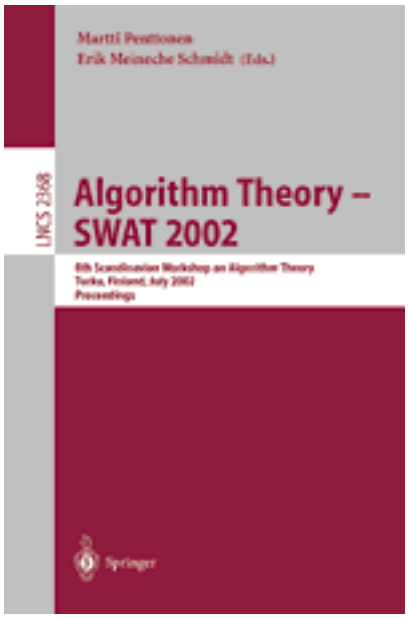


M. Penttonen, E. Meineche Schmidt (Eds.):

Algorithm Theory - SWAT 2002

8th Scandinavian Workshop on Algorithm Theory, Turku, Finland, July 3-5, 2002. Proceedings

LNCS 2368

Ordering Information

Table of Contents

Invited Speakers

Combining Pattern Discovery and Probabilistic Modeling in Data Mining
Heikki Mannila
LNCS 2368, p. 19
Abstract | Full article in PDF (33 KB)

Scheduling

Time and Space Efficient Multi-method Dispatching
Stephen Alstrup, Gerth Stølting Brodal, Inge Li Gørtz, and Theis Rauhe
LNCS 2368, p. 20 ff.
Abstract | Full article in PDF (160 KB)

Linear Time Approximation Schemes for Vehicle Scheduling
John E. Augustine and Steven S. Seiden
LNCS 2368, p. 30 ff.
Abstract | Full article in PDF (160 KB)

Minimizing Makespan for the Lazy Bureaucrat Problem
Clint Hepner and Cliff Stein
LNCS 2368, p. 40 ff.
Abstract | Full article in PDF (149 KB)

A PTAS for the Single Machine Scheduling Problem with Controllable Processing Times
Monaldo Mastrolilli
LNCS 2368, p. 51 ff.
Abstract | Full article in PDF (154 KB)

Computational Geometry

Optimum Inapproximability Results for Finding Minimum Hidden Guard Sets in Polygons and Terrains
Stephan Eidenbenz
LNCS 2368, p. 60 ff.
Abstract | Full article in PDF (138 KB)

Simplex Range Searching and k Nearest Neighbors of a Line Segment in 2D
Partha P. Goswami, Sandip Das, and Subhas C. Nandy
LNCS 2368, p. 69 ff.
Abstract | Full article in PDF (208 KB)

Adaptive Algorithms for Constructing Convex Hulls and Triangulations of Polygonal Chains
Christos Levcopoulos, Andrzej Lingas, and Joseph S.B. Mitchell
LNCS 2368, p. 80 ff.
Abstract | Full article in PDF (149 KB)

Exact Algorithms and Approximation Schemes for Base Station Placement Problems
Nissan Lev-Tov and David Peleg
LNCS 2368, p. 90 ff.
Abstract | Full article in PDF (174 KB)

A Factor-2 Approximation for Labeling Points with Maximum Sliding Labels
Zhongping Qin and Binhai Zhu
LNCS 2368, p. 100 ff.
Abstract | Full article in PDF (153 KB)

Optimal Algorithm for a Special Point-Labeling Problem
Sasanka Roy, Partha P. Goswami, Sandip Das, and Subhas C. Nandy
LNCS 2368, p. 110 ff.
Abstract | Full article in PDF (171 KB)

Random Arc Allocation and Applications
Peter Sanders and Berthold Vöcking
LNCS 2368, p. 121 ff.
Abstract | Full article in PDF (159 KB)

On Neighbors in Geometric Permutations
Micha Sharir and Shakhar Smorodinsky
LNCS 2368, p. 131 ff.
Abstract | Full article in PDF (152 KB)

Graph Algorithms

Powers of Geometric Intersection Graphs and Dispersion Algorithms
Geir Agnarsson, Peter Damaschke, and Magnús M. Halldórsson
LNCS 2368, p. 140 ff.
Abstract | Full article in PDF (179 KB)

Efficient Data Reduction for DOMINATING SET: A Linear Problem Kernel for the Planar Case
Jochen Alber, Michael R. Fellows, and Rolf Niedermeier
LNCS 2368, p. 150 ff.
Abstract | Full article in PDF (242 KB)

Planar Graph Coloring with Forbidden Subgraphs: Why Trees and Paths Are Dangerous
Hajo Broersma, Fedor V. Fomin, Jan Kratochvíl, and Gerhard J. Woeginger
LNCS 2368, p. 160 ff.
Abstract | Full article in PDF (158 KB)

Approximation Hardness of the Steiner Tree Problem on Graphs
Miroslav Chlebík and Janka Chlebíková
LNCS 2368, p. 170 ff.
Abstract | Full article in PDF (146 KB)

Robotics

LNCS 2368, p. 260 ff.
Abstract | Full article in PDF (159 KB)

Analysis of Heuristics for the Freeze-Tag Problem
Marcelo O. Sztainberg, Esther M. Arkin, Michael A. Bender, and Joseph S.B. Mitchell
LNCS 2368, p. 270 ff.
Abstract | Full article in PDF (175 KB)

Approximation Algorithms

Approximations for Maximum Transportation Problem with Permutable Supply Vector and Other Capacitated Star Packing Problems
Esther M. Arkin, Refael Hassin, Shlomi Rubinstein, and Maxim Sviridenko
LNCS 2368, p. 280 ff.
Abstract | Full article in PDF (150 KB)

All-Norm Approximation Algorithms
Yossi Azar, Leah Epstein, Yossi Richter, and Gerhard J. Woeginger
LNCS 2368, p. 288 ff.
Abstract | Full article in PDF (147 KB)

Approximability of Dense Instances of NEAREST CODEWORD Problem
Cristina Bazgan, W. Fernandez de la Vega, and Marek Karpinski
LNCS 2368, p. 298 ff.
Abstract | Full article in PDF (152 KB)

Data Communication

Call Control with k Rejections
R. Sai Anand, Thomas Erlebach, Alexander Hall, and Stamatis Stefanakos
LNCS 2368, p. 308 ff.
Abstract | Full article in PDF (95 KB)

On Network Design Problems: Fixed Cost Flows and the Covering Steiner Problem
Guy Even, Guy Kortsarz, and Wolfgang Slany
LNCS 2368, p. 318 ff.
Abstract | Full article in PDF (164 KB)

Packet Bundling
Jens S. Frederiksen and Kim S. Larsen
LNCS 2368, p. 328 ff.
Abstract | Full article in PDF (148 KB)

Algorithms for the Multi-constrained Routing Problem
Anuj Puri and Stavros Tripakis
LNCS 2368, p. 338 ff.

Abstract | Full article in PDF (178 KB)

Computational Biology

Computing the Threshold for q-Gram Filters

Juha Kärkkäinen
LNCS 2368, p. 348 ff.
Abstract | Full article in PDF (173 KB)

On the Generality of Phylogenies from Incomplete Directed Characters
Itsik Pe'er, Ron Shamir, and Roded Sharan
LNCS 2368, p. 358 ff.
Abstract | Full article in PDF (193 KB)

Data Storage and Manipulation

Sorting with a Forklift
M.H. Albert and M.D. Atkinson
LNCS 2368, p. 368 ff.
Abstract | Full article in PDF (117 KB)

Tree Decompositions with Small Cost
Hans L. Bodlaender and Fedor V. Fomin
LNCS 2368, p. 378 ff.
Abstract | Full article in PDF (166 KB)

Computing the Treewidth and the Minimum Fill-in with the Modular Decomposition
Hans L. Bodlaender and Udi Rotics
LNCS 2368, p. 388 ff.
Abstract | Full article in PDF (164 KB)

Performance Tuning an Algorithm for Compressing Relational Tables
Jyrki Katajainen and Jeppe Nejsum Madsen
LNCS 2368, p. 398 ff.
Abstract | Full article in PDF (147 KB)

A Randomized In-Place Algorithm for Positioning the kth Element in a Multiset
Jyrki Katajainen and Tomi A. Pasanen
LNCS 2368, p. 408 ff.
Abstract | Full article in PDF (165 KB)

Paging on a RAM with Limited Resources
Tony W. Lai
LNCS 2368, p. 418 ff.
Abstract | Full article in PDF (105 KB)

Online publication: June 21, 2002
helpdesk@link.springer.de

An Efficient Quasidictionary

Torben Hagerup[1] and Rajeev Raman[2⋆]

[1] Institut für Informatik, Johann Wolfgang Goethe-Universität Frankfurt, D–60054 Frankfurt am Main. hagerup@ka.informatik.uni-frankfurt.de

[2] Department of Maths and Computer Science, University of Leicester, Leicester LE1 7RH, UK. r.raman@mcs.le.ac.uk

Abstract. We define a *quasidictionary* to be a data structure that supports the following operations: check-in(v) inserts a data item v and returns a positive integer tag to be used in future references to v; check-out(x) deletes the data item with tag x; access(x) inspects and/or modifies the data item with tag x. A quasidictionary is similar to a dictionary, the difference being that the names identifying data items are chosen by the data structure rather than by its user. We describe a deterministic quasidictionary that executes the operations check-in and access in constant time and check-out in constant amortized time, works in linear space, and uses only tags bounded by the maximum number of data items stored simultaneously in the quasidictionary since it was last empty.

1 Introduction

Many data structures for storing collections of elements and executing certain operations on them have been developed. For example, the red-black tree [14] supports the operations insert, delete, and access, among others, and the Fibonacci heap [10] supports such operations as insert, decrease, and extractmin. An element is typically composed of a number of fields, each of which holds a value. Many common operations on collections of elements logically pertain to a specific element among those currently stored. E.g., the task of a delete operation is to remove an element from the collection, and that of a decrease operation is to lower the value of a field of a certain element. Therefore the question arises of how a user of a data structure can refer to a specific element among those present in the data structure.

The elements stored in a red-black tree all contain a distinguished key drawn from some totally ordered set, and the tree is ordered with respect to the keys. In some cases, it suffices to specify an element to be deleted, say, through its key. If several elements may have the same key, however, this approach is not usable, as it may not be immaterial which of the elements with a common key gets deleted. Even when keys are unique, specifying an element to be deleted from a Fibonacci heap through its key is not a good idea, as Fibonacci heaps

⋆ Supported in part by EPSRC grant GR L/92150.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 1–18, 2002.

do not support (efficient) searching: Locating the relevant key may involve the inspection of essentially all of the elements stored, an unacceptable overhead. In still other cases, such as with finger trees [5,17], even though reasonably efficient search is available, it may be possible to execute certain operations such as deletions faster if one already knows "where to start".

For these reasons, careful specifications of data-structure operations such as delete state that one of their arguments, rather than being the element to be operated on, is a pointer to that element. Several interpretations of the exact meaning of this are possible. In their influential textbook, Cormen et al. use the convention that elements are actually stored "outside of" the data structures, the latter storing only pointers to the elements [6, Part III, Introduction]. Thus insert and delete operations alike take a pointer to an element as their argument. In contrast, the convention employed by the LEDA library is that (copies of) elements are stored "inside" the data structures, and that an insertion of an element returns a pointer (a "dependent item", in LEDA parlance) to be used in future references to the element [19, Section 2.2].

The method employed by Cormen et al. can have its pitfalls. Consider, e.g., the code proposed for the procedure Binomial-Heap-Decrease-Key in [6, Section 20.2] and note that the "bubbling up" step repeatedly exchanges the *key* fields and all satellite fields of two nodes y and z, in effect causing y and z to refer to different elements afterwards. Because of this, repeated calls such as Binomial-Heap-Decrease-Key(H, x, k_1); Binomial-Heap-Decrease-Key (H, x, k_2), despite their appearance, may not access the same logical element. Indeed, as binomial heaps do not support efficient searching, it will not be easy for a user to get hold of valid arguments x to use in calls of the form Binomial-Heap-Decrease-Key(H, x, k). One might think that this problem could be solved by preserving the *key* and satellite fields of the nodes y and z and instead exchanging their positions within the relevant binomial tree. Because of the high degrees of some nodes in binomial trees, however, this would ruin the time bounds established for the Binomial-Heap-Decrease-Key operation.

Presumably in recognition of this problem, in the second edition of their book, Cormen et al. include a discussion of so-called *handles* [7, Section 6.5]. One kind of handle, which we might call an *inward handle*, is a pointer from an element of an application program to its representation in a data structure in which, conceptually, it is stored. The inward handle has a corresponding *outward handle* that points in the opposite direction. Whenever the representation of an element is moved within a data structure, its outward handle is used to locate and update the corresponding inward handle. While this solution is correct, it is not modular, in that the data structure must incorporate knowledge of the application program that uses it. It can be argued that this goes against current trends in software design.

A solution that does not suffer from this drawback can be obtained by employing the LEDA convention described above and introducing one more level of indirection: The pointer supplied by the user of a binomial heap would then point not into the heap itself, but to an auxiliary location that contains the

actual pointer into the heap and is updated appropriately whenever the element designated moves within the heap.

In the following, we adopt the LEDA convention and assume that every insertion of an element into a data structure returns a pointer to be used in future references to the element. The pointer can be interpreted as a positive integer, which we will call the *tag* of the element. At every insertion of an element, the data structure must therefore "invent" a suitable tag and hand it to the user, who may subsequently access the element an arbitrary number of times through its tag. Eventually the user may delete the element under consideration, thus in effect returning its tag to the data structure for possible reuse. During the time interval in which a user is allowed to access some element through a tag, we say that the tag is *in use*. When the tag is not in use, it is *free*. Fig. 1 contrasts the solution using handles with the one based on tags.

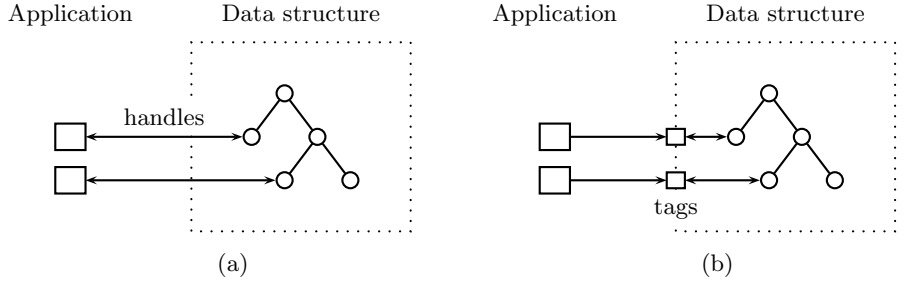


Fig. 1. The interplay between an application and a data structure based on (a) handles and (b) tags.

It might seem that the only constraint that a data structure must observe when "inventing" a tag is that the tag should not already be in use. This constraint can easily be satisfied by maintaining the set of free tags in a linked list: When a new tag is needed, the first tag on the free list is deleted from the list and put into use, and a returned tag is inserted at the front of the list. In general, however, a data structure has close to no control over which tags are in use. In particular, it may happen that at some point, it contains only few elements, but that these have very large tags. Recall that the purpose of a tag is to allow efficient access to an element stored in the data structure. The straightforward way to realize this is to store pointers to elements inside the data structure in a *tag array* indexed by tag values. Now, if some tags are very large (compared to the current number of elements), the tag array may have to be much bigger than the remaining parts of the data structure, putting any bounds established for its space consumption in jeopardy.

In his discussion of adding the delete operation to priority queues that lack it, Thorup [23, Section 2.3] proposes the use of tags (called "identifiers") and a tag array (called "D"). He recognizes the problem that identifiers may become too big for the current number of elements. The solution that he puts forward to

solve this problem places the burden on the user of a data structure: Whenever the number of elements stored has halved, the user is required to extract all elements from the data structure and subsequently reinsert them, which gives the data structure an opportunity to issue smaller tags. Our goal in this paper is to provide an algorithmic solution to the problem.

