning, but does consider conflict sets. Hence, we can envision an algorithm that explores the advantages of CBJ to compensate the disadvantages of nogood recording. This algorithm can apply nogood recording, but use conflict sets to avoid recording large nogoods that must be eventually deleted.

Acknowledgments. We would like to thank Patrick Prosser for the insightful discussions we had on CBJ. This work is partially supported by the European research project IST-2001-34607 and by Fundação para a Ciência e Tecnologia under research projects PRAXIS/C/EEI/11249/98 and POSI/34504/CHS/2000.

References

1. K. Apt. Some remarks on boolean constraint propagation. In *New Trends in Constraints*, number 1865 in Lecture Notes in Artificial Intelligence, pages 91–107. Springer, 2000.
2. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 46–60, August 1996.
3. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, July 1997.
4. C. Bessière and J. C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 61–75, August 1996.
5. X. Chen and P. van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, July 1960.
8. R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.
9. J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1979.
10. I. Gent. Arc consistency in SAT. In *Proceedings of the European Conference on Artificial Intelligence*, pages 121–125, July 2002.
11. E. Goldberg and Y. Novikov. BerkMin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.
12. C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 341–355, October 1997.
13. I. Lynce and J. P. Marques-Silva. The effect of nogood recording in MAC-CBJ SAT algorithms. Technical Report RT/04/2002, INESC, April 2002.
14. I. Lynce and J. P. Marques-Silva. An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence*, 37(3):307–326, March 2003.

15. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
16. J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, November 1996.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.
18. P. Prosser. Domain filtering can degrade intelligent backjumping search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 262–267, August 1993.
19. P. Prosser. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, August 1993.
20. P. Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report 177, University of Strathclyde, Glasgow, Scotland, May 1995.
21. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the European Conference on Artificial Intelligence*, pages 125–129, August 1994.
22. B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 290–295, August 1993.
23. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, October 1977.
24. T. Walsh. SAT *v* CSP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 441–456, September 2000.
25. R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–160, July 1988.
26. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.

POOC – A Platform for Object-Oriented Constraint Programming

Hans Schlenker and Georg Ringwelski

Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany
{hans.schlenker,georg.ringwelski}@first.fraunhofer.de

Abstract. In this paper, we describe an implementation-independent object-oriented interface for commercial and academic Constraint Solvers. This serves as a basis for evaluating different Constraint Solvers and for developing solver-independent applications. We show, how applications can use the interface, which solvers are already integrated into the framework and how additional solvers can be added. Furtermore, we provide to the community the described system as real Java packages via Internet, that even includes a basic but powerful Constraint Solver.

1 Motivation

Constraint Programming (CP, [9,8,5]) has emerged an adequate means to implement and solve many problems known from Artificial Intelligence. Constraint Satisfaction Problems (CSP) can be solved by stating the problem itself instead of defining an algorithm to find a solution. The constraints are posted in a domain-specific solver that infers a solution of the CSP from the constraints and the possible values of constraint variables. Thanks to this declarativity, constraint-based methods are used in many academic applications and in some commercial software systems. Another reason for the success of CP is its efficiency. Many combinatorial NP-hard problems can be solved with constraint-based methods.

From the application point of view, CP always needs a Constraint Solver. Thus, one is usually faced with the problem to find the solver that best fits some specific needs. In a recent project for example, we compared the available solvers wrt. performance and functional capabilities. We also wanted to be somehow independent from a particular implementation or product while making these first steps. This could allow us to develop a system, using some constraint solver, while being able to exchange the solver by another one in case we need new features or the former product becomes unavailable for some reason. This was our motivation to create POOC: a Platform for Object-Oriented Constraint Programming. This new library can be used for the implementation of constraint-based software and allows:

- runtime comparison of existing solvers,
- to keep benchmark programs and maybe even applications solver-independent

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 159–170, 2003.

POOC is an Application Programming Interface consisting of Java classes that define a finite domain constraint programming syntax and import semantics from solvers that are plugged in during runtime. We think, that Java programmers with some basic knowledge on constraint programming will be able to use the package after looking at its standardized documentation. We intended to make constraint-based methods available in Java programs in the established manner, i.e. by importing a library. But please note, that POOC itself does not provide any functionality or semantics, it only defines a syntax that can be considered an extension to the Java programming langage for constraint-based modelling and constraint satisfaction. Finite domain constraint solvers (and thus semantics) are plugged into the POOC-based applications via their implementation of the POOC interface. This allows the development of constraint-based programs without choosing any particular solver in advance.

2 Approach

Our approach is to create a solver-independent object-oriented platform for finite domain constraint programming. This platform is designed to be a wrapper around existing constraint solvers. We use Java as implementation language: POOC is a Java package that provides wrappers, it is not a solver by itself. In Figure 1 we show the different layers and interfaces that are used in a POOC-based Java program and how the evaluation of the method-calls is delegated to the constraint engines.

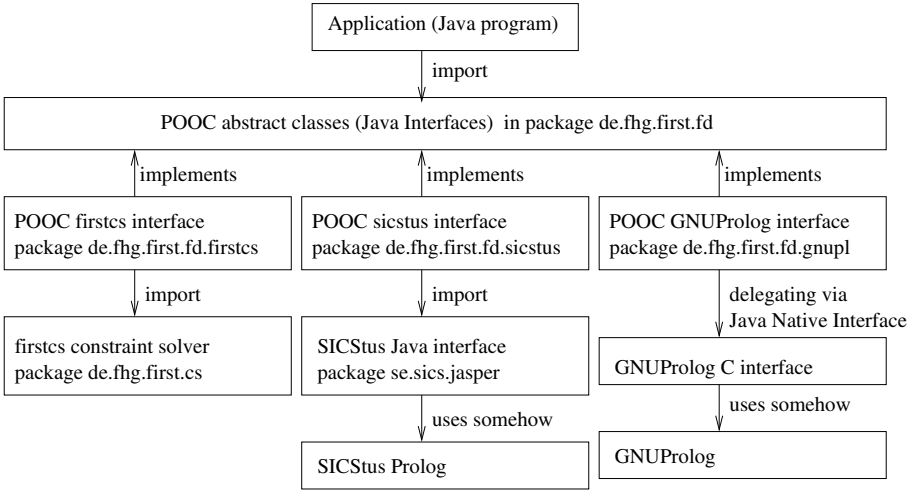


Fig. 1. The layers of a POOC-based Java application program with different constraint solvers. The interface to GNUProlog is not implemented yet.

The wrapper abstracts from implementation details: similar constraints in different solvers (like the `disjoint` constraint in SICStus Prolog [10] and `diffn` in CHIP [2]) are called through one interface and similar structures are mapped into one class hierarchy. For example, arithmetic expressions are built recursively from arithmetic operations, variables and constants. The wrapper is designed as Object Factory[1]: new objects like variables or expressions are created by the solver at runtime, instead of explicitely being linked against some solver at compile time. We achieved this by defining Interfaces that every concrete solver (or its wrapper, resp.) must implement. In Java, an *Interface* is a special kind of an abstract class and therefore especially well suited for defining abstract (here: implementation-independent) entities. In Figure 2 we show the inheritance graph of the classes of the POOC Interface package `de.fhg.first.fd` that defines all method names and the packages that implement these methods by delegating them to real solvers. We use the popular UML [1] notation of a class diagram, where a class is described by a rectangle with a middle line and a package is represented by a tagged rectangle. The classes are connected with up-arrows according to their inheritance relation. A rhombic arrow represents an object aggregation in UML.

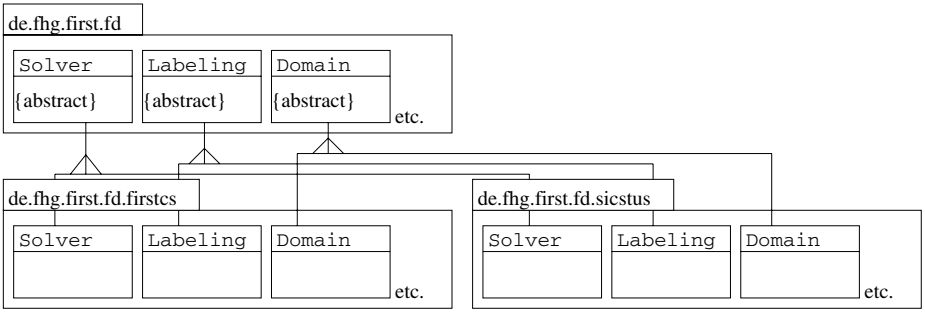


Fig. 2. Inheritance of POOC syntax to the interface implemntations of the used solvers.

Figure 3 shows the POOC-interfaces, their methods and how the interfaces are related to each other.

Each `Solver` provides methods for the creation of concrete variables, domains, expressions (represented by the interface `X`) and labelings. This allows an application to be compiled independently from the concrete solver which is linked at runtime. Additionally, each `Solver` implements the basic FD constraints, like `alldifferent` [11] and arithmetic in-/equalities, and modification of a variable's

[1] A *Factory Method* is a design pattern that is often used in Object-Oriented Software Development, defined in [6]. The canonical application case is intended to be: *"Define an interface for creating an object, but let subclasses decide which class to instantiate."*

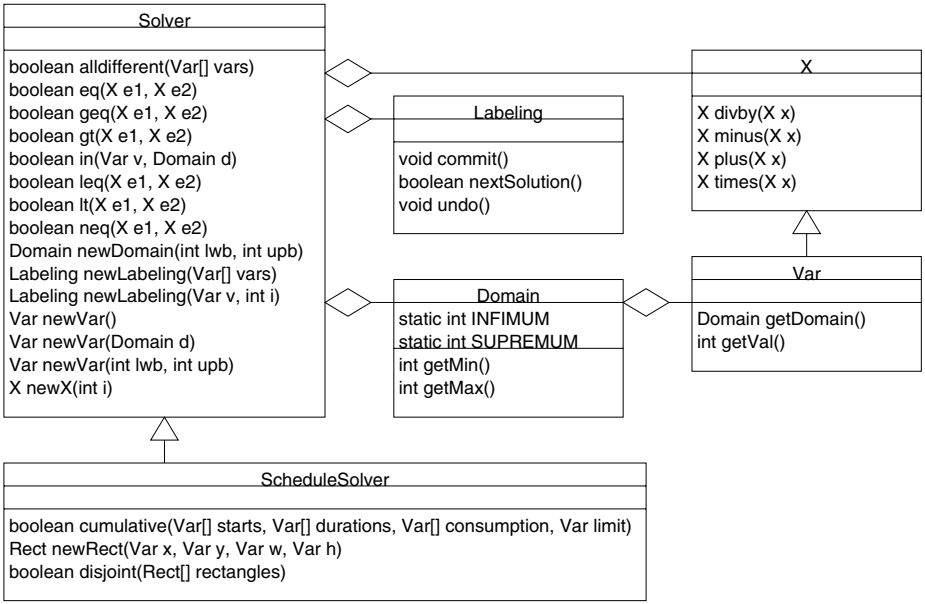


Fig. 3. POOC's main classes

domain (method `in`). A `Domain` object describes such a domain which is given at least by its minimum and maximum. If a `Domain` is defined without specifying min and max, the values `INFIMUM` and `SUPREMUM`, which are provided by POOC are used instead. Of course, in an application that uses any particular solver, also the respective values for min and max of this solver can be used. A `Var` wraps a constraint variable whose current domain and, in case it is instantiated, its value can be queried. Each `X` stands for an arithmetic expression. Expressions can be recursively built by variables and constants and expressions joined by operators. A `Labeling` object is used for assigning values to the allocated variables: new solutions can be generated, an actual assignment can be commited (accepted) or undone. We use a labeling *object* and not — what seems more intuitive — a *method* since we need references to existing labelings to generate successive solutions. Each labeling, domain, expression, and variable is related to some `Solver`. This means, it contains a reference to its solver, which is depicted by the rhombic arrow in the class diagram in Figure 3. Each variable is related to its domain. The interface `Var` is a specialization (depicted by the up-arrow) of the interface `X`. This means that every variable is an expression. A `ScheduleSolver` is a `Solver` that implements additional combinatorial constraints: `cumulative` (e.g. [4]) and `disjoint` (e.g. [10]). We will discuss such object-oriented specializations of the common `Solver` interface, that are to specify certain requirements to be met by the used solver in Section 4. A `ScheduleSolver` serves also as a factory for rectangles.