Even when no tag arrays are needed, small tags may be advantageous, simply because they can be represented in fewer bits and therefore stored more compactly. This is particularly crucial in the case of *packed* data structures, which represent several elements in a single computer word and operate on all of them at unit cost. In such a setting, tags must be stored with the elements that they designate and follow them as they move around within a data structure. Therefore it is essential that tags can be packed as tightly as the corresponding elements, which means that they should consist of essentially no more bits. The investigation reported here was motivated by an application to a linear-space packed priority queue.

We formalize the problem of maintaining the tags of a data structure by defining a *quasidictionary* to be a data structure that stores an initially empty set S of elements and supports the following operations:

check-in(v), where v is an element (an arbitrary constant-size collection of fields), inserts v in S and returns a positive integer called the tag of v and distinct from the tag of every other element of S.

check-out(x), where x is the tag of an element in S, removes that element from S.

access(x), where x is the tag of an element in S, inspects and/or modifies that element (but not its tag); the exact details are of no importance here.

The term "quasidictionary" is motivated by the similarity between the data structure defined above and the standard *dictionary* data structure. The latter supports the operations insert(x, v), delete(x), and access(x). The access operations of the two data structures are identical, and delete(x) has the same effect as check-out(x). The difference between insert(x, v) and check-in(v) is that in the case of insert(x, v), the tag x that gives access to v in later operations is supplied by the user, whereas in the case of check-in(v), it is chosen by the data structure. In many situations, the tags employed are meaningful to the user, and a dictionary is what is called for. In other cases, however, the elements to be stored have no natural a priori names, and a quasidictionary can be used and may be more convenient.

Quasidictionaries have potential applications to storage management. A storage allocator must process online a sequence of requests for contiguous blocks of memory of specified sizes, interspersed with calls to free blocks allocated earlier. It is known [21,22] that if N denotes the maximum total size of the blocks in use simultaneously, the storage allocator needs $\Omega(N \log N)$ words of memory to serve the requests in the worst case. As proved in [18], randomization does not help very much. A decision version of the corresponding offline problem, in which all requests are known initially, is NP-complete, even when all block sizes are 1 or 2 [12, Section A4.1], and the best polynomial-time approximation algorithm

known has an approximation ratio of 3 [13]. Thus the storage-allocation problem is difficult in theory. As argued in detail in [24], despite the use of a number of heuristics, it also continues to constitute a challenge in practice.

Conventional storage allocators are not allowed to make room for a new block by *relocating* other blocks while they are still in use. This is because of the difficulty or impossibility of finding and updating all pointers that might point into such blocks. Relocating storage allocators, however, can achieve a superior memory utilization in many situations. This is where quasidictionaries come into the picture: Equipped with a "front-end" quasidictionary, a relocating storage allocator can present the interface of a nonrelocating one. An application requesting a block of memory then receives a tag that identifies the block rather than a pointer to the start of the block, and subsequent accesses to the block are made by passing the tag of the block and an offset within the block to a privileged operation that accesses the quasidictionary. This has a number of disadvantages. First, the tag cannot be used as a "raw" pointer into memory, and address arithmetic involving the tag is excluded. This may not be a serious disadvantage. Indeed, the use of pointers and pointer arithmetic is a major source of difficult-to-find software errors and "security holes". Second, memory accesses are slowed down due to the extra level of indirection. However, in the interest of security and programmer productivity, considerable run-time checks (e.g., of array indices against array bounds) were already made an integral part of modern programming languages such as Java.

Our model of computation is the unit-cost *word RAM*, a natural and realistic model of computation described and discussed more fully in [15]. Just as a usual RAM, a word RAM has an infinite memory consisting of cells with addresses $0, 1, 2, \ldots$, not assumed to be initialized in any particular way. For a positive integer parameter w, called the *word length*, every memory cell stores an integer in the range $\{0, \ldots, 2^w - 1\}$, known as a *word*. We assume that the instructions in the so-called *multiplication instruction set* can be executed in constant time on w-bit operands. These include addition, subtraction, bitwise Boolean operations, left and right bit shifts by an arbitrary number of positions, and multiplication. The word length w is always assumed to be sufficiently large to allow the memory needed by the algorithms under consideration to be addressed; in the context of the present paper, this will mean that if a quasidictionary at some point must store n elements, then $w \geq \log_2 n + c$ for a suitable constant $c > 0$; in particular, we will have $n < 2^w$. An algorithm for the word RAM is called *weakly nonuniform* if it needs access to a constant number of precomputed constants that depend on w. If it needs only the single constant 1, it is called *uniform*. When nothing else is stated, uniformity is to be understood.

Since the space requirements of data structures are of central interest to the present paper, we provide a careful definition, which is somewhat more involved than the corresponding definition for language-recognition problems. Informally, the only noteworthy points are that the space consumption is related not to an input size, but to the number of elements currently stored, and that space is counted in units of w-bit words.

First, a *legal input* to a data structure D is a sequence of operations that D is required to execute correctly, starting from its initial state. We will say that D *uses space* s during the execution of a legal input σ_1 to D if, even if the contents of all memory cells with addresses $\geq s$ are altered arbitrarily at arbitrary times during the execution of σ_1, subsequently every operation sequence σ_2 such that $\sigma_1\sigma_2$ is a legal input to D is executed correctly, and if s is the smallest non-negative integer with this property. Assume now that D stores sets of elements and let $\mathbb{N} = \{1, 2, \dots\}$ and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. The *space consumption* of D is the pointwise smallest function $g : \mathbb{N}_0 \to \mathbb{N}_0$ such that for all legal inputs σ to D, if D contains n elements after the execution of σ, then the space used by D during the execution of σ is bounded by $g(n)$. If the space consumption of D is $O(n)$, we say that D works in linear space.

Our main result is the following:

Theorem 1. *There is a uniform deterministic quasidictionary that can be initialized in constant time, executes the operations* check-in *and* access *in constant time and* check-out *in constant amortized time, works in linear space, and uses only tags bounded by the maximum number of tags in use simultaneously since the quasidictionary was last empty.*

A result corresponding to Theorem 1 is not known for the more powerful dictionary data structure. Constant execution times in conjunction with a linear space consumption can be achieved, but only at the expense of introducing randomization [9]. A deterministic solution based on balanced binary trees executes every operation in $O(\log n)$ time, where n is the number of elements currently stored, whereas a data structure of Beame and Fich [4], as deamortized by Andersson and Thorup [3], yields execution times of $O(\sqrt{\log n / \log\log n})$. If we insist on constant-time access, the best known result performs insertions and deletions in $O(n^\epsilon)$ time, for arbitrary fixed $\epsilon > 0$ [16]. We are not aware of any previous work specifically on quasidictionaries.

2 Four Building Blocks

We will compose the quasidictionary postulated in Theorem 1 from four simpler data structures. One of these is a quasidictionary, except that it supports an additional operation scan that lists all tags currently in use, and that it is initialized with a set of pairs of (distinct) tags already in use and their corresponding elements, rather than starting out empty; we call such a data structure an *initialized quasidictionary*. The three other data structures are *static dictionaries*, which are also initialized with a set of (tag , element) pairs, and which subsequently support only the operation access. The four building blocks are characterized in the following four lemmas.

Lemma 1. *There is an initialized quasidictionary that, provided that every tag initially in use is bounded by N and that the number of tags simultaneously in use is guaranteed never to exceed N, can be initialized in $O(N)$ time and*

space, uses $O(N)$ space, and executes check-in, check-out *and* access *in constant time and* scan *in $O(N)$ time. Moreover, every tag used is bounded by a tag initially present or by the maximum number of tags in use simultaneously since the quasidictionary was initialized.*

Proof. Use a tag array storing for each of the integers $1, \ldots, N$ whether it is currently in use as a tag and, if so, its associated element. Moreover, maintain the set of free tags in $\{1, \ldots, N\}$ in a linked list as described in the introduction. By assumption, no tag will be requested when the list is empty, and it is easy to carry out the operations within the stated time bounds. Provided that the initial free list is sorted in ascending order, no check-in operation will issue a tag that is too large.

We will refer to the integer N that appears in the statement of the previous lemma as the *capacity* of the data structure. A data structure similar to that of the lemma, but without the need for an initial specification of a capacity, was implemented previously by Demaine [8] for use in a simulated multiprocessor environment.

Lemma 2. *There is a static dictionary that, when initialized with $n \geq 4$ tags in $\{1, \ldots, N\}$ and their associated elements, can be initialized in $O(n + N/\log N)$ time and space, uses $O(n+N/\log N)$ space, and supports constant-time accesses.*

Proof. Assume without loss of generality that the tag N is in use. The data structure begins by computing a quantity b as a power of two with $2 \leq b \leq \log N$, but $b = \Omega(\log N)$, as well as $\log b$, with the help of which divisions by b can be carried out in constant time. Similarly, it computes l as a power of two with $\log b \leq l \leq b$, but $l = O(\log b)$, as well as $\log l$.

Let S be the set of tags to be stored. For $j = 0, \ldots, r = \lfloor N/b \rfloor$, let $U_j = \{x \mid 1 \leq x \leq N \text{ and } \lfloor x/b \rfloor = j\}$ and take $S_j = S \cap U_j$. For $j = 0, \ldots, r$, the elements with tags in S_j are stored in a separate static dictionary D_j, the starting address of which is stored in position j of an array of size $r + 1$. For $j = 0, \ldots, r$, D_j consists of an array B_j of size $|S_j|$ that contains the elements with tags in S_j and a table L_j that maps every tag in S_j to the position in B_j of its associated element. L_j is organized in one of two ways.

If $|S_j| \geq l$, L_j is an array indexed by the elements of U_j. Since $|S_j| \leq b \leq 2^l$, every entry in L_j fits in a field of l bits. Moreover, by assumption, the word length w is at least $\log N \geq b$. We therefore store L_j in $l \leq |S_j|$ words, each of which contains b/l fields of l bits each. It is easy to see that accesses to L_j can be executed in constant time.

If $|S_j| < l$, L_j is a list of $|S_j|$ pairs, each of which consists of the remainder modulo b of a tag $x \in S_j$ (a *tag remainder*) and the corresponding position in B_j. Each pair can be represented in $2l$ bits, so the entire list L_j fits in $2l^2 = O((\log \log N)^2)$ bits. We precompute a table, shared between all of $D_0, \ldots, D_r$, that maps a list L_j with $|S_j| < l$ and a tag remainder to the corresponding position in B_j. The table occupies $o(r)$ words and is easily computed in $o(r)$ time. Again, accesses to L_j can be executed in constant time.

It is simple to verify that the overall dictionary uses $O(r+\sum_{j=0}^{r}(1+|S_j|)) = O(n+N/\log N)$ space, can be constructed in $O(n+N/\log N)$ time, and supports accesses in constant time.

Lemma 3. *There is a static dictionary that, when initialized with $n \geq 2$ tags in $\{1,\ldots,N\}$ and their associated elements, can be initialized in $O(n \log n)$ time and $O(n)$ space, uses $O(n)$ space, and supports accesses in $O(1+\log N/\log n)$ time.*

Proof. We give only the proof for $N = n^{O(1)}$, which is all that is needed in the following. The proof of the general case proceeds along the same lines and is hardly any more difficult.

The data structure begins by sorting the tags in use. Viewing each tag as a k-digit integer to base n, where $k = O(1+\log N/\log n) = O(1)$, and processing the tags one by one in sorted order, it is then easy, in $O(n)$ time, to construct a *trie* or *digital search tree* T for the set of tags. T is a rooted tree with exactly n leaves, all at depth exactly k, each edge of T is labeled with a digit, and each leaf u of T corresponds to a unique tag x in the sense that the labels on the path in T from the root to u are exactly the digits of x in the order from most to least significant. If the element associated with each tag is stored at the corresponding leaf of T, it is easy to execute access operations in constant time by searching in T, provided that the correct edge to follow out of each node in T can be found in constant time.

T contains fewer than n *branching nodes*, nodes with two or more children, and these can be numbered consecutively starting at 0. Since the total number of children of the branching nodes is bounded by $2n$, it can be seen that the problem of supporting searches in T in constant time per node traversed reduces to that of providing a static dictionary for at most $2n$ keys, each of which is a pair consisting of the number of a branching node and a digit to base n, i.e., is drawn from a universe of size bounded by n^2. We construct such a dictionary using the $O(n \log n)$-time procedure described in [16, Section 4].

Remark. The main result of [16] is a static dictionary for n *arbitrary* keys that can be constructed in $O(n \log n)$ time and supports constant-time accesses. The complete construction of that dictionary is weakly nonuniform, however, which is why we use only a uniform part of it.

Lemma 4. *There is a constant $\nu \in \mathbb{N}$ and functions $f_1,\ldots,f_\nu : \mathbb{N}_0 \to \mathbb{N}_0$, evaluable in linear time, such that given a set S of $n \geq 2$ tags bounded by N with associated elements as well as $f_1(b),\ldots,f_\nu(b)$ for some integer b with $\log N \leq b \leq w$, a static dictionary for S that uses $O(n)$ space and supports accesses in constant time can be constructed in $O(n^5)$ time using $O(n)$ space.*

Proof. For $\log N \leq n^3$, we use a construction of Raman [20, Section 4], which works in $O(n^2 \log N) = O(n^5)$ time.

For $\log N > n^3$, the result can be proved using the *fusion trees* of Fredman and Willard [11] (see [15, Corollary 8]). Reusing some of the ideas behind the fusion trees, we give a simpler construction.

Number the bit positions of a w-bit word from the right, starting at 0. Our proof takes the following route: First we show how to compute a set A of fewer than n bit positions such that two arbitrary distinct tags in S differ in at least one bit position in A. Subsequently we describe a constant-time transformation that, given an arbitrary word x, clears the bit positions of x outside of A and compacts the positions in A to the first n^3 bit positions. Noting that this maps S injectively to a universe of size 2^{n^3}, we can appeal to the first part of the proof.

Write $S = \{x_1, \ldots, x_n\}$ with $x_1 < \cdots < x_n$. We take

$$A = \{\mathrm{MSB}(x_i \oplus x_{i+1}) \mid 1 \le i < n\},$$

where $\oplus$ denotes bitwise exclusive-or and $\mathrm{MSB}(x) = \lfloor \log_2 x \rfloor$ is the *most significant bit* of x, i.e., the position of the leftmost bit of x with a value of 1. In other words, A is the set of most significant bit positions in which consecutive elements of S differ. To see that elements x_i and x_j of S with $x_i < x_j$ differ in at least one bit position in A even if $j \ne i + 1$, consider the digital search tree T for S described in the previous proof, but for base 2, and observe that $\mathrm{MSB}(x_i \oplus x_j) = \mathrm{MSB}(x_r \oplus x_{r+1})$, where x_r is the largest key in S whose corresponding leaf in T is in the left subtree of the lowest common ancestor of the leaves corresponding to x_i and x_j.

By assumption, $\oplus$ is a unit-time operation, and Fredman and Willard have shown how to compute $\mathrm{MSB}(x)$ from x in constant time [11, Section 5]. Their procedure is weakly nonuniform in our sense. Recall that this means that it needs to access a fixed number of constants that depend on the word length w. It is easy to verify from their description that the constants can be computed in $O(w)$ time (in fact, in $O(\log w)$ time). We define ν and $f_1, \ldots, f_\nu$ so that the required constants are precisely $f_1(b), \ldots, f_\nu(b)$. For this we must pretend that the word length w is equal to b, which simply amounts to clearing the bit positions $b, b+1, \ldots$ after each arithmetic operation. It follows that A can be computed in $O(n \log n)$ time.

Take $k = |A|$ and write $A = \{a_1, \ldots, a_k\}$. Multiplication with an integer $\mu = \sum_{i=1}^{k} 2^{m_i}$ can be viewed as copying each bit position a to positions $a + m_1, \ldots, a + m_k$. Following Fredman and Willard, we plan to choose μ so that no two copies of elements of A collide, while at least one *special copy* of each element of A is placed in one of the bit positions $b, \ldots, b + k^3 - 1$. The numbers $m_1, \ldots, m_k$ can be chosen greedily. For $i = 1, \ldots, k$, none of the k copies of a_i may be placed in one of the at most $(k-1)k$ positions already occupied, which leaves at least one position in the range $b, \ldots, b + k^3 - 1$ available for the special copy of a_i. Thus a suitable multiplier μ can be found in $O(k^4) = O(n^4)$ time.

An operation $\mathsf{access}(x)$ is processed as follows: First the bits of x outside of positions in A are cleared. Then the product μx is formed. It may not fit in one word. Since $b + k^3 \le w + n^3 \le w + \log N \le 2w$, however, it fits in two words and can be computed in constant time using simulated double-precision arithmetic. Shifting μx right by b bits and clearing positions $n^3, n^3 + 1, \ldots$ completes the transformation of x, and the transformed tag is accessed in a data structure constructed using Raman's method.

The static dictionaries of Lemmas 2, 3 and 4 can easily be extended to support deletions in constant time and scans in $O(n)$ time. For deletion, it suffices to mark each tag with an additional bit that indicates whether the tag is still in use. For scan, step through the list from which the data structure was initialized (which must be kept around for this purpose) and output precisely the tags not marked as deleted.

3 The Quasidictionary

In this section we describe a quasidictionary with the properties claimed in Theorem 1, except that check-in operations take constant time only in an amortized sense. After analyzing the data structure in the next section, we describe the minor changes needed to turn the amortized bound for check-in into a worst-case bound. Since we can assume that the data structure is re-initialized whenever it becomes empty, in the following we will simplify the language by assuming that the data structure was never empty in the past after the execution of the first (check-in) operation.

Observe first that for any constant $c \in \mathbb{N}$, the single semi-infinite memory at our disposal can emulate c separate semi-infinite memories with only a constant-factor increase in execution times and space consumption: For $i \in \mathbb{N}_0$, the real memory cell with address i is simply interpreted as the cell with address $\lfloor i/c \rfloor$ in the $((i \bmod c)+1)$st emulated memory. Then a space bound of $O(n)$ established for each emulated memory translates into an overall $O(n)$ space bound. In the light of this, we shall feel free to store each of the main components of the quasidictionary in its own semi-infinite array.

We divide the set of potential tags into *layers*: For $j \in \mathbb{N}$, the jth layer U_j consists of those natural numbers whose binary representations contain exactly j bits; i.e., $U_1 = \{1\}$, $U_2 = \{2,3\}$, $U_3 = \{4,\ldots,7\}$ and so on. In general, for $j \in \mathbb{N}$, $U_j = \{2^{j-1},\ldots,2^j-1\}$ and $|U_j| = 2^{j-1}$. Let S be the current set of tags in use and for $j \in \mathbb{N}$, take $S_j = S \cap U_j$. For $j \in \mathbb{N}$, we call $|S_j| + |S_{j-1}|$ the *pair size* of j (take $S_0 = \emptyset$), and we say that S_j is *full* if $S_j = U_j$.

Let $J = \{j \in \mathbb{N} \mid S_j \neq \emptyset\}$. For all $j \in J$ and all tags $x \in S_j$, the element with (external) tag x is stored with an (internal) tag of $x - (2^{j-1} - 1)$ in a separate data structure D_j called a *data stripe* and constructed according to one of Lemmas 1–4. The offset $o_j = 2^{j-1} - 1$ applied to tags in D_j is called for by our convention of letting all tag ranges start at 1. For $j \in J$ and $i = 1,\ldots,4$, if D_j is constructed as described in Lemma i, we will say that D_j is (stored) in *Representation* i. All data stripes are stored in an initial block of a semi-infinite array $\mathcal{D}$. The first cell of $\mathcal{D}$ following this block is pointed to by a variable *top*. When stating that a data stripe D_j is *moved to* $\mathcal{D}$, what we mean is that it is stored in $\mathcal{D}$, starting at the address *top*, after which *top* is incremented by the size of D_j. *Converting* a data stripe D_j *to* Representation i means extracting all (tag , element) pairs stored in D_j by means of its scan operation, abandoning the old D_j (which may still continue to occupy a segment of $\mathcal{D}$), initializing a new D_j in Representation i with the pairs extracted, and moving (the new) D_j

to $\mathcal{D}$. Every data stripe is *marked* or *unmarked*, the significance of which will be explained shortly.

Another main component of the quasidictionary is a dictionary H that maps each $j \in J$ to the starting address in $\mathcal{D}$ of (the current) D_j. All changes to $\mathcal{D}$ are reflected in H through appropriate calls of its operations; this will not be mentioned explicitly on every occasion. Following the first check-in operation, we also keep track of $b = \lfloor \log_2 N \rfloor + 1$, where N is the maximum number of tags simultaneously in use since the initialization. By assumption, we always have $b \leq w$, and b never decreases. The main components of the quasidictionary are illustrated in Fig. 2.

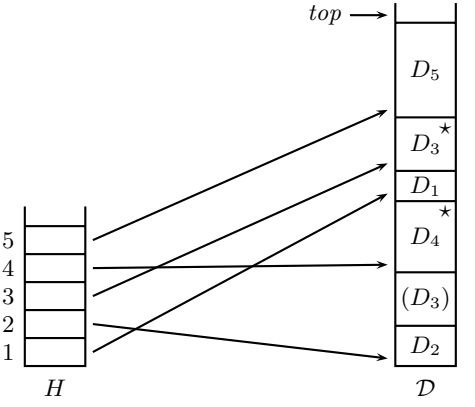


Fig. 2. The array $\mathcal{D}$ that holds the data stripes and a symbolic representation of the dictionary H that stores their starting addresses. A star denotes a marked data stripe, and an abandoned data stripe is shown with parentheses.

3.1 access

To execute access(x), we first determine the value of j such that $x \in U_j$, which is the same as computing $\mathrm{MSB}(x)+1$. Then the starting address of D_j is obtained from H, and the operation access$(x - o_j)$ is executed on D_j.

As explained in the proof of Lemma 4, $\mathrm{MSB}(x)$ can be computed from x in constant time, provided that certain quantities $f_1(b), \ldots, f_\nu(b)$ depending on b are available. We assume that $f_1(b), \ldots, f_\nu(b)$ are stored as part of the quasidictionary and recomputed at every change to b.

3.2 check-in

To execute check-in(v), we first compute the smallest positive integer j such that S_j is nonfull. For this purpose we maintain a bit vector q whose bit in position $b - i$ is 1 if and only if S_i is nonfull, for $i = 1, \ldots, b$, so that $j = b - \mathrm{MSB}(q)$.

Note that, by assumption, $|S|$ remains bounded by $2^b - 1$, which implies that q is always nonzero just prior to the execution of a check-in operation.

If $j \notin J$, we create a new, empty, and unmarked data stripe D_j with capacity 2^{j-1} in Representation 1, move it to $\mathcal{D}$ (recall that this means placing it at the end of the used block of $\mathcal{D}$, recording its starting address in H, and updating *top*), execute the operation check-in(v) on D_j, and return the tag obtained from D_j plus o_j to the caller.

If $j \in J$, we locate D_j as in the case of access. If D_j is not stored in Representation 1, we convert it to that representation with capacity 2^{j-1}, execute the operation check-in(v) on (the new) D_j, and return the tag obtained from D_j plus o_j to the caller.

3.3 check-out

To execute check-out(x), we first determine the value of j such that $x \in U_j$ as in the case of access(x). Operating on D_j, we then execute check-out$(x - o_j)$ if D_j is in Representation 1 and delete$(x - o_j)$ otherwise. For $i \in \{j, j+1\} \cap J$, if the pair size of i has dropped below 1/4 of its value when D_i was (re)constructed, we mark D_i. Finally we execute a *global cleanup* (see below) if the number of check-out operations executed since the last global cleanup (or, if no global cleanup has taken place, since the initialization) exceeds the number of tags currently in use.

3.4 Global Cleanup

A global cleanup clears $\mathcal{D}$ after copying it to temporary storage, sets $top = 0$, and then processes the data stripes one by one. If a data stripe D_j is empty (i.e., $S_j = \emptyset$), it is discarded. Otherwise, if D_j is not marked, it is simply moved back to $\mathcal{D}$. If D_j is marked, it is unmarked and moved to $\mathcal{D}$ after conversion to Representation i, where

$$i = \begin{cases} 2, & \text{if } 2^j/j \le |S_j|, \\ 3, & \text{if } 2^{j/5} \le |S_j| < 2^j/j, \\ 4, & \text{if } \quad |S_j| < 2^{j/5}. \end{cases}$$

3.5 The Dictionary H

The dictionary H that maps each $j \in J$ to the starting address h_j of D_j in $\mathcal{D}$ is implemented as follows: The addresses h_j, for $j \in J$, are stored in arbitrary order in the first $|J|$ locations of a semi-infinite array $\mathcal{H}$. When a new element j enters J, h_j is stored in the first free location of $\mathcal{H}$, and when an element j leaves J, h_j is swapped with the address stored in the last used location of $\mathcal{H}$, which is subsequently declared free. This standard device ensures that the space used in $\mathcal{H}$ remains bounded by $|S|$.

What remains is to implement a dictionary H' that maps each $j \in J$ to the position d_j of h_j in $\mathcal{H}$, a nonnegative integer bounded by $b - 1$. We do this in a

way reminiscent of Representation 2. Let l be a power of two with $\log b < l \leq b$, but $l = O(\log b)$. We assume that l and its logarithm are recomputed at every change to b.

When $|J|$ grows beyond $2l$, H' is converted to a representation as an array, indexed by $1, \ldots, b$ and stored in at most $2l \leq |J|$ words, each of which contains $\lfloor b/l \rfloor$ fields of l bits each.

When $|J|$ decreases below l, H' is converted to a representation in which the elements of J are stored in the rightmost $|J|$ fields of $l+1$ bits each in a single word X (or a constant number of words), and the corresponding positions in $\mathcal{H}$ are stored in the same order in the rightmost $|J|$ fields of $l+1$ bits in another word Y. Given $j \in J$, the corresponding position d_j can be computed in constant time using low-level subroutines that exploit the instruction set of the word RAM and are discussed more fully in [1]. First, multiplying j by the word 1_{l+1} with a 1 in every field yields a word with j stored in every field. Incrementing this word by 1_{l+1}, shifted left by l bits, changes to 1 the leftmost bit, called the *test bit*, of every field. Subtracting X from the word just computed leaves in the position of every test bit a 1 if and only if the value i of the corresponding field in X satisfies $i \leq j$. A similar computation carries out the test $i \geq j$, after which a bitwise AND singles out the unique field of X containing the value j. Subtracting from the word just created, which has exactly one 1, a copy of itself, but shifted right by l positions, we obtain a mask that can be used to isolate the field of Y containing d_j. This field still has to be brought to the right word boundary. This can be achieved through multiplication by 1_{l+1}, which copies d_j to the leftmost field, right shift to bring that field to the right word boundary, and removal of any spurious bits to the left of d_j. It can be seen that H' uses $O(|S|)$ space and can execute access, insert, and delete in constant time.

The description so far implicitly assumed b to be a constant. Whenever b increases, the representation of H and the auxiliary quantity 1_{l+1} are recomputed from scratch.

4 Analysis

The data structure described in the previous section can clearly be initialized in constant time. We will show that it works in linear space, that it executes access in constant time and check-in and check-out in constant amortized time, and that every tag is bounded by the maximum number of tags in use since the initialization.

4.1 Space Requirements

In this subsection we show the space taken up by every component of the quasidictionary to be linear in the number of tags currently in use. As concerns the dictionary H, this was already observed in the discussion of its implementation, so we can restrict attention to $\mathcal{D}$. The size of the used part of $\mathcal{D}$ is given by the value of *top*, so our goal is to show that $top = O(|S|)$ holds at all times.

Define an *epoch* to be the part of the execution from the initialization or from just after a global cleanup to just after the next global cleanup or, if there are no more global cleanups, to the end of the execution. Let $n = |S|$ and, for all $j \in \mathbb{N}$, denote $|S_j|$ by n_j and the pair size $|S_j| + |S_{j-1}|$ of j by p_j.

Let $j \geq 2$ and $i \in \{1, \ldots, 4\}$ and consider a (re)construction of D_j in Representation i at the time at which it happens. If $i = 1$, the size of D_j is $O(2^j)$ and $p_j \geq 2^{j-2}$. If $i = 2$, the size of D_j is $O(n_j + 2^{j-1}/(j-1))$, which is $O(n_j) = O(p_j)$ because Representation 2 is chosen only if $n_j \geq 2^j/j$. If $i \in \{3, 4\}$, the size of D_j is $O(n_j) = O(p_j)$. Thus, in all cases, when D_j is (re)constructed, its size is $O(p_j)$. Moreover, since D_j is marked as soon as p_j drops below $1/4$ of its value at the time of the (re)construction of D_j, the assertion that the size of D_j is $O(p_j)$ remains true as long as D_j is unmarked. This shows that at the beginning of every epoch, where every data stripe is unmarked, we have $top = O(\sum_{j=1}^{\infty} p_j) = O(n)$.

Consider a fixed epoch, let σ be the sequence of operations executed in the epoch, and let σ' be the sequence obtained from σ by removing all check-out operations. We will analyze the execution of σ' starting from the same situation as σ at the beginning of the epoch and use primes ($'$) to denote quantities that pertain to the execution of σ'. Note that n' and $p'_1, p'_2, \ldots$ never decrease.

During the execution of σ', the used part of $\mathcal{D}$ grows only by data stripes D_j in Representation 1, and there is at most one such data stripe for each value of j. Consider a particular point in time t during the execution of σ'. For each $j \geq 2$ for which a data stripe D_j was moved to $\mathcal{D}$ before time t in the epoch under consideration, the size of this newly added D_j is within a constant factor of the value of p'_j at the time of its creation and also, since p'_j is nondecreasing, at time t. At time t, therefore, the total size of all data stripes added to $\mathcal{D}$ during the execution of σ' is $O(\sum_{j=1}^{\infty} p'_j) = O(n')$. By the analysis above, the same is true of the data stripes present in $\mathcal{D}$ at the beginning of the epoch, so $top' = O(n')$ holds throughout.

Consider now a parallel execution of σ and σ', synchronized at the execution of every operation in σ'. It is easy to see that check-out operations never cause more space on $\mathcal{D}$ to be used, i.e., $top \leq top'$ holds at all times during the epoch. We may have $n < n'$. Since the epoch ends as soon as $n' - n > n$, however, we always have $n' \leq 2n + 2$. At all times, therefore, $top \leq top' = O(n') = O(n)$.

4.2 Execution Times

access operations clearly work in constant time and leave the data structure unchanged (except for changes to elements outside of their tags), so they will be ignored in the following. We will show that check-in and check-out operations take constant amortized time by attributing the cost of maintaining the quasidictionary to check-in and check-out operations in such a way that every operation is charged for only a constant amount of computation.

Whenever b increases, various quantities depending on b as well as the representation of H are recomputed. This incurs a cost of $O(b)$ that can be charged to $\Omega(b)$ check-in operations since the last increase of b or since the initialization.

When $|J|$ rises above $2l$ or falls below l, H is converted from one representation to another. The resulting cost of $O(l)$ can be charged to the $\Omega(l)$ check-in or check-out operations since the last conversion of H or since the initialization.