This design has the following properties:

- It keeps the application program independent from a concrete solver (since the application can use abstract interfaces and object factories instead of concrete classes).
- It allows for the integration of additional constraint solvers (since it is rather abstract and hides implementation details like representation of domains or rectangles).
- It allows the use of more than one solver in one program (since each entity is related to its own solver).

The last property eases runtime comparison of constraint solvers.

3 Application

A typical Constraint Program consists of three major parts: definition of decision variables, statement of constraints, and search for a solution. Using POOC, all three parts are rather easy to implement. Figure 4 shows an implementation of the famous Send-More-Money puzzle.

The first step is to load a Constraint Solver. In Java, this can be done dynamically, i.e. the runtime implementation of the solver to be used can be given at runtime (here by passing the name of an existing class to the main program). The only requirement is, that the loaded solver implements a given interface: the `Solver`. This solver is a factory that serves for creating new variables, etc.

Therefore, new decision variables are created by calling the solver's factory-methods `newVar()`. The application can provide an initial domain (using `newVar(Domain d)`) or lower and upper bounds directly (using `newVar(int lwb, int upb)`).

Constraints are then stated using the solver's methods. Since in our example, all decision variables have to be assigned different values, we use the `alldifferent` constraint. To express the requirement that $send + more = money$, we have to build some arithmetic expressions: all decision variables will get a one-digit number, the puzzle's requirement can thus be stated as follows: `1000*s + 100*e + 10*n + d + 1000*m + 100*o + 10*r + e = 10000*m + + 1000*o + 100*n + 10*e + y`. The according expressions are built recursively using the variables, new number-expressions, and the given arithmetic operators. Since Java does — in contrast to C++ or Prolog — not allow to overload symbolic operators, we have to use real method-names. This makes our *arithmetic* expressions a bit clumsy, unfortunately.

Basic labeling (search for an appropriate variable assignment, i.e. solution) is simple: each solver provides an according method `newLabeling(Var[])`. The returned reference to a `Labeling` object serves as a handle for the generation of new solutions and commitment or disposal of old ones. The use is rather straight forward: `nextSolution()` generates a new variable assignment and returns `true` in case there is one and `false` if not. The basic labeling procedure that each solver has to provide, usually comes from a native implementation and sometimes

```
public class Send {
    public static void main(String[] args) throws Exception {
        // Create solver from class name given as first program argument.
        Solver solver = (Solver) Class.forName(args[0]).newInstance();
        // Create variables with initial domains.
        Var s = solver.newVar(1,9);   Var m = solver.newVar(1,9);
        Var e = solver.newVar(0,9);   Var o = solver.newVar(0,9);
        Var n = solver.newVar(0,9);   Var r = solver.newVar(0,9);
        Var d = solver.newVar(0,9);   Var y = solver.newVar(0,9);
        // State constraints.
        solver.alldifferent(new Var[]{s,e,n,d,m,o,r,y});
        solver.eq(      s.times(solver.newX(1000))
                  .plus(e.times(solver.newX(100)))
                  .plus(n.times(solver.newX(10)))
                  .plus(d)
                  .plus(m.times(solver.newX(1000)))
                  .plus(o.times(solver.newX(100)))
                  .plus(r.times(solver.newX(10)))
                  .plus(e) ,
                        m.times(solver.newX(10000))
                  .plus(o.times(solver.newX(1000)))
                  .plus(n.times(solver.newX(100)))
                  .plus(e.times(solver.newX(10)))
                  .plus(y)
                  );
        // Create labeling (reference). No solution is computed here.
        Labeling l = solver.newLabeling(new Var[]{s,e,n,d,m,o,r,y});
        // Generate successive solutions and write them to stdout.
        while (l.nextSolution()) {
            System.out.println("   "+s+e+n+d);
            System.out.println("+  "+m+o+r+e);
            System.out.println("=======");
            System.out.println("= "+m+o+n+e+y+"\n");
        }
    }
}
```

Fig. 4. Java+POOC implementation of the Send-More-Money puzzle.

allows the specification of options. These options are implementation-specific and therefore not directly reflected by the interface. It does, however, provide technical support for passing optional parameters from external resources.

Labelings can also be implemented directly in Java: using `newLabeling( Variable v,int i)`, a single decision variable can be assigned a concrete value, which can be taken back using `undo()`. As an example, Figure 5 shows an implementation of a simple depth first search: first variable first, lowest value first.

```
static void label(Solver solver, Var[] vars) {
    label(solver,vars,0);
}
static void label(Solver solver, Var[] vars, int index) {
    if (index >= vars.length) { // solution complete
        for (int i=0; i<vars.length; i++) // print result
            System.out.print(vars[i]+" ");
        System.out.println();
        return;
    }
    Var v = vars[index]; // instantiate v
    for (int i=v.getMin(); i<=v.getMax(); i++) {
        Labeling l = solver.newLabeling(v,i);
        if (l.nextSolution())
            label(solver,vars,i+1); // instantiate next var
        l.undo();
    }
}
```

Fig. 5. User defined labeling procedure.

Found solutions can then be processed by the application: the variable's values can directly be queried using `getVal()` which throws an exception, if the variable is not yet instantiated. The standard method `Var.toString()` is overloaded such that variable's values can directly be printed.

As a final example, see Figure 6, an implementation of the Golomb Ruler problem[2]. Our implementation is inspired by Joachim Schimpf's Eclipse [15] implementation, presented in [7]. To compare e.g. the expressiveness of our approach to ILOG [12], one may want to see the ILOG Solver implementation, also given there.

You can see that also this problem is very easy to implement. We even realized a simple Branch-and-Bound method here, which effectively consists of 6 lines of Java code! We used this program to evaluate the runtime overhead produced by POOC and to compare our solver firstcs with the SICStus implementation (Figure 7).

To evaluate the POOC runtime overhead, we had to write different versions of one program: for SICStus Prolog (in Prolog), for SICStus Prolog plus *Jasper*[3] (in Java), the POOC one for both SICStus and firstcs, and a version using firstcs di-

[2] [7]: *These problems are said to have many practical applications including sensor placements for x-ray crystallography and radio astronomy. A Golomb ruler may be defined as a set of m integers* $0 = a_1 < a_2 < ... < a_m$ *such that the* $m(m-1)/2$ *differences* $a_j - a_i, 1 <= i < j <= m$ *are distinct. Such a ruler is said to contain m marks and is of length* a_m*. The objective is to find optimal (minimum length) or near optimal rulers.*

[3] SICStus (at least the new versions) actually provides already a Java interface: *Jasper*. This one is a rather low-level interface with which one can stick together Prolog

```
public class Golomb {
    public static void main(String[] args) throws Exception {
        // Create solver from class name given as first program argument.
        Solver solver = (Solver) Class.forName(args[0]).newInstance();
        // Get problem size n from second program argument.
        int n = Integer.parseInt(args[1]);
        Var[] marks = new Var[n];
        Var[] diffs = new Var[(n*(n-1))/2];
        int idiff = 0;
        // Create variables and state constraints.
        marks[0] = solver.newVar(0,0);
        for (int i=1; i<n; i++) {
            marks[i] = solver.newVar(0,Domain.SUPREMUM);
            // marks[i-1] #< marks[i]
            solver.lt(marks[i-1],marks[i]);
            for (int j=0; j<i; j++) {
                diffs[idiff] = solver.newVar(0,Domain.SUPREMUM);
                // marks[i] #= marks[j] + diffs[idiff]
                solver.eq(marks[i],marks[j].plus(diffs[idiff]));
                idiff++;
            }
        }
        solver.lt(diffs[0],diffs[diffs.length-1]); // removes symmetry
        solver.alldifferent(diffs);
        // Generate solutions.
        Labeling l = solver.newLabeling(marks);
        while (l.nextSolution()) {
            // Write solution to stdout.
            for (int i=0; i<n; i++) System.out.print(marks[i]+" ");
            System.out.println();
            // Branch-and-Bound: Remember greatest value ...
            int limit = marks[n-1].getVal();
            l.undo();
            // ... and restrict rightmost mark to be lower than that.
            solver.lt(marks[n-1],solver.newX(limit));
            // New labeling *after* statement of upper bound constraint.
            l = solver.newLabeling(marks);
        }
    }
}
```

Fig. 6. Golomb Ruler implementation.

rectly. The comparison was made on a PentiumIII-800 PC running Linux 2.2.16. We used SICStus 3.9.1 and Java 1.3.1. Figure 7 shows the runtimes for the 8,

terms, call them, and read resulting variable bindings. POOC is more abstract from Prolog and more concrete towards Constraint Programming.

9, 10 and 11-mark Golomb-Ruler problems and for all mentioned systems. We searched for and proofed the optimal solution. The times are given in milliseconds. The bracketed values give the difference between the consecutive lines in percent. The first firstcs test, called "java -server", was made calling `java` with the `-server` option. This is for computational very intensive programs and does better compilation. We only used it here, since for most of the other cases, this option is bad (see also Figure 8).

Number of marks	8	9	10	11
SICStus Prolog	700	6880	64290	1292860
	(+47.29%)	(+48.92%)	(+52.76%)	(+50.22%)
SICStus Prolog + Jasper	1031	10246	98207	1942172
	(+7.95%)	(+5.95%)	(+5.09%)	(+4.61%)
SICStus Prolog + Jasper + POOC	1113	10856	103205	2031647
firstcs, java -server	1848	13148	141422	3116121
	(-4.71%)	(+40.77%)	(+49.03%)	(+50.67%)
firstcs	1761	18509	210759	4695173
	(+4.60%)	(+9.43%)	(+1.18%)	(+2.01%)
firstcs + POOC	1842	20255	213255	4789752

Fig. 7. Runtimes [msec] for the Golomb Ruler benchmark.

The presented usage and examples show some of the advantages, our object-oriented design has:

- Rather abstract entities represent expressions, variables and more complex structures like rectangles for example.
- These entities can be related to each other using high-level constraints such as `alldifferent`, arithmetic constraints or rectangle-disjointness.
- The given classes can be aggregated to even more complex entities such as Tasks, Routes etc., or constraints like Multi-Resource or Round-Trip for the Traveling Salesman Problem.
- Even some complex search algorithms like Branch-and-Bound can be defined easily.
- The runtime overhead is very low.

The current implementation of POOC includes an implementation for SICStus Prolog. In addition to the SICStus Prolog interface, POOC provides an interface to our own constraint solver firstcs.

3.1 firstcs

firstcs[4] is our new plain-Java constraint solver. It is designed as a light-weight special purpose solver. We intend to use it as a basis for new propagation algorithms and for application development. The philosophy behind it is *right-sizing*,

[4] This acronym is derived from its package name `de.fhg.first.cs`.

i.e. building the perfect solver for our problems, while eventually losing generality. Currently, basic and some higher-level FD-constraints are implemented, using state-of-the-art propagation techniques.

For a first impression on how it performs (or *can perform*, if you want): Figure 8 shows runtimes for the *All-Interval Series* problem[5] for different problem sizes. In each test, we searched for *all solutions* to the given problem. The running times are given in milliseconds. We compared the following systems: CHIP (5.4), using a Prolog program, SICStus and firstcs, using POOC (as specified above). The benchmark programs are not given here, but contained in the downloadable package, see below.

When running the benchmarks in Figure 8, we made use of POOC's feature to link solvers against the application during runtime. We could run 75% of the benchmarks with one Java program.

Problem size	6	8	10	12	14
CHIP Prolog	10	280	6910	225870	8868160
SICStus Prolog + Jasper + POOC	51	321	6564	189258	6908643
firstcs + POOC	154	461	8373	265773	11107569
firstcs + POOC, java -server	210	2145	9627	205524	8415093

Fig. 8. Runtimes [msec] for the All-Interval Series benchmark.