For $j \in \mathbb{N}$, define a *j-phase* as the part of the execution from just before a (re)construction of D_j in Representation 1 to just before the next such (re)construction or, if there is none, to the end of the execution. Consider a particular j-phase for some $j \geq 2$. The cost of the (re)construction of D_j in Representation 1 at the beginning of the phase is $O(2^j)$. Since the pair size of j at that point is bounded by 2^j and D_j is not reconstructed until the pair size of j has dropped below 1/4 of its initial value, Lemma 2 shows the total cost of all reconstructions of D_j in Representation 2 in the j-phase under consideration to be

$$O\Big(\sum_i \big(4^{-i} \cdot 2^j + 2^{j-1}/(j-1)\big)\Big),$$

where the sum extends over all $i \geq 1$ with $4^{-i} \cdot 2^j \geq 2^j/j$. The number of such i being $O(\log j)$, the sum evaluates to $O(2^j)$. Likewise, by Lemmas 3 and 4, the total cost of all reconstructions of D_j in Representations 3 and 4 are

$$O\Big(\sum_{i=0}^{\infty} \big(4^{-i} \cdot (2^j/j) \cdot \log(2^j/j)\big)\Big) = O(2^j) \quad \text{and}$$

$$O\Big(\sum_{i=0}^{\infty} \big(4^{-i} \cdot 2^{j/5}\big)^5\Big) = O(2^j),$$

respectively. Thus the total cost of all (re)constructions of D_j in a j-phase is $O(2^j)$. For the first j-phase, we can charge this cost to 2^{j-2} check-in operations that filled S_{j-1}. For every subsequent j-phase, the pair size of j dropped below 1/4 of its initial value during the previous phase, hence to at most $(1/4)(2^{j-1} + 2^{j-2}) = (3/4) \cdot 2^{j-2}$. Since S_{j-1} was full again at the beginning of the phase under consideration, we can charge the cost of $O(2^j)$ to the intervening at least $(1/4) \cdot 2^{j-2}$ check-in operations that refilled S_{j-1}. The arguments above excluded the case $j = 1$, but the cost incurred in (re)constructing D_1 once is $O(1)$ and can be charged to whatever operation is under execution.

At this point, the only execution cost not accounted for is that of copying unmarked data stripes back to $\mathcal{D}$ during a global cleanup. By the analysis of the previous subsection, this cost is $O(|S|)$, and it can be charged to the at least $|S|$ check-out operations executed since the last global cleanup.

4.3 Tag Sizes

Consider the point in time just after the quasidictionary issues a tag x and suppose that no tag of x or larger was issued earlier and that $x \in U_j$. Then D_j is in Representation 1 and just issued the tag $x - o_j$, and neither D_j nor an earlier data stripe with index j (an "earlier incarnation" of D_j) ever before issued a

tag of $x - o_j$ or larger. The last sentence of Lemma 1 can be seen to imply that D_j has $x - o_j$ tags in use. Moreover, by the implementation of check-in, all of $S_1, \ldots, S_{j-1}$ are full, so that they contribute an additional $1+2+\cdots+2^{j-2} = o_j$ tags in use. Thus the overall quasidictionary has x tags in use after issuing x. It is now easy to see that the quasidictionary never uses a tag larger than the maximum number of tags in use since its initialization.

As an aside, we note that the binary representation of every tag issued by the quasidictionary consists of exactly as many bits as that of the smallest free tag.

5 Deamortization

In this section we show how the constant amortized time bound for check-in operations can be turned into a constant worst-case time bound. Since the deamortization techniques used are standard, we refrain from giving all details.

In addition to computation that is covered by a constant time bound, a check-in operation, in our formulation to this point, may carry out one or more of the following:

- After an increase in b, an $O(b)$-time computation of various quantities depending on b;
- After an increase in $|J|$ beyond $2l$, an $O(l)$-time conversion of the dictionary H to its array representation;
- A (re)construction of a data stripe in Representation 1.

The computation triggered by an increase in b is easy to deamortize. For concreteness, consider the calculation of $f_1(b), \ldots, f_\nu(b)$. Following an increase in b, rather than calculating $f_1(b), \ldots, f_\nu(b)$, we anticipate the next increase in b and start calculating $f_1(b+1), \ldots, f_\nu(b+1)$. The calculation is interleaved with the other computation and carried out piecemeal, $O(1)$ time being devoted to it by each subsequent check-in operation. Since the entire calculation needs $O(b)$ time, while at least 2^b check-in operations are executed before the next increase in b, it is easy to ensure that $f_1(b+1), \ldots, f_\nu(b+1)$ are ready when they are first needed.

The conversion of H to its array representation is changed to begin as soon as $|J|$ exceeds l. While the conversion is underway, queries to H are answered using its current representation, but updates are executed both in the current representation and in the incomplete array representation, so that an interleaved computation that spends constant time per check-in operation on the conversion can deliver an up-to-date copy of H in the array representation when $|J|$ first exceeds $2l$—because of the simplicity of the array representation, this is easily seen to be possible. Provided that any incomplete array representation of H is abandoned if and when $|J|$ decreases below l, a linear space bound continues to hold.

For $j \geq 2$, the analysis in Section 4 already charged the cost of (re)constructing a data stripe D_j in Representation 1 to at least $\frac{1}{4} \cdot 2^{j-2}$ check-in operations that (re)filled S_{j-1}. We now let an interleaved construction actually take place as

part of the execution of these check-in operations. In other words, every check-in operation that issues a tag from a data stripe D_{j-1} that is at least 3/4 full also spends constant time on the (re)construction of D_j in Representation 1, in such a way as to let the new D_j be ready before S_{j-1} is full. As above, updates to D_j that occur after the (re)construction has begun (these can only be deletions) are additionally executed on the incomplete new D_j. Provided that global cleanups remove every partially constructed data stripe D_j for which $|S_{j-1}| < \frac{3}{4} \cdot 2^{j-2}$, it is easy to see that a linear space bound and a constant amortized time bound for check-out operations continue to hold.

6 Conclusions

We have described a deterministic quasidictionary that works in linear space and executes every operation in constant amortized time. However, our result still leaves something to be desired.

First, we would like to deamortize the data structure, turning the amortized time bound for deletion into a worst-case bound. Barring significant advances in the construction of static dictionaries, it appears that techniques quite different from ours would be needed to attain this goal. To see this, consider an operation sequence consisting of a large number of calls of check-in followed by as many calls of check-out. During the execution of the check-out operations, the need to respect a linear space bound presumably requires the repeated construction of a static dictionary for the remaining keys. The construction cannot be started much ahead of time because the set of keys to be accommodated is unknown at that point. Therefore, if it needs superlinear time, as do the best constructions currently known, some calls of check-out will take more than constant time.

Second, one might want to eliminate the use of multiplication from the operations. However, by a known lower bound for the depth of unbounded-fanin circuits realizing static dictionaries [2, Theorem C], this cannot be done without weakening our time or space bounds or introducing additional unit-time instructions.

Finally, our construction is not as practical as one might wish, mainly due to the fairly time-consuming (though constant-time) access algorithm. Here there is much room for improvement.

References

1. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, Sorting in linear time?, *J. Comput. System Sci.* **57** (1998), pp. 74–93.
2. A. Andersson, P. B. Miltersen, S. Riis, and M. Thorup, Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient, Proc., 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1996), pp. 441–450.
3. A. Andersson and M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, Proc., 32nd Annual ACM Symposium on Theory of Computing (STOC 2000), pp. 335–342.

4. P. Beame and F. E. Fich, Optimal bounds for the predecessor problem, Proc., 31st Annual ACM Symposium on Theory of Computing (STOC 1999), pp. 295–304.
5. M. R. Brown and R. E. Tarjan, A representation for linear lists with movable fingers, Proc., 10th Annual ACM Symposium on Theory of Computing (STOC 1978), pp. 19–29.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* (1st edition), The MIT Press, Cambridge, MA, 1990.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (2nd edition), The MIT Press, Cambridge, MA, 2001.
8. E. D. Demaine, A threads-only MPI implementation for the development of parallel programs, Proc., 11th International Symposium on High Performance Computing Systems (HPCS 1997), pp. 153–163.
9. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, *SIAM J. Comput.* **23** (1994), pp. 738–761.
10. M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization problems, *J. ACM* **34** (1987), pp. 596–615.
11. M. L. Fredman and D. E. Willard, Surpassing the information theoretic bound with fusion trees, *J. Comput. System Sci.* **47** (1993), pp. 424–436.
12. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.
13. J. Gergov, Algorithms for compile-time memory optimization, Proc., 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1999), pp. 907–908.
14. L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, Proc., 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1978), pp. 8–21.
15. T. Hagerup, Sorting and searching on the word RAM, Proc., 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998), Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 1373, pp. 366–398.
16. T. Hagerup, P. B. Miltersen, and R. Pagh, Deterministic dictionaries, *J. Algorithms* **41** (2001), pp. 69–85.
17. S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Inf.* **17** (1982), pp. 157–184.
18. M. G. Luby, J. Naor, and A. Orda, Tight bounds for dynamic storage allocation, Proc., 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1994), pp. 724–732.
19. K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing,* Cambridge University Press, 1999.
20. R. Raman, Priority queues: Small, monotone and trans-dichotomous, Proc., 4th Annual European Symposium on Algorithms (ESA 1996), Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 1136, pp. 121–137.
21. J. M. Robson, An estimate of the store size necessary for dynamic storage allocation, *J. ACM* **18** (1971), pp. 416–423.
22. J. M. Robson, Bounds for some functions concerning dynamic storage allocation, *J. ACM* **21** (1974), pp. 491–499.
23. M. Thorup, On RAM priority queues, *SIAM J. Comput.* **30** (2000), pp. 86–109.
24. P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, Dynamic storage allocation: A survey and critical review, Proc., International Workshop on Memory Management (IWMM 1995), Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 986, pp. 1–116.

Combining Pattern Discovery and Probabilistic Modeling in Data Mining

Heikki Mannila[1,2]

[1] HIIT Basic Research Unit, University of Helsinki, Department of Computer Science, PO Box 26, FIN-00014 University of Helsinki, Finland
Heikki.Mannila@cs.helsinki.fi, http://www.cs.helsinki.fi/u/mannila
[2] Laboratory of Computer and Information Science, Helsinki University of Technology, PO Box 5400, FIN-02015 HUT, Finland

Abstract. Data mining has in recent years emerged as an interesting area in the boundary between algorithms, probabilistic modeling, statistics, and databases. Data mining research has come from two different traditions. The global approach aims at modeling the joint distribution of the data, while the local approach aims at efficient discovery of frequent patterns from the data. Among the global modeling techniques, mixture models have emerged as a strong unifying theme, and methods exist for fitting such models on large data sets. For pattern discovery, the methods for finding frequently occurring positive conjunctions have been applied in various domains. An interesting open issue is how to combine the two approaches, e.g., by inferring joint distributions from pattern frequencies. Some promising results have been achieved using maximum entropy approaches. In the talk we describe some basic techniques in global and local approaches to data mining, and present a selection of open problems.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, p. 19, 2002.

Time and Space Efficient Multi-method Dispatching

Stephen Alstrup[1], Gerth Stølting Brodal[2*], Inge Li Gørtz[1], and Theis Rauhe[1]

[1] The IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, Denmark. {stephen,inge,theis}@it-c.dk
[2] BRICS (Basic Research in Computer Science), Center of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. gerth@brics.dk.

Abstract. The *dispatching problem* for object oriented languages is the problem of determining the most specialized method to invoke for calls at run-time. This can be a critical component of execution performance. A number of recent results, including [Muthukrishnan and Müller SODA'96, Ferragina and Muthukrishnan ESA'96, Alstrup et al. FOCS'98], have studied this problem and in particular provided various efficient data structures for the *mono-method* dispatching problem. A recent paper of Ferragina, Muthukrishnan and de Berg [STOC'99] addresses the *multi-method* dispatching problem.
Our main result is a linear space data structure for *binary* dispatching that supports dispatching in logarithmic time. Using the same query time as Ferragina et al., this result improves the space bound with a logarithmic factor.

1 Introduction

The *dispatching problem* for object oriented languages is the problem of determining the most specialized method to invoke for a method call. This specialization depends on the actual arguments of the method call at run-time and can be a critical component of execution performance in object oriented languages. Most of the commercial object oriented languages rely on dispatching of methods with only one argument, the so-called *mono-method* or *unary dispatching problem*. A number of papers, see e.g.,[10,15] (for an extensive list see [11]), have studied the unary dispatching problem, and Ferragina and Muthukrishnan [10] provide a linear space data structure that supports unary dispatching in log-logarithmic time. However, the techniques in these papers do not apply to the more general *multi-method dispatching problem* in which more than one method argument are used for the dispatching. Multi-method dispatching has been identified as a powerful feature in object oriented languages supporting multi-methods such

* Supported by the Carlsberg Foundation (contract number ANS-0257/20). Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 20–29, 2002.

as Cecil [3], CLOS [2], and Dylan [4]. Several recent results have attempted to deal with d-ary dispatching in practice (see [11] for an extensive list). Ferragina et al. [11] provided the first non-trivial data structures, and, quoting this paper, several experimental object oriented languages' "ultimately success and impact in practice depends, among other things, on whether multi-method dispatching can be supported efficiently".

Our result is a *linear space* data structure for the *binary dispatching* problem, i.e., multi-method dispatching for methods with at most two arguments. Our data structure uses *linear space* and supports dispatching in logarithmic time. Using the same query time as Ferragina et al. [11], this result improves the space bound with a logarithmic factor. Before we provide a precise formulation of our result, we will formalize the general d-ary dispatching problem.

Let T be a rooted tree with N nodes. The tree represents a class hierarchy with nodes representing the classes. T defines a partial order $\prec$ on the set of classes: $A \prec B \Leftrightarrow A$ is a descendant of B (not necessarily a proper descendant). Let $\mathcal{M}$ be the set of methods and let m denote the number of methods and M the number of distinct method names in $\mathcal{M}$. Each method takes a number of classes as arguments. A method invocation is a query of the form $s(A_1, \ldots, A_d)$ where s is the name of a method in $\mathcal{M}$ and $A_1, \ldots, A_d$ are class instances. A method $s(B_1, \ldots, B_d)$ is *applicable* for $s(A_1, \ldots, A_d)$ if and only if $A_i \prec B_i$ for all i. The *most specialized method* is the method $s(B_1, \ldots, B_d)$ such that for every other applicative method $s(C_1, \ldots, C_d)$ we have $B_i \prec C_i$ for all i. This might be ambiguous, i.e., we might have two applicative methods $s(B_1, \ldots, B_d)$ and $s(C_1, \ldots, C_d)$ where $B_i \neq C_i$, $B_j \neq C_j$, $B_i \prec C_i$, and $C_j \prec B_j$ for some indices $1 \leq i, j \leq d$. That is, neither method is more specific than the other. Multi-method dispatching is to find the most specialized applicable method in $\mathcal{M}$ if it exists. If it does not exist or in case of ambiguity, "no applicable method" resp. "ambiguity" is reported instead.

The *d-ary dispatching problem* is to construct a data structure that supports multi-method dispatching with methods having up to d arguments, where $\mathcal{M}$ is static but queries are online. The cases $d = 1$ and $d = 2$ are the *unary* and *binary dispatching* problems respectively. In this paper we focus on the binary dispatching problem which is of "particular interest" quoting Ferragina et al. [11].

The input is the tree T and the set of methods. We assume that the size of T is $O(m)$, where m is the number of methods. This is not a necessary restriction but due to lack of space we will not show how to remove it here.

Results. Our main result is a data structure for the binary dispatching problem using $O(m)$ space and query time $O(\log\, m)$ on a unit-cost RAM with word size logarithmic in N with $O(N + m\,(\mathrm{loglog}\, m)^2)$ time for preprocessing. By the use of a reduction to a geometric problem, Ferragina et al. [11], obtain similar time bounds within space $O(m \log m)$. Furthermore they show how the case $d = 2$ can be generalized for $d > 2$ at the cost of factor $\log^{d-2} m$ in the time and space bounds.

Our result is obtained by a very different approach in which we employ a dynamic to static transformation technique. To solve the binary dispatching problem we turn it into a unary dispatching problem — a variant of the marked ancestor problem as defined in [1], in which we maintain a dynamic set of methods. The unary problem is then solved persistently. We solve the persistent unary problem combining the technique by Dietz [5] to make a data structure fully persistent and the technique from [1] to solve the marked ancestor problem. The technique of using a persistent dynamic one-dimensional data structure to solve a static two-dimensional problem is a standard technique [17]. What is new in our technique is that we use the class hierarchy tree to denote the time (give the order on the versions) to get a fully persistent data structure. This gives a "branching" notion for time, which is the same as what one has in a fully persistent data structure where it is called the version tree. This technique is different from the plane sweep technique where a plane-sweep is used to give a partially persistent data structure. A top-down tour of the tree corresponds to a plane-sweep in the partially persistent data structures.

Related and Previous Work. For the unary dispatching problem the best known bound is $O(N+m)$ space and $O(\text{loglog}\, N)$ query time [15,10]. For the d-ary dispatching, $d \geq 2$, the result of Ferragina et al. [11] is a data structure using space $O(m\ (t\ \log\ m/\log\ t)^{d-1})$ and query time $O((\log\ m/\log\ t)^{d-1}\text{loglog}\, N)$, where t is a parameter $2 \leq t \leq m$. For the case $t = 2$ they are able to improve the query time to $O(\log^{d-1} m)$ using fractional cascading. They obtain their results by reducing the dispatching problem to a point-enclosure problem in d dimensions: Given a point q, check whether there is a smallest rectangle containing q. In the context of the geometric problem, Ferragina et al. also present applications to approximate dictionary matching.

In [9] Eppstein and Muthukrishnan look at a similar problem which they call *packet classification*. Here there is a database of m filters available for preprocessing. Each query is a packet P, and the goal is to *classify* it, that is, to determine the filter of highest priority that applies to P. This is essentially the same as the multiple dispatching problem. For $d = 2$ they give an algorithm using space $O(m^{1+o(1)})$ and query time $O(\text{loglog}\, m)$, or $O(m^{1+\varepsilon})$ and query time $O(1)$. They reduce the problem to a geometric problem, very similar to the one in [11]. To solve the problem they use a standard plane-sweep approach to turn the static two-dimensional rectangle query problem into a dynamic one-dimensional problem,which is solved persistently such that previous versions can be queried after the plane sweep has occurred.

2 Preliminaries

In this section we give some basic concepts which are used throughout the paper.

Definition 1 (*Trees*). *Let T be a rooted tree. The set of all nodes in T is denoted $V(T)$. The nodes on the unique path from a node v to the root are denoted*

$\pi(v)$, which includes v and the root. The nodes $\pi(v)$ are called the ancestors *of v. The* descendants *of a node v are all the nodes u for which $v \in \pi(u)$. If $v \neq u$ we say that u is a* proper *descendant of v. The distance dist(v,w) between two nodes in T is the number of edges on the unique path between v and w. In the rest of the paper all trees are rooted trees.*

Let C be a set of colors. A labeling $l(v)$ of a node $v \in V(T)$ is a subset of C, i.e., $l(v) \subseteq C$. A labeling $l : V(T) \to 2^C$ of a tree T is a set of labelings for the nodes in T.

Definition 2 (*Persistent data structures*). *The concept of persistent data structures was introduced by Driscoll et al. in [8]. A data structure is* partially persistent *if all previous versions remain available for queries but only the newest version can be modified. A data structure is* fully persistent *if it allows both queries and updates of previous versions. An update may operate only on a single version at a time, that is, combining two or more versions of the data structure to form a new one is not allowed. The versions of a fully persistent data structure form a tree called the* version tree*. Each node in the version tree represents the result of one update operation on a version of the data structure. A persistent update or query take as an extra argument the version of the data structure to which the query or update refers.*

Known results. Dietz [5] showed how to make any data structure fully persistent on a unit-cost RAM. A data structure with worst case query time $O(Q(n))$ and update time $O(F(n))$ making worst case $O(U(n))$ memory modifications can be made fully persistent using $O(Q(n) \text{ loglog } n)$ worst case time per query and $O(F(n) \text{ loglog } n)$ expected amortized time per update using $O(U(n) \text{ loglog } n)$ space.

Definition 3 (*Tree color problem*).

Let T be a rooted tree with n nodes, where we associate a set of colors with each node of T. The tree color problem *is to maintain a data structure with the following operations:*

color(v,c): add c to v's set of colors, i.e., $l(v) \leftarrow l(v) \cup \{c\}$,
uncolor(v,c): remove c from v's set of colors, i.e., $l(v) \leftarrow l(v) \setminus \{c\}$,
findfirstcolor(v,c): find the first ancestor of v with color c (this may be v itself).

The incremental *version of this problem does not support uncolor, the* decremental *problem does not support color, and the* fully dynamic *problem supports both update operations.*

Known results. In [1] it is showed how to solve the tree color problem on a RAM with logarithmic word size in expected update time $O(\text{loglog } n)$ for both *color* and *uncolor*, query time $O(\log n/\text{loglog } n)$, using linear space and preprocessing time. The expected update time is due to hashing. Thus the expectation can be removed at the cost of using more space. We need worst case time when we make the data structure persistent because data structures with

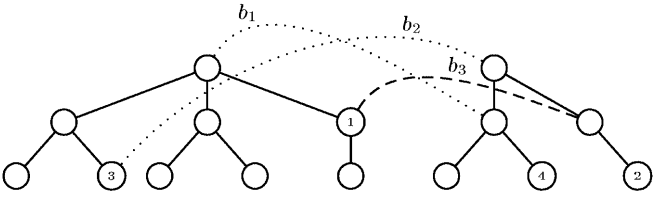


Fig. 1. The solid lines are tree edges and the dashed and dotted lines are bridges of color c and c', respectively. *firstcolorbridge*(c,v_1,v_2) returns b_3. *firstcolorbridge*(c',v_3,v_4) returns ambiguity since neither b_1 or b_2 is closer than the other.

amortized/expected time may perform poorly when made fully persistent, since expensive operations might be performed many times.

Dietz [5] showed how to solve the incremental tree color problem in $O(\text{loglog } n)$ amortized time per operation using linear space, when the nodes are colored top-down and each node has at most one color.

The unary dispatching problem is the same as the *tree color problem* if we let each color represent a method name.

Definition 4. *We need a data structure to support insert and predecessor queries on a set of integers from* $\{1, \ldots, n\}$. *This can be solved in worst case* $O(\text{loglog } n)$ *time per operation on a RAM using the data structure of van Emde Boas [18] (VEB). We show how to do modify this data structure such that it only uses* $O(1)$ *memory modifications per update.*

3 The Bridge Color Problem

The binary dispatching problem ($d = 2$) can be formulated as the following tree problem, which we call the *bridge color problem.*

Definition 5 (*Bridge Color Problem*). *Let* T_1 *and* T_2 *be two rooted trees. Between* T_1 *and* T_2 *there are a number of bridges of different colors. Let* C *be the set of colors. A bridge is a triple* $(c, v_1, v_2) \in C \times V(T_1) \times V(T_2)$ *and is denoted by* $c(v_1, v_2)$. *If* $v_1 \in \pi(u_1)$ *and* $v_2 \in \pi(u_2)$ *we say that* $c(v_1, v_2)$ *is a bridge* over (u_1, u_2). *The* bridge color problem *is to construct a data structure which supports the query firstcolorbridge(c,*v_1*,*v_2*). Formally, let* B *be the subset of bridges* $c(w_1, w_2)$ *of color* c *where* w_1 *is an ancestor of* v_1, *and* w_2 *an ancestor of* v_2. *If* $B = \emptyset$ *then firstcolorbridge(c,*v_1*,*v_2*) = NIL. Otherwise, let* $b_1 = c(w_1, w_1') \in B$, *such that dist(*v_1*,*w_1*) is minimal and* $b_2 = c(w_2', w_2) \in B$, *such that dist(*v_2, w_2*) is minimal. If* $b_1 = b_2$ *then firstcolorbridge(c,*v_1*,*v_2*)=* b_1 *and we say that* b_1 *is the* first bridge *over* (v_1, v_2), *otherwise firstcolorbridge(c,*v_1*,*v_2*) = "ambiguity". See Fig. 1.*

The binary dispatching problem can be reduced to the bridge color problem the following way. Let T_1 and T_2 be copies of the tree T in the binary dispatching

problem. For every method $s(v_1, v_2) \in \mathcal{M}$ make a bridge of color s between $v_1 \in V(T_1)$ and $v_2 \in V(T_2)$.

The problem is now to construct a data structure that supports *firstcolorbridge*. The object of the remaining of this paper is show the following theorem:

Theorem 1. *Using expected $O(m\ (\text{loglog}m)^2)$ time for preprocessing and $O(m)$ space, firstcolorbridge can be supported in worst case time $O(\log\ m)$ per operation, where m is the number of bridges.*

4 A Data Structure for the Bridge Color Problem

Let B be a set of bridges ($\mid B \mid = m$) for which we want to construct a data structure for the bridge color problem. As mentioned in the introduction we can assume that the number of nodes in the trees involved in the bridge color problem is $O(m)$, i.e., $\mid V(T_1) \mid + \mid V(T_2) \mid = O(m)$. In this section we present a data structure that supports *firstcolorbridge* in $O(\log\ m)$ time per query using $O(m)$ space for the bridge color problem.

For each node $v \in V(T_1)$ we define the labeling l_v of T_2 as follows. The labeling of a node $w \in V(T_2)$ contains color c if w is the endpoint of a bridge of color c with the other endpoint among ancestors of v. Formally, $c \in l_v(w)$ if and only if there exists a node $u \in \pi(v)$ such that $c(u, w) \in B$. Similar define the symmetric labelings for T_1. In addition to each labeling l_v, we need to keep the following extra information stored in a sparse array $H(v)$: For each pair $(w, c) \in V(T_2) \times C$, where $l_v(w)$ contains color c, we keep the node v' of maximal depth in $\pi(v)$ from which there is a bridge $c(v', w)$ in B. Note that this set is sparse, i.e., we can use a sparse array to store it.

For each labeling l_v of T_2, where $v \in V(T_1)$, we will construct a data structure for the static tree color problem. That is, a data structure that supports the query *findfirstcolor*(u,c) which returns the first ancestor of u with color c. Using this data structure we can find the first bridge over $(u, w) \in V(T_1) \times V(T_2)$ of color c by the following queries.

In the data structure for the labeling l_u of the tree T_2 we perform the query *findfirstcolor*(w,c). If this query reports NIL there is no bridge to report, and we can simply return NIL. Otherwise let w' be the reported node. We make a lookup in $H(u)$ to determine the bridge b such that $b = c(u', w') \in B$. By definition b is the bridge over (u, w') with minimal distance between w and w'. But it is possible that there is a bridge (u'', w'') over (u, w) where *dist*(u,u'') < *dist*(u,u'). By a symmetric computation with the data structure for the labeling $l(w)$ of T_1 we can detect this in which case we return "ambiguity". Otherwise we simply return the unique first bridge b.

Explicit representation of the tree color data structures for each of the labelings l_v for nodes v in T_1 and T_2 would take up space $O(m^2)$. Fortunately, the data structures overlap a lot: Let $v, w \in V(T_1)$, $u \in V(T_2)$, and let $v \in \pi(w)$. Then $l_v(u) \in l_w(u)$. We take advantage of this in a simple way. We make a fully persistent version of the *dynamic* tree color data structure using the technique of Dietz [5]. The idea is that the above set of $O(m)$ tree color data structures corresponds to a persistent, survived version, each created by one of $O(m)$ updates in total.

Formally, suppose we have generated the data structure for the labeling l_v, for v in T_1. Let w be the child of node v in T_1. We can then construct the data structure for the labeling l_w by simply updating the persistent structure for l_v by inserting the color marks corresponding to all bridges with endpoint w (including updating $H(v)$). Since the data structure is fully persistent, we can repeat this for each child of v, and hence obtain data structures for all the labelings for children of v. In other words, we can form all the data structures for the labeling l_v for nodes $v \in V(T_1)$, by updates in the persistent structures according to a top-down traversal of T_1. Another way to see this, is that T_1 is denoting the time (give the order of the versions). That is, the version tree has the same structure as T_1.

Similar we can construct the labelings for T_1 by a similar traversal of T_2. We conclude this discussion by the following lemma.

Lemma 1. *A static data structure for the bridge color problem can be constructed by $O(m)$ updates to a fully persistent version of the dynamic tree color problem.*

4.1 Reducing the Memory Modifications in the Tree Color Problem

The paper [1] gives the following upper bounds for the tree color problem for a tree of size m. Update time expected $O(\text{loglog}\, m)$ for both *color* and *uncolor*, and query time $O(\log\, m/\text{loglog}\, m)$, with linear space and preprocessing time.

For our purposes we need a slightly stronger result, i.e., updates that only make worst case $O(1)$ memory modifications. By inspection of the dynamic tree color algorithm, the bottle-neck in order to achieve this, is the use of the van Emde Boas predecessor data structure [18] (VEB). Using a standard technique by Dietz and Raman [6] to implement a fast predecessor structure we get the following result.

Theorem 2. *Insert and predecessor queries on a set of integers from $\{1, \ldots, n\}$ can be performed in $O(\text{loglog}\, n)$ worst case time per operation using worst case $O(1)$ memory modifications per update.*

To prove the theorem we first show an amortized result[1]. The elements in our predecessor data structure is grouped into buckets $S_1, \ldots, S_k$, where we maintain the following invariants:

$$\begin{array}{ll} (1)\ \max S_i < \min S_{i+1} & \text{for } i = 1, \ldots k-1, \text{ and} \\ (2)\ 1/2\ \log\ n < |S_i| \leq 2\ \log\ n & \text{for all } i. \end{array}$$

We have $k \in O(n/\ \log\ n)$. Each S_i is represented by a balanced search tree with $O(1)$ worst case update time once the position of the inserted or deleted element is known and query time $O(\log\, m)$, where m is the number of nodes in the tree [12,13]. This gives us update time $O(\text{loglog}\, n)$ in a bucket, but only $O(1)$ memory modifications per update. The minimum element s_i of each bucket S_i is stored in a VEB.

[1] The amortized result (Lemma 2) was already shown in [14], bur in order to make the deamortization we give another implementation here.

When a new element x is inserted it is placed in the bucket S_i such that $s_i < x < s_{i+1}$, or in S_1 if no such bucket exists. Finding the correct bucket is done by a predecessor query in the VEB. This takes $O(\text{loglog}\, n)$ time. Inserting the element in the bucket also takes $O(\text{loglog}\, n)$ time, but only $O(1)$ memory modifications. When a bucket S_i becomes to large it is split into two buckets of half size. This causes a new element to be inserted into the VEB and the binary trees for the two new buckets have to be build. An insertion into the VEB takes $O(\text{loglog}\, n)$ time and uses the same number of memory modifications. Building the binary search trees uses $O(\log\, n)$ time and the same number of memory modifications. When a bucket is split there must have been at least $\log\, n$ insertions into this bucket since it last was involved in a split. That is, splitting and inserting uses $O(1)$ amortized memory modifications per insertion.

Lemma 2. *Insert and predecessor queries on a set of integers from* $\{1, \ldots, n\}$ *can be performed in* $O(\text{loglog}\, n)$ *worst case time for predecessor and* $O(\text{loglog}\, n)$ *amortized time for insert using* $O(1)$ *amortized number of memory modifications per update.*

We can remove this amortization at the cost of making the bucket sizes $\Theta(\log^2 n)$ by the following technique by Raman [16] called thinning.

Let $\alpha > 0$ be a sufficiently small constant. Define the *criticality* of a bucket to be: $\rho(b) = \frac{1}{\alpha\ \log\ n}\max\{0, \text{size}(b) - 1.8\ \log^2\ n\}$. A bucket b is called *critical* if $\rho(b) > 0$. We want to ensure that $\text{size}(b) \leq 2\ \log^2\ n$. To maintain the size of the buckets every $\alpha\ \log\ n$ updates take the most critical bucket (if there is any) and move $\log\ n$ elements to a newly created empty adjacent bucket. A bucket rebalancing uses $O(\log\ n)$ memory modifications and we can thus perform it with $O(1)$ memory modifications per update spread over no more than $\alpha\ \log\ n$ updates.

We now show that the buckets never get too big. The criticality of all buckets can only increase by 1 between bucket rebalancings. We see that the criticality of the bucket being rebalanced is decreased, and no other bucket has its criticality increased by the rebalancing operations. We make use of the following lemma due to Raman:

Lemma 3 (*Raman*). *Let* $x_1, \ldots, x_n$ *be real-valued variables, all initially zero. Repeatedly do the following:*

(1) Choose n *non-negative real numbers* $a_1, \ldots, a_n$ *such that* $\sum_{i=1}^{n} a_i = 1$*, and set* $x_i \leftarrow x_i + a_i$ *for* $1 \leq i \leq n$.
(2) Choose an x_i *such that* $x_i = max_j\{x_j\}$*, and set* $x_i \leftarrow max\{x_i - c, 0\}$ *for some constant* $c \geq 1$.