4 Extension

POOC's design allows the integration of an arbitrary number of finite domain constraint solver implementations. The effort of adding a new one depends on whether a Java-interface already exists and on the solver's execution model. First steps have been made plugging-in ILOG [12] and GNU Prolog [3]. Since it is a CLP-solver similar to the already plugged-in SICStus, the integration of the latter turns out to be not very problematic. We use Java's native interface JNI [14] to invoke GNU Prolog predicates via its C-interface. The integration of ILOG Solver v5.0, however, is very different in many respects. POOC generaly allows for applications mixing the statement of constraints and the search for a solution. ILOG needs these things be seperated, i.e. first the model being

[5] [7]: *[...] The problem of finding such a series can be easily formulated as an instance of a more general arithmetic problem on Z_n, the set of integer residues modulo n. Given $n \in \mathbb{N}$, find a vector $s = (s_1, ..., s_n)$, such that (i) s is a permutation of $Z_n = \{0, 1, ..., n-1\}$ and (ii) the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, ...|s_n - s_{n-1}|)$ is a permutation of $Z_n - \{0\} = \{1, 2, ..., n-1\}$. A vector v satisfying these conditions is called an* all-interval series of size n*; the problem of finding such a series is the* all-interval series problem of size n*. We may also be interested in finding all possible series of a given size.*

built and finally the solution being searched. Additionally, the definition of a Labeling is very special in ILOG. This is a bit a challenge for its integration into our framework, though not being impossible.

Solvers that provide special functionality to be used in POOC applications can be integrated with extensions of the `Solver`-interface. For example solvers that provide the `disjoint` resp. `diffn` constraint are subsumed in the `Scheduling-Solver` interface as shown in Figure 3. Very specific features (i.e. related to some particular solver) can be integrated by providing a `Solver` class, that implements at least the necessary methods, and additionally specific ones. So, a new `oz.Solver`[13] would have to implement the necessary methods `newVar()`, `eq()`, `newLabel()`, and so forth. Additionally it could provide a search method like `oz.Solver.newBestFirstLabeling()`.

Everybody is invited to use the interface for her own solver. As we stated above, we propose POOC as a general syntax for formulating CSPs and writing problem solvers in the Java programming language. The integration is easy! And with integrating your solver into POOC, you have all above mentioned CSPlib benchmarks available immediately!

5 Download!

We realized the described interface as a Java package: POOC, which we are able to make open to the public! Please visit the dedicated web page at `http://www.first.fhg.de/pooc` .

The package offered there for free download contains the POOC interface definitions, documentation, application examples, and implementations for SICStus Prolog and our solver firstcs. The latter solver is even contained in the package as well! Therefore you can get there a complete easy-to-use Java-Constraint-Solver. You can evaluate the performance of this solver with the multiple benchmark programs from CSPlib [7]. If you have SICStus Prolog with Jasper installed on your computer you can also run the benchmark programs with SICStus and compare the performance.

6 Conclusion

POOC gives a nice framework for the development of object-oriented Java-based constraint programs in the finite domain system and the evaluation of existing Constraint-Solver-implementations. It

- is easy to use.
- allows for specifying underlying solvers at runtime.
- allows for multiple solvers even from different vendors within one program.
- is extensible with new solvers and interfaces.

The system is realized and provided to the public community. Please visit `http://www.first.fhg.de/pooc` .

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language Semantics and Notation Guide 1.3*. Rational Software Cooperation, San Jose, CA, 2001.

[2] Cosytec. Chip 5, 2001. `http://www.cosytec.com`.

[3] Daniel Diaz. GNU Prolog, 2001. `http://gnu-prolog.inria.fr`.

[4] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS88)*, 1988.

[5] Thom Frühwirth and Slim Abdenadher. *Constraint-Programmierung*. Springer, Berlin Heidelberg, 1997. In German.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[7] Ian Gent and Toby Walsh. CSPLib, a problem library for constraints, `http://www.csplib.org`.

[8] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[9] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL-87*, pages 111–119, 1987.

[10] Swedish Institute of Computer Science. SICStus Prolog v3, 2001. `http://www.sics.se/sicstus.html`.

[11] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[12] ILOG S.A. ILOG internet pages, 2001. `http://www.ilog.fr`.

[13] Chritian Schulte. *Programming Constraint Services*. LNAI 2302. Springer, 2002. PhD thesis.

[14] SUN Microsystems, `http://java.sun.com/j2se/1.3/docs/guide/jni/`. *Java Native Interface (JNI)*.

[15] M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, UK, 1997.

A Coordination-Based Framework for Distributed Constraint Solving

Peter Zoeteweij

Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, NL-1090 GB AMSTERDAM, The Netherlands
P.Zoeteweij@cwi.nl

Abstract. This paper gives an overview of DICE (DIstributed Constraint Environment), a framework for the construction of distributed constraint solvers from software components in four categories: (1) variable domain types, (2) (incomplete) solvers, (3) splitting strategies, to build search trees, and (4) search strategies, to traverse these search trees. DICE is implemented using the Manifold coordination language, and coordinates the components of a distributed solver. In addition to the coordination protocols and the algorithms that they implement, the paper describes the construction of solvers both from a constraint programming and a software engineering point of view.

1 Introduction

The applicability of the Idealized Worker Idealized Manager (IWIM) coordination model in the area of distributed constraint solving has been studied in [3, 10]. This work has materialized in DICE (DIstributed Constraint Environment), a software framework, implemented using the Manifold coordination language, where distributed constraint solvers are constructed from solver components in four categories:

1. domain types for the variables of a constraint satisfaction problem (CSP),
2. constraint solvers that can act as a domain reduction operator inside a constraint propagation algorithm,
3. strategies to split the domains of variables in order to build a search tree, and
4. search strategies, which determine how to explore this search tree.

Each component instance can run in a separate process, and the framework coordinates the activities of the components by means of coordination protocols that implement

- a distributed constraint propagation algorithm,
- a distributed termination detection algorithm, and
- facilities that allow a splitting strategy component and a search strategy component to coordinate the network of processes to perform search.

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 171–184, 2003.

The constraint propagation algorithm applies (incomplete) solvers until none of these can reduce the problem any further. In general, this will not lead to a solution to the CSP, and propagation must be interleaved with splitting the domain of a variable in order to systematically search for solutions. Termination detection is needed because usually, we don't want to split the domain of a variable until constraint propagation has finished.

The aim of this paper is to give an overview of DICE. In Sect. 2 we describe the model of constraint solving that it is based on, and provide pointers to more information about coordination programming, the IWIM model, and Manifold. Also here we recall the algorithms and facilities implemented by the coordination protocols. In Sect. 3 and 4 we describe solver configuration from constraint programming and software engineering perspectives, respectively, and Sect. 5 is an overview of a set of components that is available for the construction of solvers. In Sect. 6 we discuss our plans for future work on DICE.

2 Preliminaries

2.1 Model of Constraint Solving

A CSP consists of a set of variables and their associated domains (sets of possible values), and a set of constraints. A constraint is defined on a subset of the variables, and restricts the combinations of values that these variables may assume. The model of constraint solving supported by DICE is further characterized by:

- branch-and-prune tree search,
- branching by splitting variable domains, pruning by constraint propagation,
- constraint propagation by application of domain reduction functions (DRF's) in a chaotic iteration algorithm. DRF's enforce the constraints.

At every node of the search tree, starting with the root node, which corresponds to the original CSP, constraint propagation is applied before branching. When propagation finishes in a node of the search tree, one of three situations occurs: if the domains of all variables have been reduced to singletons, the (leaf) node represents a solution. If one or more variable domains have become empty, the node represents a failure. In all other cases, the node is an internal node of the search tree, and the *splitting strategy* determines how the search tree is expanded.

Splitting involves selecting a variable, and determining the domains for that variable in the resulting sub-CSP's. These two aspects are referred to as *variable selection* and *value selection*, respectively. Example variable selection methods are to select a variable that has the smallest number of possible assignments (fail-first), or to select a variable that occurs in the largest number of constraints. Example value selection methods are enumeration, which creates a subdomain for every value in the original domain, and bisection, which splits the domain in two halves.

The collection of rules for selecting the nodes of the search tree where constraint propagation and consecutive branching are applied in the course of the

solving process is referred to as the *search strategy*. Examples are depth-first chronological backtracking and limited discrepancy search.

It is important to note that the model of constraint solving supported by DICE does not involve constraint checking. Violation of a constraint is concluded when the domain reduction functions that enforce the constraint void the domains of one or more variables.

2.2 Coordination Model and Language

Coordination languages offer language support for composing and controlling software architectures made of parallel or distributed components [11]. In the IWIM model of coordination [4], these components are represented by *processes*. In addition to processes, the basic concepts of IWIM are *ports*, *channels* and *events*. A process is a black box that exchanges *units* of information with the other processes in its environment through its input ports and output ports, by means of standard I/O primitives analogous to read and write. The interconnections between the ports of processes are made through directed channels. Independent of channels, there is an event mechanism for information exchange in IWIM. Events are broadcast by their sources, yielding event *occurrences*. Processes can tune in to specific event sources, and react to event occurrences.

The IWIM view of a software system is a dynamic ensemble of interconnected processes. A process can be regarded as a worker process or a manager process. The responsibility of a worker process is to perform a (computational) task. The responsibility of a manager is to coordinate the communications among a set of worker processes. For this purpose, manager processes can create worker processes and make channel connections to their ports. A manager process may be considered a worker processes by another manager. At the bottom of this hierarchy there is always a layer of *atomic workers*.

Manifold [4,5] is a coordination language for writing program modules (coordinator processes) to manage complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes that comprise a single application. The conceptual model behind Manifold is based on IWIM. A Manifold application consists of a (potentially very large) number of processes running on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages and some of them may not know anything about Manifold, nor the fact that they are cooperating with other processes through Manifold in a concurrent application.

2.3 A Distributed Constraint Propagation Algorithm

The purpose of a constraint propagation algorithm in this context is to apply atomic reduction steps. For the model of constraint solving supported by DICE, these atomic reduction steps are domain reduction functions. Many constraint propagation algorithms maintain a set of atomic reduction steps that still need to be applied, and keep applying reduction steps until this set becomes empty.

After each reduction step, the algorithm considers the changes that have been made, and reduction steps that depend on these changes are added to the set.

In contrast to such inherently sequential algorithms, DICE implements the coordination-based chaotic iteration algorithm of [10]. In this algorithm, each CSP variable is represented by a process that maintains the domain of that variable. Also each domain reduction function is represented by a process that receives input from the processes corresponding to the CSP variables that the function applies to. Channel connections are made between the ports of Variable and DRF processes according to the structure of the CSP. The DRF processes have a buffer associated with each input port, which stores the variable domain last seen on that port. These buffers are initialized by having a Variable process send its domain each time a connection to a DRF process is made.

Figure 1(a) shows an example process network of this algorithm. Variable processes send *reduction requests* to DRF processes. Reduction requests contain the domain of the CSP variable. The DRF process uses this domain to update the buffer associated with the input port that delivers the reduction request. Then it applies the domain reduction function to the domains in the buffers[1]. This yields new domains for the output variables of the domain reduction function. These domains are dispatched through the output ports of the DRF process to the corresponding Variable processes as *update commands*.

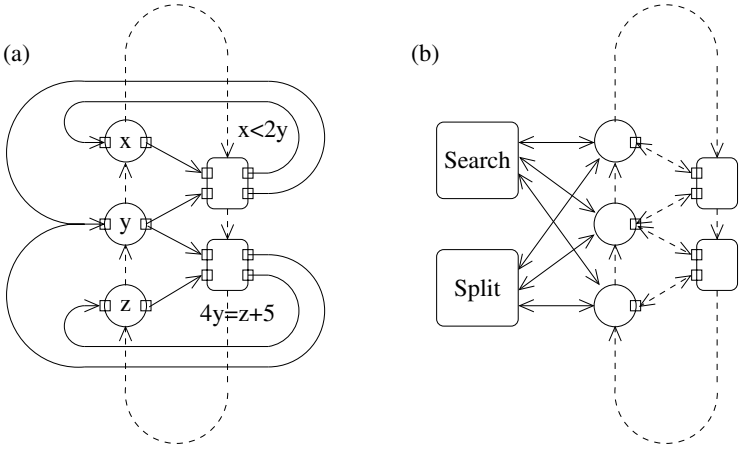


Fig. 1. (a) An example process network of the distributed constraint propagation algorithm, (b) the propagation engine is coordinated from the outside to perform search

Upon receiving an update command, a Variable process computes the intersection of the domain held in the update command and the domain of the CSP

[1] Actually, application of the function is postponed until no more reduction requests are available on the input ports. Such details are omitted from this presentation.

variable held in its internal store. If this intersection is a proper subset of the current domain, the store is updated with the intersection, and the new domain of the CSP variable is dispatched through the output port of the Variable process as a reduction request. The reduction request is broadcast to all DRF processes that connect to this output port. If the intersection does not reduce the domain of the CSP variable, the update command has no effect.

In [10] it is argued that this distributed algorithm implements a restriction of the Generic Iteration Algorithm for Compound Domains of [1]. This allows us to benefit from several properties that have been proven for that algorithm. One of these properties is that the algorithm is guaranteed to terminate if the domains are finite and the DRF's are inflationary. The latter is effectively ensured by having Variable processes compute intersections.

2.4 Termination Detection

Although this is not strictly necessary, usually we do not want to consider expanding the search tree by splitting the domain of a variable before constraint propagation has finished. Therefore, we need to know when the propagation algorithm terminates. With a sequential algorithm, this is easy: it terminates when the set of atomic reduction steps that still need to be applied becomes empty.

In the case of the distributed algorithm of the previous section, the conditions for concluding that constraint propagation has finished are more difficult to verify. The algorithm terminates when:

1. all Variable and DRF processes are idle, and
2. there are no pending update commands or reduction requests in the channels.

DICE employs the algorithm described in [6] to detect these conditions. For the purpose of this algorithm, the processes of the constraint propagation algorithm are connected in a ring network. The dashed lines in Fig. 1(a) show the extra channels for termination detection. All processes maintain a local counter of the number of update commands and reduction requests in the network. The ring network is used for circulating a token. This token is forwarded when the process that holds it becomes idle. When it returns to the process that created it, the token has accumulated the local counters of all process. Termination can be concluded only if this sum equals 0. Together with a black/white coloring of the processes and the token the algorithm ensures correctness in case of asynchronous channels. This corresponds to the Manifold communication model.

2.5 Distributed Splitting

DICE employs a scheme similar to that of [3], where the network of processes of the constraint propagation algorithm is performing work in several nodes of the search tree simultaneously. As a result, multiple tokens of the termination detection algorithm may be circulating on the ring network, one for every instance of the constraint propagation algorithm, and all administration inside the

Variable and DRF processes for the purpose of the propagation and termination detection algorithms is per node of the search tree:

Variable::	*DRF*::
v: $World \xrightarrow{m} Domain$	I: ARRAY $[1..n]$ OF $World \xrightarrow{m} Domain$
$color$: $World \xrightarrow{m} \{\text{black}, \text{white}\}$	$color$: $World \xrightarrow{m} \{\text{black}, \text{white}\}$
n_msg: $World \xrightarrow{m} \mathbb{Z}$	n_msg: $World \xrightarrow{m} \mathbb{Z}$

where n is the number of input ports of the DRF process, and *World* is a datatype whose elements serve as identifiers for nodes of the search tree. Maps v and I hold the data for the propagation algorithm, and *color* and *n_msg* represent the state of the termination detection algorithm.

A partial order is defined on the elements of *World*, stating that an ancestor node is compatible to its descendants, and that a descendant is smaller than its parent. On several occasions, we look for information in the smallest compatible world of a world w. For example, the update commands of the propagation algorithm now consist of a world w, and a domain d. If the world w is not yet known to the Variable process, it intersects d with the domain d' of the CSP variable in the smallest compatible world of w. Only if $d' \cap d \subset d'$, the element $w \rightarrow d' \cap d$ is added to the map v of the Variable process.

The facilities offered by this administration per world are used by two new processes *Split* and *Search*, which implement the splitting strategy (involving variable selection and value selection) and search strategy, respectively. These processes have connections to all Variable processes (Fig. 1(b)), and coordinate the network of the propagation algorithm to perform search.

The Split process is triggered when propagation finishes in a certain world, and may query Variable processes for their domains in that world. On the basis of this information, the Split process can then decide which variable to split (if any), and construct a set of new *World-Domain* pairs for that variable. The worlds of this set correspond to the sub-CSP's created by the split.

Upon receiving new worlds and corresponding domains from the Split process, a Variable process notifies the Search process. This allows the Search process to maintain an administration of worlds where the constraint propagation algorithm still needs to be applied. The Search process coordinates the activities of the propagation network through the search tree, by issuing commands that start propagation in worlds that it knows about. In the current implementation of DICE, the Search process may consider starting propagation in a new world on two occasions: when propagation finishes in a certain world, and when a Variable process notifies that new worlds have become available.

3 Solver Configuration

From a user's point of view, solvers are configured in a small language. The following example shows a solver configuration for the SEND + MORE = MONEY puzzle.

```
VARIABLE s IS DiscreteDomain "1..9";
VARIABLE e IS DiscreteDomain "0..9";
...
VARIABLE y IS DiscreteDomain "0..9";
DRF IntegerArithmetic "s != e";
DRF IntegerArithmetic "s != n";
...
DRF IntegerArithmetic "r != y";
DRF IntegerArithmetic
   "             1000*s + 100*e + 10*n + d "
   "           + 1000*m + 100*o + 10*r + e "
   " = 10000*m + 1000*o + 100*n + 10*e + y ";
```

This example demonstrates the `VARIABLE` and `DRF` statements, used to set up the process network described in Sect. 2.3. Figure 2 shows the relevant part of the solver configuration language syntax, where {...} should be read as "followed by zero or more instances of the enclosed."

⟨*Solver*⟩ → ⟨*Statement*⟩ {⟨*Statement*⟩}

⟨*Statement*⟩ → `ENUM` ⟨*Identifier*⟩ {⟨*IdentifierList*⟩} `;`
→ `VARIABLE` ⟨*Identifier*⟩ `IS` ⟨*Identifier*⟩ ⟨*Specifier*⟩ `;`
→ `VARIABLE` ⟨*Identifier*⟩ `IS` ⟨*Identifier*⟩ `ENUM` ⟨*Identifier*⟩ `;`
→ `VARIABLE` ⟨*Identifier*⟩ `IS` ⟨*Identifier*⟩ `ENUM` ⟨*Identifier*⟩ ⟨*Specifier*⟩ `;`
→ `DRF` ⟨*Identifier*⟩ ⟨*Specifier*⟩ `;`
→ `WAIT;`
→ `PRINT` ⟨*Identifier*⟩ `;`
→ `SEARCH` ⟨*Identifier*⟩ ⟨*Specifier*⟩ ⟨*Identifier*⟩ ⟨*Specifier*⟩ `;`
→ `SOLUTIONS;`

⟨*Specifier*⟩ → ⟨*QuotedString*⟩ {⟨*QuotedString*⟩}
→ `ENUM` ⟨*QuotedString*⟩ {⟨*QuotedString*⟩}

Fig. 2. Solver configuration language syntax

The functionality of the Variable, DRF, Search and Split processes that constitute a solver is left unspecified to a large extent. Instead, these processes are placeholders for software components that complement the functionality offered by the DICE framework. This component-based design is reflected by the configuration language. As an example, consider the statement

```
VARIABLE s IS DiscreteDomain "1..9";
```

This creates a new Variable process. The identifier `s` is stored as a reference to this process. In the newly created process, a component in the "variable domain type" category is installed. The second identifier `DiscreteDomain` designates a particular component in that category. The specifier `"1..9"` has no meaning to

the framework. It is assumed that the component identified by `DiscreteDomain` knows how to deal with it.

The `DRF` statement creates a DRF process and has similar semantics. Although the specifier strings have no meaning, any component in the domain reduction function category should be able to report the names of the variables that it applies to, when requested by the framework. The mechanism for this is described in the next section. This information is used to make the connections to the Variable processes. Propagation starts as soon as the first domain reduction function is installed. The `WAIT` statement activates the termination detection algorithm to postpone execution of the next statement until propagation has finished. The `PRINT` statement outputs the contents of a variable.

The first identifier-specifier pair of the `SEARCH` statement specifies the component installed in the Split process. The second pair concerns the Search process. This statement starts the search for solutions by creating and activating these processes. The solutions can be retrieved by means of the `SOLUTIONS` statement.

The `ENUM` keyword can be used as a statement, to provide symbolic names for a range of integers starting at 0. The same symbolic name may not be used in more than one such statement. Also any specifier may be prefixed with `ENUM`. This effects a textual replacement of symbolic names by the corresponding integers. A third use of the `ENUM` keyword is in a `VARIABLE` statement. Here it is used to specify a set of symbolic names that should be used when printing integer values. In this case, textual replacement of symbolic names in the specifier is always carried out, and the specifier may be omitted to implicitly specify all values for which a symbolic name was provided.

4 Component Model

The interface between the DICE framework and the four categories of components is defined by four C++ abstract classes. As an example, consider the abstract class for components in the domain reduction function category:

```
class DRF_Component
{
public:
   static DRF_Component *UnitToDRF_Component( AP_Unit u );
   static DRF_Component *DescriptionToDRF_Component(
                                const char *type, const char *spec);
   virtual ~DRF_Component( );
   virtual AP_Unit ToUnit() const =0;
   virtual const std::vector<char*> &VariableNames( ) const =0;
   virtual int Compute( std::vector<Domain*> &arguments ) const =0;
   virtual int Termination( NodeType type,
                            std::vector<Domain*> &arguments );
};
```

In order to add a new component in any of the four categories, a subclass of the corresponding abstract class must be provided. In the case of the `DRF_ Component` example, the following provisions need to be made:

1. Through the `ToUnit` member function, objects in this class should be able to wrap themselves in a Manifold *unit*. By means of their unit representation, objects are transported through the process network.
2. Two static member functions `UnitToDRF_Component` and `DescriptionToDRF_Component` must be modified such that objects of the new class can be initialized from the unit representation generated by `ToUnit`, and from a textual representation, during parsing.
3. The actual functionality of the domain reduction function is embodied in the member functions `Compute` and `Termination`. `Compute` performs domain reduction as a part of the constraint propagation algorithm. `Termination` is called by the DICE framework upon termination of a single run of the propagation algorithm. Both member functions expect a vector of variable domain representations as an argument. The other argument to `Termination` indicates a solution, failure, or internal node. `Domain` is another abstract class, corresponding to the variable domain type component category.
4. In order for a DRF process to be able to relate the buffers associated with its input ports to the arguments of its domain reduction function, `DRF_Component::VariableNames` should supply a vector of variable names. The elements of this vector must correspond to the elements of the argument vector of the `Compute` and `Termination` member functions.

Most of this work can be automated. In particular, a utility was written that generates the code for the `Compute` member function from rule-based specifications of operators on small finite domains [2].

Each of the Variable, DRF, Search, and Split processes of a solver executes a command loop. All coordination protocols are implemented by these processes exchanging commands. These commands are *tuples* of the Manifold language. As an example, consider the reduction request of Sect. 2.3. This command is implemented by the tuple

$$<\texttt{CMD\_DRF\_REDUCE}, w, d>$$

where w and d are the unit representations of the world of the reduction request, and the new domain of the CSP variable in that world, respectively. The handler for `CMD_DRF_REDUCE` eventually invokes the `Compute` member function of the `DRF_Component` object installed in the receiving process.

5 Solver Building Blocks

In this section we describe a set of components that have been implemented in DICE to provide basic constraint solving capabilities.

Variable Domain Type

Components `IntegerInterval` and `DiscreteDomain` represent integer values by intervals and arbitrary sets of possible values, respectively. `DoubleInterval` provides a representation of real numbers by intervals, of which the bounds are double-precision floating point numbers.