Then each x_i *will always be less than* $\ln\, n + 1$*, even when* $c = 1$.

Apply the lemma as follows: Let the variables of Lemma 3 be the criticalities of the buckets. The reals a_i are the increases in the criticalities between rebalancings and $c = 1/\alpha$. We see that if $\alpha \leq 1$ the criticality of a bucket will never exceed $\ln + 1 = O(\log\ n)$. Thus for sufficiently small α the size of the buckets will never exceed $2\ \log^2\ n$. This completes the proof of Theorem 2.

We need worst case update time for *color* in the tree color problem in order to make it persistent. The expected update time is due to hashing. The expectation

can be removed at the cost of using more space. We now use Theorem 2 to get the following lemma.

Lemma 4. *Using linear time for preprocessing, we can maintain a tree with complexity* $O(\text{loglog } n)$ *for color and complexity* $O(\log\ n/\text{loglog } n)$ *for findfirstcolor, using* $O(1)$ *memory modifications per update, where* n *is the number of nodes in the tree.*

4.2 Reducing the Space

Using Dietz' method [5] to make a data structure fully persistent on the data structure from Lemma 4, we can construct a fully persistent version of the tree color data structure with complexity $O((\text{loglog } m)^2)$ for *color* and *uncolor*, and complexity $O((\log\ m/\text{loglog } m) \cdot \text{loglog } m) = O(\log\ m)$ for *findfirstcolor*, using $O(m)$ memory modifications, where m is the number of nodes in the tree.

According to Lemma 1 a data structure for the first bridge problem can be constructed by $O(m)$ updates to a fully persistent version of the dynamic tree color problem. We can thus construct a data structure for the bridge color problem in time $O(m\ (\text{loglog } m)^2)$, which has query time $O(\log\ m)$, where m is the number of bridges.

This data structure might use $O(c{\cdot}m)$ space, where c is the number of method names. We can reduce this space usage using the following lemma.

Lemma 5. *If there exists an algorithm* A *constructing a* static *data structure* D *using expected* $t(n)$ *time for preprocessing and expected* $m(n)$ *memory modifications and has query time* $q(n)$*, then there exists an algorithm constructing a data structure* D' *with query time* $O(q(n))$*, using expected* $O(t(n))$ *time for preprocessing and space* $O(m(n))$*.*

Proof. The data structure D' can be constructed the same way as D using dynamic perfect hashing [7] to reduce the space. □

Since we only use $O(m)$ memory modifications to construct the data structure for the bridge color problem, we can construct a data structure with the same query time using only $O(m)$ space. This completes the proof of Theorem 1.

If we use $O(N)$ time to reduce the class hierarchy tree to size $O(m)$ as mentioned in the introduction, we get the following corollary to Theorem 1.

Corollary 1. *Using* $O(N + m\ (\text{loglog } m)^2)$ *time for preprocessing and* $O(m)$ *space, the* multiple dispatching problem *can be solved in worst case time* $O(\log\ m)$ *per query. Here* N *is the number of classes and* m *is the number of methods.*

References

1. S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 534–543, 1998.
2. D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common LISP object system specification X3J13 document 88-002R. *ACM SIGPLAN Notices*, 23, 1988. Special Issue, September 1988.

3. Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.
4. Inc. Apple Computer. Dylan interim reference manual, 1994.
5. P. F. Dietz. Fully persistent arrays. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74, Berlin, August 1989. Springer-Verlag.
6. Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.
7. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 524–531. IEEE Computer Society Press, 1988.
8. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Computer Systems Sci.*, 38(1):86–124, 1989.
9. David Eppstein and S. Muthukrishnan. Internet packet filter manegement and rectangle geometry. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.
10. P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In *European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 107–120, 1996.
11. P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching problems. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 483–491, New York, May 1–4 1999. ACM Press.
12. R. Fleischer. A simple balanced search tree with O(1) worst-case update time. *International Journal of Foundations of Computer Science*, 7:137–149, 1996.
13. C. Levcopoulos and M. Overmars. A balanced search tree with O(1) worstcase update time. *Acta Informatica*, 26:269–277, 1988.
14. K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log \log n)$ time and $O(n)$ space. *Information Processing Letters*, 35:183–189, 1990.
15. S. Muthukrishnan and Martin Müller. Time and space efficient method-lookup for object-oriented programs (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–51, Atlanta, Georgia, January 28–30 1996.
16. R. Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time.* PhD thesis, University of Rochester, Computer Science Department, October 1992. Technical Report TR439.
17. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.
18. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1978.

Linear Time Approximation Schemes for Vehicle Scheduling

John E. Augustine[1] and Steven S. Seiden[2⋆]

[1] Dept. of Electrical & Computer Eng., Louisiana State University, Baton Rouge, LA 70803, USA,
augustine@ieee.org

[2] Department of Computer Science, 298 Coates Hall, Louisiana State University, Baton Rouge, LA 70803, USA,
sseiden@acm.org,

Abstract. We consider makespan minimization for vehicle scheduling problems on trees with release and handling times. 2-approximation algorithms were known for several variants of the single vehicle problem on a path [16]. A 3/2-approximation algorithm was known for the single vehicle problem on a path where there is a fixed starting point and the vehicle must return to the starting point upon completion [13]. Karuno, Nagamochi and Ibaraki give a 2-approximation algorithm for the single vehicle problem on trees. We develop a linear time PTAS for the single vehicle scheduling problem on trees which have a constant number of leaves. This PTAS can be easily adapted to accommodate various starting/ending constraints. We then extended this to a PTAS for the multiple vehicle problem where vehicles operate in disjoint subtrees. For this problem, the only previous result is a 2-approximation algorithm for paths [10]. Finally, we present competitive online algorithms for some single vehicle scheduling problems.

1 Introduction

In this paper we study the multiple vehicle scheduling problem (MVSP), which involves scheduling a set of vehicles to handle jobs at different sites. There are a large number of applications for such problems, for instance, scheduling automated guided vehicles [10], scheduling delivery ships on a shoreline [16], scheduling flexible manufacturing systems [10], etc.... MVSP is also equivalent to certain machine scheduling problems where there are costs for reconfiguring machines to perform different operations [2], and to power consumption minimization in CPU instruction scheduling [3].

Problem Description: MVSP is a variation of the well-known traveling salesman problem. In the most general formulation, the problem consists of a metric space $\mathcal{M}$ along with n jobs. Each job j becomes available for processing

⋆ Contact author. This research was partially supported by the Research Competitiveness Subprogram of the Louisiana Board of Regents.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 30–39, 2002.

at a time $r_j \geq 0$ known as its *release time*. Job j requires a specific amount of time $h_j \geq 0$ for its completion known as its *handling time*. Job handling is non-preemptive in nature meaning that if the vehicle starts processing a job, it is required to complete the processing. Finally, each job has a *position* p_j in $\mathcal{M}$. We are given a set of k vehicles that can traverse $\mathcal{M}$ and handle or *serve* these jobs. Our goal is to minimize the maximum completion time over all jobs, called the *makespan*, using the given set of vehicles.

Note that without loss of generality $\mathcal{M}$ is finite. $\mathcal{M}$ can be represented by a weighted graph, where the points are vertices, and the distance from $p \in \mathcal{M}$ to $q \in \mathcal{M}$ is the length of the shortest path from p to q in the graph. In an abuse of notation, we also use $\mathcal{M}$ to denote this graph. We are interested in the case where $\mathcal{M}$ is a tree; unless stated otherwise, all the discussion to follow pertains to this case. We use m and t to denote the number of vertices and leaves in $\mathcal{M}$, respectively. Note that a particularly interesting special case is $t = 2$, where $\mathcal{M}$ is a path.

Several different variants of this problem are possible:

- The single vehicle scheduling problem (SVSP) is just the special case $k = 1$.
- In the zone multiple vehicle scheduling problem (ZMVSP), the vehicles all operate in disjoint subtrees of G called *zones*. Part of the problem is to specify the zones.
- There are a large number of possibilities for vehicle starting/ending constraints. It is possible that the starting positions of the vehicles are given as part of the problem, or that they can be chosen by the algorithm. We call a problem specified starting point for a vehicle the *origin* of that vehicle. There are analogous possible constraints on vehicle ending positions. The most common variant when an ending position is specified is that the origin and the ending position are the same. We denote this variant as RTO (return to origin). When no ending position is specified, the most common variant is that each vehicle has a fixed origin. We denote this variant as FO (fixed origin).
- In the *online* variants of these problems, jobs are unknown before their release times, and even the number of jobs is a priori unspecified.

Since even SVSP on a path is NP-hard [7,17], we shift our focus from finding an optimal solution to finding an approximate solution with cost that is guaranteed to be within a certain bound relative to the optimal cost. Suppose we have an algorithm $\mathcal{A}$ for problem $\mathcal{P}$. We define $\text{cost}_{\mathcal{A}}(\sigma)$ to be the cost of the solution produced by $\mathcal{A}$ on instance σ of $\mathcal{P}$. Let $\text{cost}(\sigma)$ be minimum possible cost for σ. A *polynomial time ρ-approximation algorithm* $\mathcal{A}$ guarantees that for every instance σ of $\mathcal{P}$, $\text{cost}_{\mathcal{A}}(\sigma) \leq \rho\,\text{cost}(\sigma)$ and that $\mathcal{A}$ runs in polynomial time in $|\sigma|$. A *polynomial time approximation scheme* (PTAS) for problem $\mathcal{P}$ is a family of approximation algorithms $\{\mathcal{A}_\epsilon\}_{\epsilon \geq 0}$ such that each is a polynomial time $(1+\epsilon)$-approximation algorithm for $\mathcal{P}$. A *fully polynomial time approximation scheme* (PTAS) is a PTAS whose running time is polynomial in both $|\sigma|$ and $1/\epsilon$. The reader is referred to [8] for a more comprehensive treatment of approximation algorithms.

Previous Results: The traveling salesman problem, which is a precursor and special case of SVSP, is known to be NP-complete [9]. Further, it has been shown that a polynomial time ρ-approximation algorithm is only possible for $\rho \geq 203/202$, unless P=NP [6]. Because of these negative results, many people have attempted to solve special cases of this problem. Researchers have often exploited the different network topologies on which the TSP or more generally MVSP are formulated. Papadimitriou [15] shows that the TSP is NP-complete even in the Euclidean plane. For the general TSP, a 3/2-approximation is known, due to Christofides [4]. Approximation algorithms are known for several special cases [8].

Psaraftis *et al.* [16] consider SVSP on a path when all handling times are zero. They show that the RTO version can be solved exactly in $O(n)$ time, while the FO version can be solved exactly in $O(n^2)$ time. Psaraftis *et al.* further give 2-approximation algorithms for both these versions of SVSP with positive handling times. Tsitsiklis [17] shows that the FO and RTO versions of SVSP on paths with release and handling times are NP-complete. MVSP is NP-complete for all $k \geq 2$ even if all release times are zero and there is only a single point in $\mathcal{M}$, since this is exactly the multiprocessor scheduling problem [7]. If k is part of the input, then MVSP is strongly NP-complete. For paths, Karuno *et al.* develop a 3/2-approximation algorithm for the RTO version of SVSP [13] and a 2-approximation for the version of MVSP where the origins and ending points are not pre-specified [10]. For SVSP on trees, Karuno, Nagamochi and Ibaraki [12] develop a 2-approximation algorithm. For SVSP on trees with zero handling times, Nagamochi, Mochizuki and Ibaraki [14] give an exact algorithm that runs in time $O(n^t)$ and show strong NP-hardness. For SVSP on general metrics, they give an exact algorithm which runs in time $O(n^2 2^n)$, and a 5/2-approximation algorithm.

There has also been interest in the online version of SVSP. Ausiello *et al.* [1] investigate the online RTO and FO versions of the problem, where all handling times are zero. For a general class of metrics spaces, they show 2.5 and 2-competitive algorithms for the FO and RTO variants, respectively. For the FO version, they give a 7/3-competitive online algorithm, and a lower bound of 2 on the competitive ratio of any online algorithm. For the RTO variant, their upper and lower bounds are 7/4 and $(9 + \sqrt{17})/8 > 1.64038$, respectively.

There is a large body of work on vehicle scheduling problems with different job requirements, metric spaces and objective functions. For instance, Tsitsiklis [17] considers job deadlines, while Charikar *et al.* [3] consider precedence constraints. We do not give a comprehensive treatment of all variations here, but refer the reader to the survey of Desrosiers *et al.* [5].

Our Results: In this paper, we develop a PTAS that can be applied to many of the variants of SVSP. We begin in Section 2 by giving an exact algorithm for solving the FO variant of SVSP on a tree when the number of distinct release times is at most R. This algorithm runs in time $O(R(m+1)^{(R-1)(t+1)+1}n)$. In Section 3, we use this result to provide an $O(f(1/\epsilon, t)n)$ time $(1+\epsilon)$-approximation algorithm for the FO variant of SVSP, where $f(1/\epsilon, t)$ is a function exponential

in both t and $1/\epsilon$. This is accomplished by running the algorithm of Section 2 on a modified problem on a modified metric space. In this modified problem, R and m are constants depending only on ϵ. Our PTAS is easily adapted to all the other starting/finishing constraints previously mentioned, as well as others. In Section 4, we extend our algorithm to include ZMVSP using a dynamic programming approach. Essentially, this multiplies the running time by a factor of $O(n^t)$. In Section 5, we show that an extension of SVSP to include deadlines is NP-hard, even when all release times are zero. Finally, in Section 5, we show how to adapt the algorithms of Ausiello *et al.* [1] to get competitive online algorithms for some SVSP variants.

Note: Recently and independently, Karuno and Nagamochi [11] have also developed PTAS's for the vehicle scheduling problems described here. Their approach is different than ours. They develop an exact pseudopolynomial time algorithm for MVSP. The running time of this algorithm is exponential in k and polynomial in $\sum_j h_j$. We get a linear time PTAS for SVSP where they do not. Our PTAS for ZMSVP runs in time polynomial in k, whereas all their algorithms have running times exponential in k.

2 A Special Case

In this section, we consider the single vehicle scheduling problem on trees when there are a constant number R of distinct release times. We show that this problem can be solved exactly in time $O(R(m+1)^{(R-1)(t+1)+1}n)$. We assume the FO variant, but the algorithm given here can easily be adapted to handle all of the different starting and ending conditions described in the introduction.

We denote the origin by p_0. We assume that in the input, $\mathcal{M}$ is in adjacency list form. We use $d(x,y)$ to mean the distance from point x to point y in $\mathcal{M}$. We use $x \rightsquigarrow y$ to denote the set of vertices on the shortest path from x to y, including x and y. It is easy to see that vertices of degrees one and two containing no request can be eliminated from $\mathcal{M}$. Therefore we have $m \leq n + t - 2$.

A *schedule* for the single vehicle problem is just a permutation π on $\{1, \ldots, n\}$. In an abuse of notation, we define $\pi(0) = 0$. The *arrival time* $a^\sigma_\pi(i)$ and *completion time* $c^\sigma_\pi(i)$ of the vehicle at the ith request in π are defined

$$\begin{aligned} a^\sigma_\pi(i) &= c^\sigma_\pi(i-1) + d(p_{\pi(i-1)}, p_{\pi(i)}), \\ c^\sigma_\pi(0) &= 0, \\ c^\sigma_\pi(i) &= \max\{r_{\pi(i)}, a^\sigma_\pi(i)\} + h_{\pi(i)}. \end{aligned}$$

If the problem instance is clear from the context, we drop the σ superscript. The cost of π is $c^\sigma_\pi(n)$.

We say that schedule π *eagerly* serves request ℓ if for all i such that $p_\ell \in p_{\pi(i-1)} \rightsquigarrow p_{\pi(i)}$ either $\pi(\ell) \leq i$ or $r_\ell > c_\pi(i-1) + d(p_{\pi(i-1)}, p_\ell)$. If π eagerly serves all requests, we say that π is *eager*. Intuitively, an eager schedule never passes through the location of an available request without serving the request.