Domain Reduction Function

`IntegerArithmetic` and `DoubleArithmetic` components provide support for simple arithmetic on integers and reals. In both cases, the component instances are initialized from a string, representing a relation between two polynomials. `IntegerArithmetic` supports linear polynomials only. `DoubleArithmetic` is based on the interval arithmetic operators of the JAIL library [7].

`IntegerPairs` is initialized by two variable names, and a set of integer pairs that constrains the values that these variables may assume. For example:

```
ENUM Word5 {LASER,HOSES,SAILS,SHEET,STEER};
VARIABLE position1 IS DiscreteDomain ENUM Word5;
VARIABLE position2 IS DiscreteDomain ENUM Word5;
DRF IntegerPairs ENUM "position1,position2 IN"
   "(LASER,SAILS),(LASER,SHEET),(LASER,STEER),"
   "(HOSES,SAILS),(HOSES,SHEET),(HOSES,STEER)";
```

The `MultiDRF` domain reduction function takes as an argument a sequence of DRF statements. For example:

```
DRF MultiDRF
   'DRF IntegerArithmetic "q2 != q7";'
   'DRF IntegerArithmetic "q2 - q7 != 5";'
   'DRF IntegerArithmetic "q2 - q7 != -5";'
;
```

The purpose of this component is to group several domain reduction functions into a single process of the distributed solver[2]. The `Compute` member function of a `MultiDRF` computes a fixpoint of its constituent domain reduction functions, by repeatedly applying these functions in sequence until a full sequence passes in which no changes to a variable domain are made.

As an alternative to having special components that implement optimizing search strategies, such as branch-and-bound, a `Minimize` component is available, which can be used to transform any search strategy into an optimization strategy. Figure 3 illustrates the use of this component. An additional variable is constrained such that its domain contains the outcome of an objective function. `Minimize::Termination` records the smallest value (singleton set) for this variable, encountered for a solution so far. `Minimize::Compute` enforces a dynamic constraint, stating that only improvements of this bound will be considered as new solutions.

Splitting Strategy

Currently, only one splitting strategy implementation exists, which splits a variable domain according to the default splitting strategy for its domain type. For `DiscreteDomain` the default is to generate a subdomain of size one for each element of the original domain. For `IntegerInterval` and `DoubleInterval` the default is bisection. The variable selection method is dynamic fail-first, selecting a variable with the smallest number of alternatives at each internal node.

[2] It is also possible to group several variables into a single process.

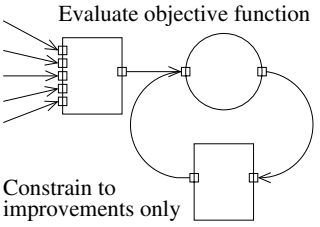


Fig. 3. The approach to optimization encouraged by DICE

Search Strategy

Three search strategies have been implemented:

1. A search strategy that reconstructs the search tree on the basis of the textual representation of the worlds that it learns about. When propagation finishes in an internal node, it waits for new worlds to become available. In a leaf node, or when new worlds are reported by a variable, the next world is selected according to a chronological backtracking strategy.
2. A search strategy that puts all new worlds in a queue. When new work is required by the propagation network, the world at the front of the queue is selected. This behaves in a breadth-first fashion.
3. A similar strategy, but employing a stack, instead of a queue. This strategy behaves in a depth-first fashion.

The latter two strategies have a target number of worlds where propagation is carried out at the same time. They keep injecting work into the network until this target is met.

Work in Progress

The set of components discussed in this section demonstrates the effectiveness of DICE on many standard examples. However, much work still needs to be done before we can claim to have a general-purpose constraint solver. Notably, our integer arithmetic presently cannot multiply variables, our constraints over the reals are currently limited to quadratic equations, our default splitting strategy does not yet properly take into account domain sizes of reals, and it is not yet possible to relate integer and real variables.

Also, in order that we can use DICE as a platform for experimenting with different search procedures, alternatives must be provided in the splitting strategy and search strategy component categories:

- chronological, static and dynamic fail-first, and maximum cardinality (most constrained first) variable selection methods,
- enumeration, bisection and middle-first value selection methods,
- limited discrepancy search.

Likely, the abstract classes for the component categories will still change. For example, it seems desirable to make the distinction between variable selection and value selection in splitting explicit. Also, there are many similarities between DRF and Split processes, and it may make sense to integrate the two component categories to indicate that splitting is a special case of domain reduction.

Currently, complex domain reduction functions can be configured from atomic reduction steps by means of the `MultiDRF` component. We plan to provide similar composition operators in the other component categories. Imagine a composite splitting strategy that dynamically changes from one basic splitting strategy to another, as search progresses. Other components that we plan to implement are:

- a box-consistency enforcer for constraints over the reals;
- domain reduction functions that take a set of constraints as a specifier and enforce these constraints inside a single reduction step by means of well-known propagation algorithms, such as AC-3 and AC-4;
- a domain reduction function that constrains an integer variable to count the number of violations in a set of soft constraints. Together with the `Minimize` component, this can be used to implement Max-CSP.

In addition to development of these components, in the next section we discuss our plans for extending the framework to support efficient sequential solvers and or-parallel search.

6 Discussion and Directions for Future Work

Many libraries, frameworks and toolkits for constraint solving exist. DICE is an implementation of the ideas presented in [3,10], and is distinguished from other platforms by the combination of an open-ended and inherently distributed design.

An example of a (non-distributed) system that is open-ended in a similar way is Figaro [8]. Also in Figaro new components for solver configuration can be added by inheriting from abstract classes corresponding to a number of (different) solver design dimensions. Often such systems are libraries. Inside the same application source we may both code new solver components, and employ the solvers constructed from these and other, readily available components. While at some point it will be desirable to provide an API to DICE, adding new components and using DICE for constraint solving are two distinct matters by design. This emphasizes that DICE is intended as a workbench environment, where constraint solvers are configured from basic building blocks. As a consequence, the application programmer is shielded from the Manifold system to a large extent. Only the component programmer is aware, for example, that components should be able to wrap themselves into Manifold units.

It could be argued that this scheme fails for components generated from a rule-based definition of reduction operators. These will be application specific, and should be part of the application source instead of being installed in DICE.

A solution would be to provide a generic component, to which the rules are submitted as a specifier.

The current framework demonstrates the effectiveness of the coordination-based approach, but does not yield efficient solvers. The primary reason is that communication is involved with every application of a DRF, and every node of the search tree. On top of this, the termination detection algorithm that we use has a considerable overhead, and a DICE solver is easily outperformed by any sequential solver on standard combinatorial benchmarks like the n-queens problem. Performance can already be improved by grouping CSP variables and (using `MultiDRF`) DRF's into single processes. In order to allow efficient sequential solvers to be constructed in DICE, we plan to take this one step further, and allow an arbitrary distribution of solver components over processes.

Yet the current implementation is well suited to solve CSP's of a distributed nature, where not performance, but the physical distribution of the constraints is the main issue. Imagine for example that constraints are defined by the contents of databases, running on different machines. DRF components could be written that query these databases to derive and enforce the constraints. Compared to the Asynchronous Backtracking algorithm, and derived algorithms for solving distributed CSP's described in [12], DICE focusses on distributed constraint propagation instead of distributed search. Even though propagation may be running in several nodes of the search tree simultaneously, the scheduling of nodes for propagation and subsequent splitting is a synchronous and sequential operation, and on each path of the search tree, variables are instantiated one after the other.

While DICE can already be applied in the area of distributed problems, reduction of turn-around time by exploiting parallelism would be an important additional motivation for distributed processing. The current implementation focusses on parallelism on the level of individual reduction steps only, but in general, a reduction step is too small to justify the overhead of coordination from a parallelization perspective. A straightforward way to achieve larger task granularity is to add support for or-parallel search. Search strategies that perform propagation in multiple nodes of the search tree at the same time are already available in DICE, but the propagation engine employs one process for each reduction step, and multiplexes the work in different worlds, to a large extent. One way to increase the capacity of the propagation network is to add an extra level of coordination, and replace the DRF processes by coordinators managing a number of DRF processes that handle the actual reduction requests in parallel.

In many cases, however, the size of the search tree will be too large to justify even having communication involved with every node of the search tree. Therefore support should be added for components that can autonomously explore a part of the search space. This can be seen as a form of splitting: a full solver can be incorporated as a splitting strategy that generates a new world for each solution, and possibly several worlds for the parts of the search space that still need to be explored. This way, splitting becomes a compute-intensive task, and parallel search becomes a matter of coordinating multiple Split processes. The

distributed constraint propagation algorithm can then be used for optimization among the results of these parallel solvers.

Finally, DRF's can be seen as incomplete constraint solvers. From that perspective, DICE is also a framework for solver cooperation. But compared to a solver cooperation language such as BALI (see for example [11]), and the system described in [9] the possibilities are limited: the cooperation scheme is always the distributed fixpoint computation, and the model of constraint solving is fixed to branch-and-prune search. However, in [10] it is explained how the distributed propagation algorithm can be forced to apply DRF's in a specific sequence. Also because DICE abstracts from variable domain types, the propagation algorithm could be used to transform problems, instead of domains of CSP variables. This would bypass the search mechanism, and allow for a large variety of models of constraint solving.

References

1. K.R. Apt. The Rough Guide to Constraint Propagation. In J. Jaffar (ed.) *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, LNCS 1713, pp. 1–23, Springer-Verlag, 1999.
2. K.R. Apt, E. Monfroy. Automatic Generation of Constraint Propagation Algorithms for Small Finite Domains. In J. Jaffar (ed.) *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, LNCS 1713, pp. 58–72, Springer-Verlag, 1999.
3. F. Arbab, E. Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In Porto, Roman (eds.) *Coordination Languages and Models*, LNCS 1906, pp. 115–132, Springer-Verlag, 2000.
4. F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In Ciancarini, Hankin (eds.) *Coordination Languages and Models*, LNCS 1061, pp. 34–56, Springer-Verlag, April 1996.
5. F. Arbab. *Manifold version 2: Language reference manual*, Technical report, Centrum voor Wiskunde en Informatica. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z.
6. E.W. Dijkstra. *Shmuel Safra's Version of Termination Detection* (Note EWD998).
7. F. Goualard. *JAIL – Just Another Interval Library*, Manual rev. 0.0.9, Institut de Recherche en Informatique de Nantes, January 2002.
8. M. Henz, T. Müller, K.B. Ng. Figaro: Yet Another Constraint Programming Library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, held in conjunction with ICLP'99.
9. P. Hofstedt. Better Communication for Tighter Cooperation. In Lloyd et al. (eds.) *Proceedings of the 1st International Conference on Computational Logic (CL 2000)*, LNAI 1861, pp. 342–357, Springer-Verlag, 2000.
10. E. Monfroy. A Coordination-based Chaotic Iteration Algorithm for Constraint Propagation. In Carroll, Damiani, Haddad, Oppenheim (eds.) *Proceedings of the 2000 ACM Symposium on Applied Computing*, pp. 262–269, ACM Press.
11. E. Monfroy, F. Arbab. Constraints Solving as the Coordination of Inference Engines. In Omicini, Zambonelli, Klusch, Tolksdorf (eds.) *Coordination of Internet Agents: Models, Technologies and Applications*, Springer-Verlag, 2001.
12. M. Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*, Springer-Verlag, 2001.

Visopt ShopFloor: Going Beyond Traditional Scheduling

Roman Barták

Charles University, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
bartak@kti.mff.cuni.cz

Abstract. Visopt ShopFloor is a generic scheduling system for solving complex scheduling problems. It differentiates from traditional schedulers by offering some planning capabilities. In particular, the activities to achieve the goal are planned dynamically during scheduling. In the paper, we give a motivation for the integration of planning and scheduling and we describe how such integration is realised in the scheduling engine of the Visopt ShopFloor system.