Lemma 1. *For any finite metric $\mathcal{M}$, if there is a schedule π for a single vehicle scheduling problem σ with cost x then there is also an eager schedule ϖ for σ with cost at most x.*

Proof. Consider some schedule π. Define

$$e(i,\ell) = c_\pi(i-1) + d(p_{\pi(i-1)},\, p_\ell),$$
$$f_\ell = \min\{i \mid r_\ell \le e(i,\ell), p_\ell \in p_{\pi(i-1)} \rightsquigarrow p_{\pi(i)}\}.$$

Intuitively, $e(i,\ell)$ is the earliest point in time that position p_ℓ can be reached after servicing requests $\pi(1),\ldots,\pi(i-1)$. The vehicle crosses request ℓ for the first time after it becomes available when traveling from request $\pi(f_\ell-1)$ to request $\pi(f_\ell)$. f_ℓ is well defined since $p_\ell \in p_{\pi(i-1)} \rightsquigarrow p_{\pi(i)}$ and $r_\ell \le e(i,\ell)$ for $i = \pi^{-1}(\ell)$. If $f_\ell = \pi^{-1}(\ell)$ for all ℓ, then π is eager. Otherwise, there is some request ℓ with $f_\ell < \pi^{-1}(\ell)$. Among these requests, let L be the one which minimizes $e(f_\ell,\ell)$. L is the first request crossed by π which is not eagerly served. Define $q = \pi^{-1}(L)$. Basically, we modify π to get π' by removing L from its current position in the order defined by π and inserting it between requests $\pi(f_L-1)$ and $\pi(f_L)$. This causes the service of requests $\pi(f_L),\ldots,\pi(q-1)$ to be delayed by at most h_L. However, in the modified schedule, we go directly from request $\pi(q-1)$ to $\pi(q+1)$, and so we arrive at $\pi(q+1)$ at least as early as before. Thus we see that $a_{\pi'}(q+1) \le a_\pi(q+1)$ (we show this formally in the full version). Using this fact, it is easy to show by induction that $c_{\pi'}(i) \le c_\pi(i)$ for $q < i \le n$. Therefore, the cost of π' is at most the cost of π. We have increased the number of eagerly served requests by one. By iterating this process, we eventually reach an eager schedule ϖ. □

We use $0 \le u_1 < \cdots < u_R$ to denote the possible release times. Define $u_0 = 0$ and $u_{R+1} = \infty$. Define *phase* i to be the time interval $[u_i,\, u_{i+1})$ for $0 \le i \le R$.

We show that it is possible to construct the optimal schedule in polynomial time. Let π be an optimal schedule. Without loss of generality, π is eager. For the remainder of the paragraph, let i be in $\{1,\ldots,R\}$. Let X_i be the set of requests whose service is initiated during phase i. If X_i is non-empty, define T_i to be the minimal subtree of $\mathcal{M}$ which contains all the requests in X_i. Define L_i to be the set of leaves of T_i. Note that $|L_i| \le t$ since T_i is subtree of $\mathcal{M}$, and $\mathcal{M}$ has at most t leaves. Let X_i^0 be the position of the first request served during phase i in schedule π. For $1 \le j \le |L_i|$, let X_i^j be the jth leaf visited by the vehicle during phase i in schedule π. For $|L_i| < j \le t$, define $X_i^j = X_i^{|L_i|}$. If X_i is empty then we define $X_i^j = -1$ for $0 \le j \le t$. Define $X_0^t = p_0$.

We claim that the structure of π is completely defined by X_i^j for $1 \le j \le t$, $0 \le i \le R$. This follows from the fact that since π is eager, and all requests released during phase i are released at the beginning of the phase. X_i consists of exactly those requests which lie in T_i and which are released at or before time u_i. Define a *sweep* to be a time period during which the vehicle travels along some path, possibly stopping to serve requests, but without changing direction. Essentially, $t+1$ sweeps per phase are sufficient. If we sweep from X_{i-1}^t to X_i^0,

then sweep from X_i^0 to X_i^1, sweep from X_i^1 to X_i^2 etc..., we pass through all requests in X_i. We take this route and service all the requests in X_i when they are first encountered. Clearly, this is the optimal route that serves all request in X_i visiting $X_i^0, \ldots, X_i^{|L_i|}$ in order.

If we fix X_i^j for $1 \leq j \leq t$, $0 \leq i \leq R-1$ then note that this determines X_R and T_R, since all requests not served in phases $0 \ldots R-1$ must be served during phase R. The number of choices for X_R^0 is $m+1$. Once X_R^0 is fixed, it is easy to determine the remaining schedule in $O(n)$ time, since this is just the Hamiltonian path problem on a tree. For $1 \leq i < R$ there are $m^{t+1}+1$ possible choices for $X_i^0, , \ldots, X_i^t$. Therefore, the total number of possible schedules is at most $(m+1)(m^{t+1}+1)^{R-1} \leq (m+1)^{(t+1)(R-1)+1}$, which is constant with respect to n.

From these observations, we conclude that there is a polynomial time algorithm for finding the optimal schedule: We enumerate the possible schedules, of which there are at most $(m+1)^{(t+1)(R-1)+1}$, calculating the cost for each, and return the minimum cost schedule.

The calculation of the cost of a schedule, given X_i^j for $1 \leq j \leq t$, $0 \leq i \leq R$, can be accomplished in time $O(Rn)$: We first determine π. This can be accomplished by using depth first search on each sweep to determine the requests served. This takes time $O(Rn)$. From π we can calculate the cost in time $O(n)$.

Therefore, the total running time of the algorithm is $O(R(m+1)^{(t+1)(R-1)+1}n)$, which is linear in n, since t and R are constants and $m \leq n+t-2$.

3 The Offline Single Vehicle Problem

In this section, we present a $(1+\epsilon)$-approximation algorithm for SVSP, for all $\epsilon > 0$. Denote the input problem instance as σ.

Define $r_{\max} = \max_{1 \leq i \leq n} r_i$. Let $a = 2\lceil 1/\epsilon \rceil$ and $\delta = r_{\max}/a$. Since $\text{cost}(\sigma) \geq r_{\max}$, we have $\delta \leq \epsilon\, \text{cost}(\sigma)/2$.

Let P be the sum of all edge weights in $\mathcal{M}$. Define $b = 2(t+1)a^2$ and $\Delta = P/b$. Since every edge must be traversed to serve all requests, $\text{cost}(\sigma) \geq P$ and therefore $\Delta \leq \epsilon\, \text{cost}(\sigma)/(4(t+1)m)$. We define a new metric $\mathcal{N}$ with a constant number of points, which we use to approximate $\mathcal{M}$. A *junction* of $\mathcal{M}$ is defined to be a vertex of degree three or more. Define a *essential path* of $\mathcal{M}$ to be path whose endpoints are either leaves or junctions. $\mathcal{M}$ has a unique decomposition into a set E of at most $2t-2$ essential paths. We find this decomposition and perform the following operation on each essential path $p \in E$: We embed p in the real line, with an arbitrary endpoint at position 0. The other endpoint lies at position $|p|$. This assigns each vertex v in p a non-negative coordinate value $x(v)$. We get a new path p' by rounding the coordinates to get $x'(v) = \min\{|p|, \Delta\lfloor x(v)/\Delta + 1/2 \rfloor\}$. p' consists $y = \lceil |p|/\Delta \rceil$ vertices, $y-1$ edges of length Δ, and one edge of length $|p| - \Delta(y-1)$. There is an obvious mapping from vertices in p to those in p'. From this, we get a mapping ϕ from points in $\mathcal{M}$ to points in $\mathcal{N}$. Note that the number of points in $\mathcal{N}$ is at most

$$\sum_{p\in E} \lceil |p|/\Delta \rceil \le \sum_{p\in E} |p|/\Delta + 1$$
$$\le P/\Delta + 2t - 2 = b + 2t - 2.$$

Using $\mathcal{N}$, we define two new problem instances $\sigma^{\uparrow}$ and $\sigma^{\downarrow}$. Problem $\sigma^{\downarrow}$ is defined in terms of σ by $r_i^{\downarrow} = \delta\lfloor r_i/\delta \rfloor$, $p_i^{\downarrow} = \phi(p_i)$, $h_i^{\downarrow} = h_i$, for $1 \le i \le n$. For purposes that shall become clear, we add a request at the origin to $\sigma^{\downarrow}$, the 0th request, with $p_0^{\downarrow} = p_0$, $r_0^{\downarrow} = 0$ and $h_0^{\downarrow} = 0$. Clearly, this additional request does not affect the solution of $\sigma^{\downarrow}$ in any way, since the vehicle is already at p_0 at time 0, and the request has zero handling time. Note that in $\sigma^{\downarrow}$ there are at most $a + 1$ distinct release times and $b + 2t - 2$ distinct job positions (not including p_0). Problem $\sigma^{\uparrow}$ is defined by $r_i^{\uparrow} = r_i^{\downarrow} + \delta$, $p_i^{\uparrow} = p_i^{\downarrow}$ and $h_i^{\uparrow} = h_i$ for $1 \le i \le n$. As with $\sigma^{\downarrow}$, in $\sigma^{\uparrow}$ there is an additional request at the origin, the 0th request, with $p_0^{\uparrow} = p_0$, $r_0^{\uparrow} = 0$ and $h_0^{\uparrow} = \delta$. Using the algorithm described in the preceding section, we can solve $\sigma^{\downarrow}$ exactly in time $O(n\,(a+1)(b+2t-1)^{(t+1)a+1}) = O(n\,(8(t+1)\lceil 1/\epsilon \rceil^2 + 2t - 1)^{2(t+1)\lceil 1/\epsilon\rceil+1}/\epsilon)$, which is linear in n for constant t and ϵ.

We now observe that an optimal schedule π for $\sigma^{\downarrow}$ is also an optimal schedule for $\sigma^{\uparrow}$. Intuitively, problem $\sigma^{\uparrow}$ is the same as problem $\sigma^{\downarrow}$ but with all requests except 0 shifted back δ time units. Applied to $\sigma^{\uparrow}$ schedule π stays at p_0 until time δ, since $\pi(0) = 0$ and $h_0^{\uparrow} = \delta$, and then travels the same route as for $\sigma^{\downarrow}$, except that each point is reached δ time units later. The cost incurred by π on $\sigma^{\uparrow}$ is therefore $\mathrm{cost}(\sigma^{\downarrow}) + \delta$. Note that when π is used for $\sigma^{\uparrow}$ every request is served after its release time in σ. With a bit of care, we can also use π as a schedule for σ. To ensure that the vehicle reaches all jobs, we have to increase the length of each sweep, but by at most Δ each. Therefore π is also a schedule for σ with cost at most $\mathrm{cost}(\sigma^{\downarrow}) + \delta + (t+1)m\Delta$.

We now relate $\mathrm{cost}(\sigma^{\downarrow})$ to $\mathrm{cost}(\sigma)$. To accomplish this, we consider a third modified instance, which we denote σ^*. This instance is defined in terms of the original metric $\mathcal{M}$ by $r_i^* = r_i^{\downarrow}$, $p_i^* = p_i$, $h_i^* = h_i$, for $1 \le i \le n$. We first observe that clearly, $\mathrm{cost}(\sigma) \ge \mathrm{cost}(\sigma^*)$. The optimal schedule π^* for σ^* has the structure that we have explained in the preceding section; i.e. at most $t + 1$ sweeps per phase are sufficient. Note that if we apply π^* to $\sigma^{\downarrow}$, we have a feasible schedule for $\sigma^{\downarrow}$. Each sweep still covers the same jobs, since the rounding scheme used to obtain $\mathcal{N}$ does not change the order of points along any essential path. Further, we increase the length of each sweep by at most Δ. Therefore, $\mathrm{cost}(\sigma^*) + (t+1)m\Delta \ge \mathrm{cost}(\sigma^{\downarrow})$.

We conclude that the cost incurred by the algorithm is at most

$$\mathrm{cost}(\sigma^{\downarrow}) + \delta + (t+1)m\Delta \le \mathrm{cost}(\sigma^*) + \delta + 2(t+1)m\Delta$$
$$\le \mathrm{cost}(\sigma) + \delta + 2(t+1)m\Delta \le (1+\epsilon)\,\mathrm{cost}(\sigma).$$

4 The Offline Zone Multiple Vehicle Problem

In this section, we show that if we have a ρ-approximation algorithm $\mathcal{A}$ for SVSP which runs in time $O(g(n))$, then we also have a ρ-approximation algorithm $\mathcal{B}$ for ZMVSP which runs in time $O(tkn^t + n^t g(n))$. The basic idea is to generalize the dynamic programming algorithm given by Karuno and Nagamochi [10] for computing the optimal one way zone schedule for the multiple vehicle scheduling problem. The general case is quite complicated, so we begin by looking at the special case of $t = 2$, where $\mathcal{M}$ is a path. We assume in this section that the starting and finishing positions of each vehicle can be selected by the algorithm.

Since the requests are all on a single path, we assume that they are given in order along this path, i.e. request 1 is at one end of the path, request 2 is adjacent to request 1, etc.... Define $C^*(i,j)$ for $1 \le i \le j \le n$ to be the optimal cost for serving requests $i, \ldots, j$ using a single vehicle. Further define $x^*(i,\ell)$ for $1 \le i \le n$ and $1 \le \ell \le k$ to be the cost of the optimal zone schedule for serving requests $1, \ldots, i$ with ℓ vehicles. Then the cost of the optimal zone schedule for the entire problem is given by $x^*(n,k)$. We calculate x^* using the following recurrence $x^*(i,1) = C^*(1,i)$ giving

$$x^*(i,\ell) = \min_{1 \le j < i} \max\left\{x^*(j,\ell-1),\, C^*(j+1,i)\right\}. \tag{1}$$

Of course, we do not know how to calculate $C^*(i,j)$ in polynomial time. We are therefore led to consider the following modified recurrence. Define $C(i,j)$ for $1 \le i \le j \le n$ to be the cost incurred by $\mathcal{A}$ for serving requests $i, \ldots, j$ with a single vehicle. Define $x(i,\ell)$ for $1 \le i \le n$ and $1 \le \ell \le k$ to be minimum cost of a zone schedule for serving requests $1, \ldots, i$ with ℓ vehicles using $\mathcal{A}$ to serve requests in each zone. Similar to the situation with x^*, we calculate $x(n,k)$ using $x(i,1) = C(1,i)$ giving

$$x(i,\ell) = \min_{1 \le j < i} \max\left\{x(j,\ell-1),\, C(j+1,i)\right\}. \tag{2}$$

Using induction, one can show that $x(i,\ell) \le \rho\, x^*(i,\ell)$ for $1 \le i \le n$ and $1 \le \ell \le k$. In particular, this means that $x(n,k) \le \rho\, x^*(n,k)$, which leads us to a ρ-approximation algorithm $\mathcal{B}$:

1. Calculate the values $C(i,j)$ for $1 \le i \le j \le n$, storing them in an array.
2. Calculate $x(n,k)$ using dynamic programming (i.e. store x in an array).
3. From x find the zone partition and use the schedule of $\mathcal{A}$ within each zone.

Step 1 takes $O(n^2 g(n))$ time. Step 2 takes $O(kn^2)$ time. Step 3 can be accomplished in $O(k)$ time if we record the values of j minimizing (2) in Step 2.

We now sketch the general solution. To begin, we calculate the cost $C(T)$ for serving the requests in each subtree T of $\mathcal{M}$ with a single vehicle. The total number of subtrees is at most n^t, so the time to do this is $O(n^t g(n))$. We pick an arbitrary leaf r and designate it to be the root. The partial solutions we build are subtrees of $\mathcal{M}$ containing r. To find the minimum cost $x(T,\ell)$ of an ℓ vehicle solution for a rooted tree T, we use depth first search starting from each

leaf of T, excluding the root. At each point in the depth first search, we have a decomposition of T into two disjoint subtrees: the portion of T visited in the depth first search, which we call U, and the remainder, which we call $V = T - U$. V contains the root. The minimum of $\max\{x(V, \ell - 1), C(U)\}$ over all possible U and V gives us the minimum cost for T. The time required to calculate $x(T, \ell)$ in $O(tn)$, since T has at most t leaves. The number of rooted trees T is at most $O(n^{t-1})$. The total time used is therefore $O(n^t g(n) + tkn^t)$, as claimed.