1 Introduction

Planning and scheduling are closely related areas but usually the problems from these areas are solved separately using a different technology. The planning task is to generate activities to achieve some goal while the scheduling task is to allocate the known activities to available resources and time. When both tasks are included in the real-life problem then usually the planning component generates the activities in advance and the separate scheduling component allocates the activities to the resources and time [18]. As we argued in [2], such separation is not appropriate if the problem environment is more complex and if the planning decisions are strongly influenced by the scheduling decisions (like the introduction of set-up activities with by-products). Our proposal how to solve the problems on the edge of planning and scheduling is based on the integration of planning and scheduling in a single solver [3].

In [6] we described our realisation of the integrated planning and scheduling system called Visopt ShopFloor. In this paper, we present the unique capabilities of this system using a particular example of the problem going beyond traditional scheduling.

The paper is organised as follows. First, we highlight the main features of the problem area and we describe one particular benchmark problem that can be solved by our system and that the conventional schedulers cannot handle. Then we present the technology and the basic ideas behind the solver. The paper is concluded with the results of the benchmark problem and we also show some results of real-life models.

B. O'Sullivan (Ed.): Constraint Solving and CLP, LNAI 2627, pp. 185–199, 2003.

2 The Problem

Traditional scheduling deals with the problem of allocating known activities to available resources and time. Usually, the resources are rather simple; they define a limited capacity for processing the activities. Either we have a unary resource, where only one activity can be processed at a time - this is sometimes called disjunctive scheduling. Or we have a cumulative resource where more activities can be processed in parallel provided that the resource capacity is not exceeded - this is called cumulative scheduling. Distinction of unary and cumulative resources is important because a resource constraint with stronger filtering can be defined for unary resources [1]. Despite the widespread use of unary and cumulative resources in traditional scheduling applications, neither one cares about alignment or sequencing of activities in the resource (we explain these notions later in Section 2.1).

In addition to the resource constraints restricting the allocation of activities, the traditional schedulers allow the definition of precedence constraints between the activities. Usually, the activities are grouped into tasks, where a prescribed sequence of activities must be followed. Therefore we are speaking about the task-centric models [9,2]. Job-shop scheduling [7] is a typical example of the task-centric view of the scheduling problem. Constraint-based scheduling [20] is more general by allowing precedence relations between arbitrary activities but it still requires knowing the activities in advance.

In the following sub-sections we show that the real-life problems are more complex than the above pure schema of the scheduling problems. In particular we give examples of the resources with more complex behaviour going beyond the unary/cumulative classification. We also explain why a fixed task schema is not appropriate to model some production processes. The section is concluded by a description of an example problem that contains some of these features.

2.1 Complex Resources

Unary (machine) and cumulative (store) resources are typical representatives of resources but in some production environments like process industries the behaviour of resources is more complex. In particular, alignment and sequencing of activities is important. In Visopt ShopFloor we are modelling batch production with a complex transition scheme.

Batch production means that the activities can be processed in parallel but if two activities overlap in time, they must start and finish at the same times. Such overlapping activities form a batch. In addition to the capacity restriction we also have a compatibility restriction, i.e., the activities are tagged by a type and only the activities of the same type can be processed in parallel.

In addition to batch production we can model a *complex transition scheme*. The resources are described using states and transitions between the states. At any time, a resource is in exactly one state and the state can be changed only according to the transition scheme. Moreover, the number of batches processed at each state can be limited. We now give some examples how the transition scheme is used to model behaviour of a real resource.

Let us consider a resource with two modes of production, parallel and serial. There is no restriction about the number of batches processed in the serial mode but exactly three batches are processed in the parallel mode. The restricted number of batches in the parallel mode is due to the following technological reason. Some by-product is outputted during the parallel production and this by-product is temporarily stored close to the machine. The temporal storage is full after three production batches and thus a recycling batch must be processed before the production can continue.

To make the transition scheme even more complex, we can consider that from time to time there must be a cleaning batch inserted. Moreover, cleaning cannot be done while some by-product is stored in the resource. We discuss the rules about insertion of the cleaning batch later in the paragraph about batch counters.

The above transition scheme can be easily described via a state transition graph where each state is tagged by a minimum and a maximum number of batches processed in this state (Figure 1).

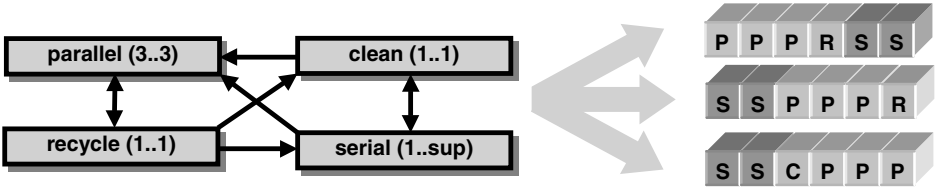


Fig. 1. Behaviour of many resources can be described using states with a minimum and a maximum number of batches per state (in brackets) and using a transition scheme between the states (left). This transition scheme must be followed during batch sequencing (right).

The transition scheme with the minimum and the maximum number of batches per state provides a flexible framework for modelling real-life resources. For example, it is easy to describe a learning curve of the worker. Let us assume that the worker needs first four batches to learn how to use the machine, i.e., duration of these batches is longer than duration of all following batches. We allow tagging the states by attributes, like duration and time windows, and these attributes are then applied to all batches of the state. Thus, the above worker can be modelled via a state transition scheme with two states: beginner and experienced (Figure 2). The batches processed in the beginner state are longer than the batches processed in the experienced state.

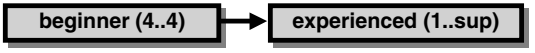


Fig. 2. State transitions can describe evolution of the resource, e.g., after a sequence of batches of given state, the resource irreversibly changes its state.

The above described transition scheme allows counting the batches of the same state. However, in many situations the users need to count batches of different states, e.g. to model insertion of the cleaning batch after a specified number of production batches. Thus, in Visopt ShopFloor we introduce the concept of a general batch counter that counts the batches across several states (Figure 3). This counter restricts further the sequencing of batches.

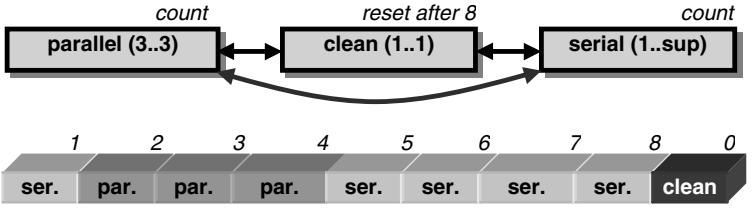


Fig. 3. Batch counters count batches across more states to model situations like forced cleaning after eight production (parallel or serial) batches.

It is hard or even impossible to model the above-described resources in the conventional scheduling. The main difficulty here is the transition scheme with the batch counters that forbid some transitions while force other transitions. It means that sequencing of batches is not arbitrary and the appearance of the batch depends on the allocation of other batches [2,17]. Thus the batches cannot be introduced in advance and it is more convenient to plan the batches dynamically during scheduling, i.e., to integrate planning and scheduling as we proposed in [3].

2.2 Resource Dependencies

In the conventional scheduling systems, the direct relations between the activities are described via precedence constraints. These precedence constraints can be seen as an abstraction of the item flow between the activities - the item must be produced before it can be consumed. However, a simple precedence relation is not enough to model many real-life dependencies. The item must be produced before it is consumed but sometimes the delay between the end of production and the start of consumption should not be too long. For example, the item is cooling after its production and some minimal temperature is required when the item is consumed. This can be modelled easily in constraint-based scheduling where the simple precedence relation is substituted by tighter constraints:

```
min_delay ≤ consumer_start - supplier_end ≤ max_delay.
```

The problem is when we do not know which activities are connected using the above precedence constraints, e.g., when there are several process routes for a single item. For example, assume that either the item is produced in a parallel mode when two machines co-operate and a worker is necessary (Figure 4 left), or the item is produced in a serial mode when the item flows from the first machine to the second machine and no worker is necessary (Figure 4 right). The structure of the production route is different in the above cases, namely different batches are used with different relations between them. Conventional scheduling requires one production route (task) to be chosen before scheduling (i.e. during planning). We propose to postpone this decision to the scheduling stage when more information about the batches is available [3].
Another difficulty of the conventional scheduling is modelling many-to-many relations between the batches, i.e., the batch has more suppliers and/or more consumers, and modelling recycling. In recycling, the set-up batch produces a by-product that can be used to satisfy some demands. Because the set-ups are not known

until the production batches are allocated, it is not possible to plan recycling in the foregoing planning stage.

Fig. 4. In the real-life factories, the item can be typically produced using more processing routes, e.g. via a parallel production when two machines run in parallel and a worker is required (left) or via a serial production when the item is pre-processed in the first machine and then finished in the second machine (right).

To address the above issues, we propose to describe supplier-consumer dependencies between the resources rather than to specify precedence relations between particular activities. Each supplier-consumer dependency is specified by the supplying and the consuming resource (and their states) and by the delay between the end of the supplying batch and the start of the consuming batch. When the dependency is established between two batches, the dependency describes also the quantity moved between the batches. Therefore a single supplying batch can be connected to more consuming batches and vice versa.

The supplier-consumer dependencies model naturally the item flow in the factory so they provide a declarative description of the processes in the factory. We can see them as a specimen for the precedence constraints that are posted when the batches of given type are introduced dynamically during scheduling (see Section 4). Figure 5 shows an example how the user describes the processes, i.e. the supplier-consumer dependencies using the graphical user interface of Visopt ShopFloor.

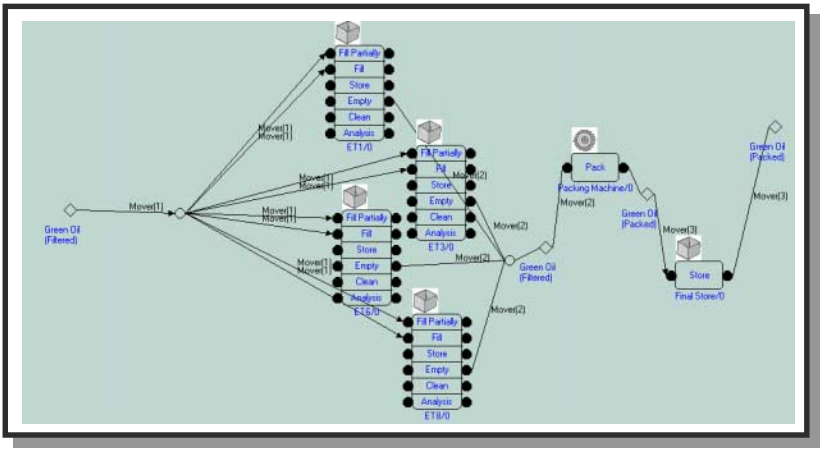


Fig. 5. Visopt ShopFloor graphical user interface describing an item flow.

2.3 The Task at a Glance

The Visopt ShopFloor system concentrates primarily on the problems going beyond traditional scheduling. Let us now summarise the task solved by the Visopt ShopFloor by giving a particular benchmark example.

Let us consider a small factory with two machines, r1 (Figure 3) and r2 (Figure 1), producing a single final item. These machines run either in a parallel mode or in a serial model (Figure 4). In the parallel mode, the batches of both machines run in parallel and a worker is required. One final item is outputted from the batch and duration of this batch depends on the experience of the worker (see below). In the serial mode, the machine r1 pre-processes the item (3 time units) that is finished in the machine r2 (3 time units). There is no delay for moving the item from r1 to r2.

During the parallel production, a by-product is produced. This by-product can be recycled only on the machine r2 and we need three by-products to get a single final item. Recycling takes 2 time units and it must be done immediately after the three batches of the parallel processing (Figure 1).

Both machines require cleaning after eight production batches or sooner (Figure 3) and the cleaning must be done at the same time on both machines. At the beginning, both machines are clean.

The worker, who is necessary for parallel processing, is a beginner. After four production batches, the worker becomes experienced (Figure 2). The parallel production takes 3 time units for the beginner and 2 time units for the experienced worker. Moreover, the worker is available only in the following time windows (0..10), (30..40), (60..70).

The task is to plan/schedule production starting from time 0 in such a way that 5 final items are ready in time 20 and additional 25 items are ready in time 100.

As we can see from the above example the goal of the system is to find out the batches that are necessary to satisfy the demands (planning) and to allocate these batches to available resources (scheduling). A plan/schedule for a given time period is returned to the user.

In this paper we discuss only feasibility issues, but the Visopt ShopFloor does optimisation based on cost as well. The conventional schedulers use some objective function like makespan, tardiness, or earliness to define quality of the schedule. However, in real-life environment the schedule quality is usually subjective, evaluated by the plant persons. To model these subjective criteria we use the cost parameters attached to batches, transitions, dependencies etc. The total cost is then used to guide scheduling, for details see [6].

3 The Technology

Traditional scheduling technology is either based on special scheduling algorithms [7] or some general schema like constraint-based scheduling [20] is applied. If the activities are known in advance then it is quite natural to model the scheduling problem as a constraint satisfaction problem (CSP). However if the planning component is present then the static approach is hardly applicable due to variability of possible plans [16]. Some approaches try to fit the planning problem into the static concept of CSP via dummy activities [8,17] but it works only when the planning

branching does not lead to many different structures of the plan. Other researchers propose to use some generalised concept of CSP that provide more dynamic features like Dynamic CSP [14] or Structural CSP [15].

In the Visopt ShopFloor system, we solve the scheduling problems where the appearance of the activity depends on allocation of other activities. In terms of CSP it means that the existence of some variables and constraints depends on assigning a value to another variable. Moreover, the variable/constraint disappears from the system only when the original assignment is withdrawn, i.e., during backtracking. Having this in mind we decided to use the existing technology of Constraint Logic Programming (CLP) in the way this framework was originally defined [10], i.e., the constraints are used to reduce the search space of the logic program.

Opposite to the standard CSP technique (i.e., define the variables and the constraints first and then do labelling) we propose to interleave the labelling stage with the introduction of new variables and constraints. Basically, it means that we model the planning decisions (branching) using the disjunctive constraints (constructive disjunction). When some element of the disjunction is selected then the system automatically introduces other variables and constraints corresponding to the selected planning branch. This gives us the freedom to define different sets of variables and constraints in different branches of the search tree, i.e., to explore different plans. Thus planning decisions are resolved during scheduling.

4 The Solver

The Visopt ShopFloor system consists of two independent parts: the ShopFloor graphical modelling environment and the scheduling engine (see Figure 6).

In the *ShopFloor*, the user specifies completely the problem to be solved. In particular he or she describes the available resources, i.e., their states and transitions, the item flow, i.e., the supplier-consumer dependencies (see Figure 5), and the customer orders (demands). Data can be entered and modified manually or they can be extracted automatically from the databases of ERP systems. The ShopFloor module generates the problem description in the form of a text file called a factory model that is passed to the solver.

The *factory model* contains a complete description of the problem (resources, dependencies, and orders) in a human readable form. It means that the factory model can be explored, prepared, and modified in an arbitrary text editor. This file is the only input to the scheduling engine.

The *scheduling engine* (the solver) first generates a constraint model from data (from the factory model) and then it searches for the solution. The solver has a modular architecture so it is possible to add a new module describing a new type of resource. Also, the search strategy is a separate module so it can be exchanged by a new strategy. The scheduling engine returns the plan/schedule into the ShopFloor that displays it in the form of a Gantt chart.

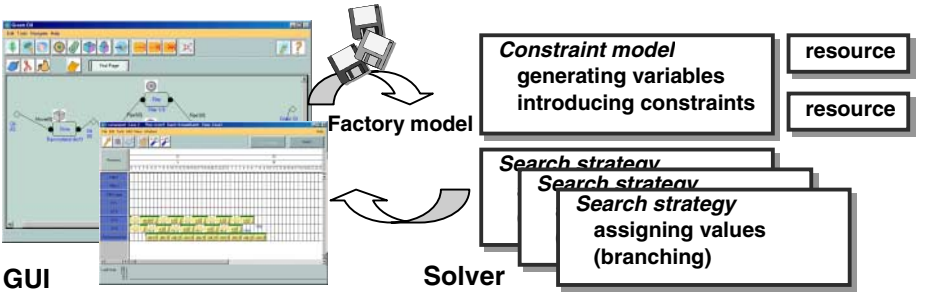


Fig. 6. The Visopt ShopFloor system architecture consists of two independent modules: the graphical modelling environment (left) and the scheduling engine (right).

4.1 The Constraint Model

The traditional static constraint models are defined by the set of variables, their domains, and by the set of constraints restricting possible combinations of values. As we discussed in Section 3, we need a more dynamic approach to CSP, namely, the variables and the constraints are introduced as search progresses. There exist some static approaches to overcome difficulties with the unknown set of variables/constraints based on dummy variables and deactivated constraints [8,17]. Unfortunately, such approaches lead to huge models so they cannot be used to model the problem completely statically. Nevertheless, we use the dummy variables partially to do look-ahead for planning decisions (via constructive disjunction, see Section 3) and to realise the idea of active decision postponement [11].

The constraint model in the Visopt solver is a piece of code responsible for introduction of variables and constraints. The basic idea is as follows: at the beginning we introduce only the objects that are known, i.e., the customer orders. As these customer orders should be satisfied, we also start dependencies to the resources that can produce the ordered items. When the actual supplier is found (this is usually decided during labelling), we need to find suppliers for this supplier etc. To summarise it: if there is a planning decision, i.e. the decision about what objects should be part of the plan/schedule, we introduce all of them (via dependencies). Together with these objects, the relevant constraints are posted so we can exploit the power of constraint propagation. Let us now describe some details about what variables and what constraints are used.

The Slot Representation

Opposite to most scheduling systems that use the task-centric model of the problem, we decided to apply the resource-centric model because it simplifies modelling of the complex transition schemes [2]. It means that the batches are grouped per resource rather than per task. Of course, we do not know the batches in the resource at the beginning so we use a chain of empty slots to represent the schedule for each resource. Opposite to the slots used in the timetabling applications, the slots in our

system may slide in time and they may have variable duration. The only restriction is that the ordering of slots must be preserved (due to the transition constraints).

Each slot has some attributes like the start time, the end time, and the duration represented as finite domain variables. Also, there is a special variable describing the type of the batch (the state) that can be filled in the slot. When this state variable becomes a singleton we know the batch in the slot - we say that the slot is filled by the batch. This may introduce other variables that are specific for the particular batch, e.g. quantities of consumed and produced items. Naturally, all the slot variables are connected via constraints describing the time windows etc. and these constraints can be posted even if the batch in the slot is not known yet. Moreover, there can be also constraints between the neighbouring slots to describe the transition scheme.

To model the minimum and the maximum batches per state we introduce a special variable called a serial number that "counts" the batches of the same state. This variable participates in the transition constraints so it may force the state change when the maximum number of batches is reached or it may forbid the state change when the minimum number of batches is not reached. Figure 7 illustrates this mechanism, for technical details see [5]. The same mechanism is used to model the batch counters.

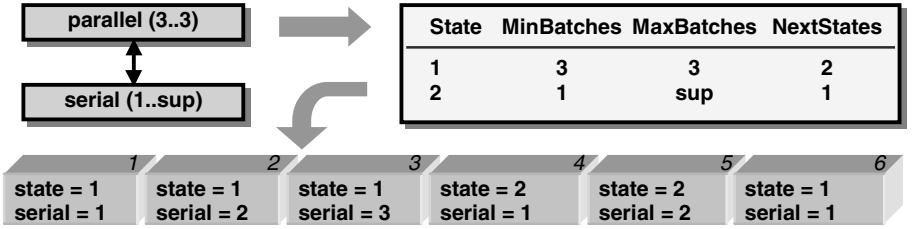


Fig. 7. The transition scheme (top left) is modelled using the serial numbers and the special transition constraints defined over the transition table (top right).

As we mentioned above the slots are also introduced dynamically which saves some memory. In fact, a new slot is attached to the end of the slot list when there is a demand to the resource but there is no free slot to satisfy this demand. Note however, that it does not mean that the new slot will be filled by the coming batch that caused its introduction. Perhaps some waiting (not yet allocated) batch or a future batch overhauls it or the slot will stay empty if we find later that the batch is not necessary. Still, the ordering of slots is fixed so it is not possible to introduce a new slot in-between two existing slots. Thus, deciding to which slot the batch is allocated corresponds to the decision about the absolute ordering of batches in the resource. This view is similar to the idea of permutation based scheduling presented in [21]. The main difference of our approach is that we can solve problems where the appearance of the batch depends on allocation of other batches. In particular, the structure of the batches in the resource depends on the demands from other resources as well as on the transition scheme for the resource.

Notice that although we do not know the batches in the resource, thanks to the slot representation we can post many constraints in advance and thus to use the power of constraint propagation. The main reason for using the slot representation is modelling complex transition schemes.

Dependencies

The slots of different resources are connected via dependencies modelling the supplier-consumer relations. Because the dependency is closely related to the item we cannot introduce the dependency until we know the item and its quantity. As described in the previous section, the variable specifying the item quantity is generated as soon as we know the batch - the state - in the slot. At the same time we can start the dependencies from the given slot.

Assume that we have an input item defined for the batch in the slot. Dependencies should connect this batch with all the supplying batches. It is possible to post the dependency to every slot that can be filled by the supplying batch. However, this eager method has huge memory consumption when applied to large-scale problems with hundreds or thousands of slots. Thus, we use a more lazy method that posts a minimal number of dependencies covering the required quantity. Typically, these dependencies go to the first "free" slot of every possible supplying resource. If we find later that the slot cannot be filled by the supplying batch then we move the dependency to the next slot and so on (see Figure 8). This is realised by setting the quantity in the dependency to zero (so the constraint connecting the slot times "evaporates") and by introducing a new dependency going to the next slot. If no supplying batch is found in the resource then the dependency to the resource is finally made empty and the supplying batch must be found in another resource. Other dependencies can be introduced as soon as we find that the dependencies generated so far are not enough to cover the requested quantity (e.g. because some of them have been made empty).

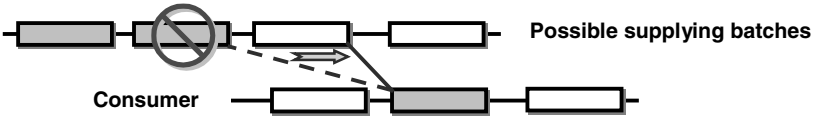


Fig. 8. The dependency generator introduces the dependencies to the first possible slot (from left) of each candidate supplier. If the slot is not dependent (⃠) then the dependency is moved to the next free slot etc.

For each slot, the system maintains the links to all non-empty dependencies going to this slot. These links are used during scheduling for decision about which batch will be filled in the slot (see Section 4.2). Of course, we also know all the dependencies going to a particular resource so we can post special ordering constraints between the dependencies [12]. Because, every dependency defines a demand for one batch, these constraints decide about the ordering of the batches in the resource. Note finally that these global constraints must be open to accept incoming batches [4].

4.2 The Search Strategy

The Visopt constraint model is responsible for introduction of all variables and all constraints. The scheduling strategy then decides about the batches in the slots and about the connections between the slots. Recall that the slots are introduced from left to right and the dependencies are started first from the orders. Thus, the labelling procedure must be aware of this ordering.

In the Visopt scheduling engine, we are closing the slots (decide about the batch types in the slots) in slices going from left (past) to right (future). We call processing the slice a *scheduling step*. The decision whether the slot belongs to the slice or not is done using the time variables in the slot. In the slice, the slots are closed in the order-to-purchase ordering.

When the slot is selected, the second problem is which batch should be filled in the slot. This decision is naturally based on the dependencies going to the slot. The labelling strategy first selects the "best" dependencies and then it connects them to the slot. This is done via setting the quantity variable in the dependency to be greater than zero (the maximum value is tried first). The relation "better" between the dependencies is defined using the time of the dependency (the earliest time is preferred), using the cost information (the smallest cost is preferred), and using other heuristic criteria. Recall that when some dependencies are fixed to the slot, the incompatible dependencies are moved automatically to the next possible slot (see Figure 9). Thus, the decision about the ordering of dependencies is equivalent to the decision about the ordering of batches in the slots. After selecting the dependencies in the slot, the labelling strategy assigns a value to the state variable. In the end of each scheduling step, the time variables in the closed slots are labelled - the earlier times are preferred.

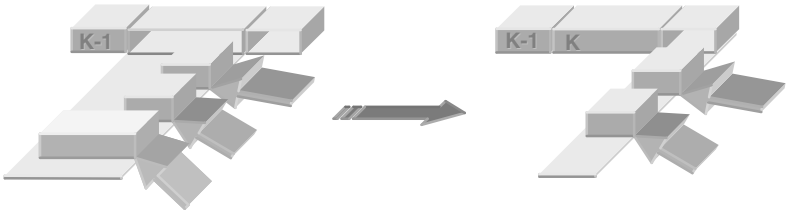


Fig. 9. The basic decision of the scheduling strategy is which dependencies will go to a particular slot, i.e., which batch will be filled in the slot (left). Then the incompatible dependencies are moved automatically to the next slot (right).

As described in the above paragraphs, the scheduling strategy is based on depth-first search (backtracking). Because the variable ordering is selected carefully, the scheduling strategy knows nothing about the dynamic character of the constraint model. Every time the labelling procedure attempts to assign a value to the variable, the variable is present in the system. Still, notice that the structure of the variables is different in different branches of the search tree (because different dependencies are introduced). This dynamic character complicates the usage of more advanced search techniques like the limited discrepancy search.

So far we have enhanced the base backtracking mechanism by *user defined backjumping*. We have developed this new search algorithm using the following idea. If we cannot satisfy the demand in a given scheduling step then we leave this demand open (no batches for the demand are closed) and we try to satisfy it in the next step. This is realised by jumping to a pre-selected variable after failure, unassigning this variable and continuing with labelling of the other variables. This unassigned variable will be tried in the next scheduling step again. This algorithm is similar to graph-directed backjumping, the main difference is that the back jump is realised only for some pre-selected variables.

5 The Results

The Visopt ShopFloor scheduling engine is completely implemented in SICStus Prolog (currently we use the version 3.8.7). It has been tested in several pilot projects in one of the biggest chemical enterprises in Europe, in one of the biggest and famous candy producers in The Netherlands, and in one of the biggest dairies in Israel among others. We are not aware of any other scheduling system that can model and solve the problems described in this paper, so we cannot compare our solver to existing schedulers. In this section we first show the plans produced by our solver for the problem from Section 2.3 and then we summarise the results of some real-life models.

The problem from Section 2.3 requires both planning, i.e., deciding which batches are necessary to satisfy the demands, and scheduling, i.e., allocating the batches to available resources. Recall, that there are three different ways of producing the final item, namely parallel production, serial production, and recycling. Moreover, there is a complex transition scheme describing the resource including insertion of a cleaning batch after a specified number of the production batches. Last but not least, there is a worker that influences the timing of the parallel production.

Figure 10 shows a Gantt chart of the plan produced by our solver (3 seconds on 1.7 GHz Mobile Pentium 4). We can see that this plan satisfies all the production rules, in particular using the recycling and the cleaning batches. Also the duration of the parallel batches decreases when the worker became experienced (roughly at time 35).

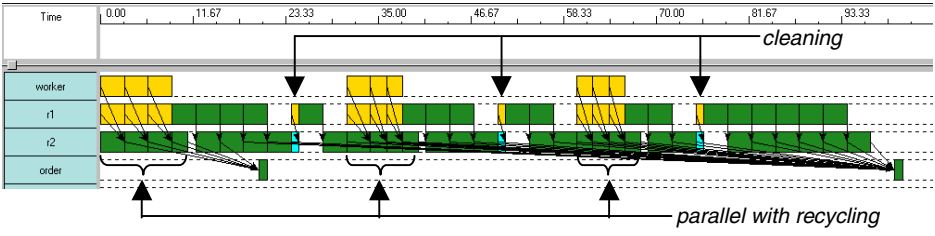


Fig. 10. The Gantt chart of the plan for the problem from Section 2.3

We have relaxed the restriction about parallel cleaning for both machines to see if the production can be more efficient. Figure 11 shows that the resulting plan is shorter because the cleaning batches can be scheduled asynchronously (planning took 3 seconds on 1.7 GHz Mobile Pentium 4).

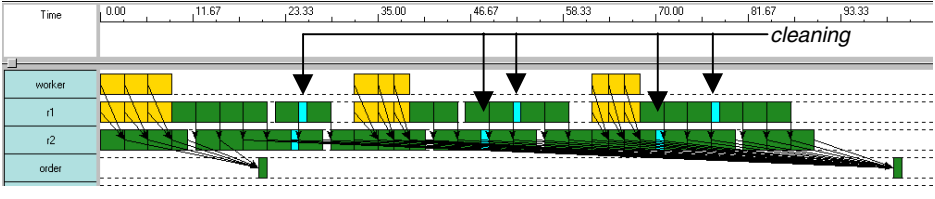


Fig. 11. The Gantt chart of the plan for the problem from Section 2.3 where the cleaning is asynchronous.

To show the size and the complexity of the real-life problems we have prepared a summary of some pilot problems solved by the Visopt ShopFloor system. These test problems are based on the real-life production lines so the actual models and the plans are confidential. Thus we can present only some global parameters of the models. In particular, Table 1 shows the model size and the runtime results. For each model we include the number of resources, the total number of states in the resources, the number of orders together with the ordered quantity, the number of items, and duration of the scheduled period. The ordered quantity corresponds roughly to the size of domains of the quantity variables - we track every quantity unit in the production. The schedule duration corresponds to the size of the domains for the time variables - it shows the resolution of scheduling (e.g. 10.080 time units means a one week production with a minute resolution). The solution is characterised by the runtime and by the solution size measured in the number of batches and in the number of dependencies.

Table 1. Model and solution size for some test problems. Runtime is measured in seconds on a Mobile Pentium 4 1.7 GHz.

	model					solution		
	res.	states	orders # / quantity	items	duration time units	runtime sec.	batches	dependencies
1	19	334	1 / 144000	47	10080	105	1000	1441
2	28	115	1 / 50	34	8640	79	256	310
3	22	677	9 / 7600	56	3168	40	651	898
4	57	704	256 /196748	45	840	77	990	1428
5	34	574	45 / 88485	294	11520	2339	5807	10175

For comparison, the state of the art schedulers handle about 20.000 batches [personal communication to Wim Nuijten from ILOG] but all these batches are known in advance. In planning, the size of plans is measured in tens of actions [13]. As Table 2 shows we can handle problems with hundreds to thousands batches. However, when the number of batches increases, the large memory consumption becomes a limitation. Thus in the model 5 there is a trade off between the memory consumption and the runtime. This observation confirms our claim that such problems cannot be handled in a fully static way using the dummy activities because then the memory consumption becomes critical.

To summarise the results, we can handle problems much larger than the traditional planning problems and close to the size of the problems in conventional static scheduling. Recall that the input to the Visopt engine consists of the model of the factory and the list of demands. All the batches are introduced (planned) during the problem solving and allocated to the resources (scheduling). Thus, we are basically solving a (limited) planning problem under time and resource constraints. Moreover, our system can handle more complex resource constraints (a transition scheme) and resource dependencies (recycling, many-to-many relations etc.) than the conventional schedulers can.

6 Conclusion

In this paper we described the heart of the Visopt ShopFloor system - the scheduling engine. The integrated planning component is the main difference of our system from the conventional schedulers. We are not aware of any other system doing such deep integration so it is hard to compare Visopt to existing systems. As we showed in Section 2, the planning component provides flexibility that cannot be reached by conventional scheduling software. The unique features of Visopt, which the other scheduling systems cannot cover, include modelling of complex transition schemes for resources, modelling of an arbitrary dependency structure of the factory, modelling of set-ups, cleaning, and maintenance including by-products, and modelling of process and item alternatives. Moreover, Visopt ShopFloor attempts to be a general scheduler where the customer describes the problem in a declarative way and the system generates schedules automatically. Other scheduling software is either provided as a toolkit (e.g. ILOG Scheduler), so the particular scheduler must be programmed using this toolkit, or the software solves a particular scheduling problem but it cannot be extended to other problem areas. Opposite to these systems, Visopt ShopFloor [22] provides intuitive graphical modelling environment independent of the solver, generality covering many scheduling problems, and extendibility via adding new types of resources.

Acknowledgements. The research is supported by the Grant Agency of the Czech Republic under the contract 201/01/0942 and by Visopt B.V. I would like to thank the reviewers of the paper for useful comments and to Petr Štěpánek and Ondřej Čepek for proofreading.

References

1. Baptiste, P. and Le Pape, C.: Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling, in *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group* (1996).
2. Barták, R.: Conceptual Models for Combined Planning and Scheduling. *Electronic Notes in Discrete Mathematics,* Volume 4, Elsevier (1999).
3. Barták, R.: Dynamic Constraint Models for Planning and Scheduling Problems. *Proceedings of the ERCIM/CompulogNet Workshop on Constraint Programming*, LNAI Series, Springer Verlag (2000).
4. Barták, R.: Dynamic Global Constraints in Backtracking Based Environments, in *Annals of Operations Research 118*, Kluwer (2003) 101–119. Forthcoming.
5. Barták, R.: Modelling Resource Transitions in Constraint-based Scheduling. In. W.I. Grosky, F. Plášil (eds.): *Proceedings of SOFSEM 2002: Theory and Practice of Informatics,* LNCS 2540, Springer Verlag (2002) 186–194.
6. Barták, R.: Visopt ShopFloor: On the Edge of Planing and Scheduling. In P. van Hentenryck (ed.): *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002),* LNCS 2470, Springer Verlag, Ithaca, (2002) 587–602.
7. Brucker P. *Scheduling Algorithms*. Springer Verlag (2001).
8. Beck, J.Ch. and Fox, M.S.: Scheduling Alternative Activities. *Proceedings of AAAI-99*, USA (1999) 680–687.

9. Brusoni, V., Console, L., Lamma. E., Mello, P., Milano, M., Terenziani, P.: Resource-based vs. Task-based Approaches for Scheduling Problems. *Proceedings of the 9th ISMIS96*, LNCS Series, Springer Verlag (1996).
10. Gallaire, H.: Logic Programming: Further Developments, in: IEEE Symposium on Logic Programming, Boston, IEEE (1985).
11. Joslin, D. and Pollack M.E.: Passive and Active Decision Postponement in Plan Generation. *Proceedings of the Third European Conference on Planning* (1995).
12. Laborie P.: Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results. In *Proceedings of 6th European Conference on Planning,* Toledo, Spain (2001), 205–216.
13. Long D. and Fox. M. International Planning Competition 2002. Toulouse, France (2002). http://www.dur.ac.uk/d.p.long/competition.html
14. Mittal, S. and Falkenhainer, B.: Dynamic Constraint Satisfaction Problems. *Proceedings of AAAI-90*, USA (1990), 25–32.
15. Nareyek, A.: Structural Constraint Satisfaction. *Proceedings of AAAI-99 Workshop on Configuration* (1999).
16. Nareyek, A.: AI Planning in a Constraint Programming Framework. *Proceedings of the Third International Workshop on Communication-Based Systems* (2000).
17. Pegman, M.: Short Term Liquid Metal Scheduling. *Proceedings of PAPPACT98 Conference*, London (1998), 91–99.
18. Srivastava B. and Kambhampati S.: Scaling up Planning by teasing out Resource Scheduling. Technical Report ASU CSE TR 99–005, Arizona State University (1999).
19. Van Hentenryck, P.: *Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge, Mass. (1989).
20. Wallace, M.: Applying Constraints for Scheduling, in: *Constraint Programming*, Mayoh B. and Penjaak J. (eds.), NATO ASI Series, Springer Verlag (1994).
21. Zhou, J.: A Permutation-Based Approach for Solving the Job-Shop Problem. *Constraints*, vol. 2 no. 2 (1997), 185–213.
22. Visopt B.V. http://www.visopt.com

Author Index