We now make a number of remarks on the relationship between the cost of the optimal zone schedule, and the optimal non-zone schedule. If we allow multiple requests to appear at a single location, then clearly the cost of the optimal zone schedule can be k times the cost of the optimal non-zone schedule: Consider an input where $n = k$, $r_j = 0$, $h_j = 1$ and $p_j = p_1$ for $1 \leq j \leq k$. Then in a zone schedule, a single vehicle must serve all requests, whereas in a non-zone schedule we can devote a vehicle per request. If requests must occur at distinct locations, then we get a weaker bound stated in the following lemma. The proof will be given in the full version.

Lemma 2. *For all $k \geq 2$ and $t \geq 2$, there exists an MVSP problem instance σ where the cost of the optimal zone schedule is $2 - 1/t$ times the cost of the optimal non-zone schedule.*

5 Other Results

Tsitsiklis [17] shows that SVSP with deadlines on paths is strongly NP-hard, but leaves open the complexity of SVSP with general deadlines and zero release times. In the full version, we show that this problem is NP-hard on paths and strongly NP-hard on trees.

The problem of scheduling a single vehicle online when all handling times are zero is investigated by Ausiello *et al.* [1]. The FO and RTO variants of this problem are of interest. For both FO and RTO, they obtain results for general metrics, and stronger results for paths.

We show that if one has a c-competitive online algorithm for zero handling times, then it is possible to get a $(c + 1)$-competitive online algorithm for non-negative handling times. The proof will be given in the full version.

6 Conclusions

We have presented the first approximation schemes for single and multiple vehicle scheduling problems on trees. Such problems are well motivated, having a large number of applications [10]. We believe that this paper is just an initial step in the exploration of such problems and so we state several open problems. Can the 2-approximation given in [10] for the non-zone multiple vehicle problem on a path be extended to trees? Is an FPTAS possible for SVSP on paths or trees with a constant number of leaves? For what other metrics is a PTAS possible? In [1], lower bounds are given for online vehicle scheduling problems with zero handling costs. Can these lower bounds be increased using handling costs?

References

1. Ausiello, G., Feuerstein, E., Leonardi, S., Stougie, L., and Talamo, M. Algorithms for the on-line travelling salesman. *Algorithmica 29*, 4 (2001), 560–581.
2. Bruno, J., and Downey, P. Complexity of task sequencing with deadlines, set-up times and changeover costs. *SIAM Journal on Computing 7*, 4 (Nov. 1978), 393–404.
3. Charikar, M., Motwani, R., Raghavan, P., and Silverstein, C. Constrained TSP and low-power computing. In *Proceedings of the 5th International Workshop on Algorithms and Data Structures* (Aug. 1997), pp. 104–115.
4. Christofides, N. Worst-case analysis of a new heuristic for the travelling salesman problem. Tech. Rep. CS-93-13, Carnegie Mellon University, Graduate School of Industrial Administration, 1976.
5. Desrosiers, J., Dumas, Y., Solomon, M., and Soumis, F. Time constrained routing and scheduling. In *Network Routing, Volume 8 of Handbooks in Operations Research and Management Science*, M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, Eds. Elsevier Science, 1995.
6. Engebretsen, L., and Karpinski, M. Approximation hardness of TSP with bounded metrics. In *Proceedings of the 28th Annual International Colloquium on Automata, Languages and Programming* (July 2001), pp. 201–212.
7. Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the theory of of NP-Completeness.* Freeman and Company, San Francisco, 1979.
8. Hochbaum, D. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, 1997.
9. Karp, R. M. *Reducibility Among Combinatorial Problems.* Plenum Press, NY, 1972, pp. 85–103.
10. Karuno, Y., and Nagamochi, H. A 2-approximation algorithm for the multi-vehicle scheduling problem on a path with release and handling times. In *Proceedings of the 9th Annual European Symposium on Algorithms* (Aug. 2001), pp. 218–229.
11. Karuno, Y., and Nagamochi, H. A polynomial time approximation scheme for the multi-vehicle scheduling problem on a path with release and handling times. In *Proceedings of the 12th International Symposium on Algorithms and Computation* (Dec. 2001), pp. 36–47.
12. Karuno, Y., Nagamochi, H., and Ibaraki, T. Vehicle scheduling on a tree with release and handling times. *Annals of Operations Research 69* (1997), 193–207.
13. Karuno, Y., Nagamochi, H., and Ibaraki, T. A 1.5-approximation for single-vehicle scheduling problem on a line with release and handling times. In *Japan-U.S.A. Symposium on Flexible Automation* (July 1998), pp. 1363–1366.
14. Nagamochi, H., Mochizuki, K., and Ibaraki, T. Complexity of the single vehicle scheduling problem on graphs. *INFOR: Information Systems and Operational Research 35*, 4 (1997), 256–276.
15. Papadimitriou, C. H. The Euclidean traveling salesman problem is NP-complete. *Theoretical Computer Science 4*, 3 (June 1977), 237–244.
16. Psaraftis, H., Solomon, M., Magnanti, T., and Kim, T. Routing and scheduling on a shoreline with release times. *Management Science 36*, 2 (1990), 212–223.
17. Tsitsiklis, J. Special cases of traveling salesman and repairman problems with time windows. *Networks 22* (1992), 263–282.

Minimizing Makespan for the Lazy Bureaucrat Problem*

Clint Hepner[1] and Cliff Stein[2]

[1] Department of Computer Science,
Dartmouth College, Hanover NH, 03755, chepner@cs.dartmouth.edu
[2] Department of Industrial Engineering and Operations Research,
Columbia University, New York NY, 10027, cliff@ieor.columbia.edu

Abstract. We study the problem of minimizing makespan for the Lazy Bureaucrat Scheduling Problem. We give a pseudopolynomial time algorithm for a preemptive scheduling problem, resolving an open problem by Arkin et al. We also extend the definition of Lazy Bureaucrat scheduling to the multiple-bureaucrat (parallel) setting, and provide pseudopolynomial-time algorithms for problems in that model.

1 Introduction

The Lazy Bureaucrat Scheduling Problem is a modification of traditional scheduling models in which the goal is to minimize, rather than maximize, the amount of work done. The *bureaucrat* is given a list of jobs that have deadlines. His goal is to do as little work as possible (as defined by a given objective), under the *greedy constraint* that he must work on a job if one is available (otherwise, the obvious choice would be to do no work at all). Jobs whose deadlines pass before they are completed expire and can no longer be scheduled; this is desirable, since unscheduled jobs reduce the amount of work the bureaucrat needs to do. The Lazy Bureaucrat Scheduling Problem was introduced by Arkin et al. [1], who considered several different objectives and variants of the greedy constraint.

Arkin et al. motivate the study of this problem with two examples. One supposes an office worker who wants to do as little work as possible while maintaining a busy appearance. For example, suppose he is allowed to go home at 5:00 p.m. At 3:00 p.m., he has two jobs available, which will take 15 minutes and one hour, respectively, to complete. He may work on either, but he must work on one. At 3:15 p.m. he has a personnel meeting which he can skip if he is otherwise busy. He could do the 15-minute job, go to the meeting, then finish the hour-long job by 5:00. However, he can do less work by doing the hour-long job first, which will excuse him from the meeting, followed by the 15-minute job which he completes at 4:15 p.m. The other is the real-life example shown in the movie *Schindler's List*[3], in which the factory workers needed to stay busy without making any real contribution to the German war effort.

* Research partially supported by NSF Grant EIA-98-02068, NSF Grant DMI-9970063 and an Alfred P. Sloan Foundation Fellowship

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 40–50, 2002.

Another example might be a professor scheduling students during his office hour. Being a dedicated teacher, Professor Green does not want to leave before each student present has a chance to ask his question. However, he is leaving for a conference after his office hour, and he would like to finish up as soon as possible so that he has time to pack. Luckily, he knows that some students will leave before seeing him if they get tired of waiting or if they figure out the answer themselves. Therefore, Professor Green may want to see students in an order that will get through all the students as soon as possible, whether or not he actually talks to them.

In this paper, we will study preemptive variants of the Lazy Bureaucrat problem. We further motivate our examination of the preemptive Lazy Bureaucrat problem with the following observation. In the usual deterministic single-machine scheduling models, the use of preemption can "correct" scheduling decisions made prior to the arrival of a new job. When all jobs have the same release date, preemption does not help because the scheduler can already choose from every unscheduled job at every point in time. In the Lazy Bureaucrat Scheduling Problem, preemption can be beneficial even when all release dates are equal, since a job can be left partially completed if it is preempted and never resumed. Partially processing a job allows the scheduler to stay busy just long enough for the deadline of a more costly job to pass. We shall use this unique feature of the Lazy Bureaucrat Scheduling Problem to show that preemption can be used to reduce the makespan of a schedule even if the all the release dates are equal.

We also extend the Lazy Bureaucrat Scheduling Problem by allowing multiple bureaucrats. We restrict ourselves to a model where each bureaucrat works independently so that one bureaucrat cannot prevent another from working. When preemption is allowed, we also make the restriction that a job can only be run by the bureaucrat that started the job, *i.e.*, migration is not allowed.

Arkin et al. define many different Lazy Bureaucrat scheduling problems in [1] and they present NP-hardness results for most problems. They give polynomial-time algorithms for some special cases, and they give pseudopolynomial-time algorithms for some weakly NP-complete problems. Of particular interest to us is a pseudopolynomial-time algorithm for minimizing the makespan of a schedule for a set of jobs which share a common release date.

In this paper, we give pseudopolynomial-time algorithms for the problem of minimizing makespan when preemption is allowed, assuming a job may only be scheduled if it is possible to finish it by its deadline, in both the single- and multiple-machine settings. We first prove structural results that show any preemptive schedule can be converted to a schedule with at most one preemption. Our algorithm converts an instance I of a preemptive problem into a pseudopolynomial number of nonpreemptive instances. One of these new instances has the optimal preemptive schedule of I as its optimal nonpreemptive schedule. The nonpreemptive version has a pseudopolynomial-time algorithm, so the optimal preemptive schedule can be found in pseudopolynomial time as well. In the multiple-bureaucrat setting, we introduce pseudopolynomial-time algorithms for minimizing the makespan of both nonpreemptive and preemptive

schedules with equal release dates by modifying the algorithms for the corresponding single-bureaucrat problems. We show that for a fixed number of bureaucrats, we can assign jobs to bureaucrats with dynamic programming to find the optimal nonpreemptive schedule. When preemption is allowed, we can again convert a problem instance to a set of nonpreemptive problem instances and take the best optimal nonpreemptive schedule as the optimal preemptive schedule.

2 Preliminaries

Definitions and Notation. A *bureaucrat* is an entity capable of doing work, similar to a machine in a traditional scheduling problem. A bureaucrat, however, must obey a *greedy principle*, which states that if at time t there is an unscheduled job that can be run, the bureaucrat must run a job.

An instance of a Lazy Bureaucrat problem is a set J of n jobs, a number m of bureaucrats and a specification of the notion of availability. Each job j has a processing time p_j and a deadline d_j. We define $p_{\max} = \max_j p_j$ to be the maximum processing time and $d_{\max} = \max_j d_j$ to be the maximum deadline. The critical point of a job, $c'_{jt}(\sigma)$, is the latest point in time, as of time t, that job j can be scheduled in schedule σ and still complete by its deadline. If $y_{jt}(\sigma)$ is the amount of processing remaining for job j at time t in schedule σ, then $c'_{jt}(\sigma) = d_j - y_{jt}(\sigma)$. When the schedule σ and the point in time t are clear from context, we abbreviate the critical point and the remaining processing time as c'_j and y_j, respectively. The starting time of a job j is referred to as S_j, and the completion time C_j is the time at which the job has run for p_j units of time. Each job also has a release date r_j before which it may not be processed. For the rest of this paper, we assume that all release dates are equal. This means that by the greedy principle, the schedule may never be idle.

In the preemptive setting, the notion of availability must be defined precisely. For example, if a job cannot be completed before its deadline, should the bureaucrat be allowed to work on it anyway? Three definitions of availability for incomplete jobs are given in [1], which we will denote as *pmtnI*, *pmtnII*, and *pmtnIII*; see that paper for details. Under *pmtnI*, a job may be processed at any time before its deadline passes. A job is only available under *pmtnII* if it can be completed by its deadline; this means a job cannot be scheduled after its critical point. The final constraint, *pmtnIII* requires a job to be completed if it is started. We consider preemptive problems only of type *pmtnII* in this paper.

To more easily refer to Lazy Bureaucrat problems, we introduce the following extensions to the three-field classification scheme due to Graham et al. [2] as follows. Since bureaucrats are identified with machines, they continue to be represented by the usually symbols (1, P, R, *etc.*). The tag *lazy* in the constraints field indicates that the problem is a Lazy Bureaucrat problem. The three objectives studied in [1], makespan, total time spent working, and weighted sum of completed jobs, are specified by $C_{\max}$, $\sum_{ij} t_{ij}$, and $\sum_j w_j(1 - U_j)$, respectively, where t_{ij} is a binary variable that indicates if job j executes at time i, and U_j is a binary variable that indicates if job j does not finish (compare with the slightly

different normal meaning of U_j, which is 1 if a j finishes after its deadline). For example, the problem of minimizing the makespan of a nonpreemptive schedule would be $1\,|\,lazy\,|\,C_{\max}$. If preemption is allowed and only completable jobs are available, the problem would be $1\,|\,lazy, pmtnII\,|\,C_{\max}$.

Previous Results. Many results for Lazy Bureaucrat problems were given in [1]. A brief summary of results relevant to this paper is given here. In the general nonpreemptive case the Lazy Bureaucrat Scheduling Problem is strongly NP-complete for all three objective functions $C_{\max}$, $\sum_{ij} t_{ij}$, and $\sum_j w_j(1 - U_j)$. They are also hard to approximate to within any constant factor. When all release dates are equal, there is a pseudopolynomial-time algorithm. Several special cases have polynomial-time algorithms. When preemption is allowed, hardness results depend on the type of preemption allowed. For all three objective functions, the Lazy Bureaucrat Scheduling Problem is polynomially solvable under *pmtnI*, weakly NP-complete under *pmtnII*, and strongly NP-complete under *pmtnIII*. Open problems include finding algorithms for $1\,|\,lazy, pmtnII\,|\,C_{\max}$, $1\,|\,lazy, pmtnII\,|\,\sum_{ij} t_{ij}$, and $1\,|\,lazy, pmtnII\,|\,\sum_j w_j(1 - U_j)$ that run in pseudopolynomial time, with or without release dates.

3 New Results for the Single Bureaucrat

In this section, we present an algorithm for $1\,|\,lazy, pmtnII\,|\,C_{\max}$ which runs in pseudopolynomial time. We first discuss why it is interesting to study a preemptive scheduling problem when the release dates are equal. We next study the structure of an optimal preemptive schedule for $1\,|\,lazy, pmtnII\,|\,C_{\max}$. Finally, we give a dynamic programming algorithm that runs in $O(n^2\, d_{\max}\, p_{\max})$ time. As pointed out in [1], when all release dates are equal the makespan objective is equivalent to total time spent working and, if all scheduled jobs are completed, weighted sum of completed jobs. Therefore, we focus on the problem of minimizing makespan; our results hold for the other two problems as well.

3.1 Preemption and Makespan with Equal Release Dates

Under normal single-machine scheduling models, preemption does not give a better schedule when all the release dates are equal. Because every job must be completed, one always has the option of running job i before j for any pair of jobs i and j. In other words, any time a job i is preempted by a job j, it is also feasible for job j to run prior to job i in the first place. However, in the Lazy Bureaucrat Scheduling Problem, there are times when it is beneficial to preempt a job and not resume it later, running just enough of the preempted job to prevent other jobs from being scheduled. Consider a three-job instance with $p_1 = 7$, $d_1 = 8$; $p_2 = 7$, $d_2 = 15$; and $p_3 = 20$, $d_3 = 30$. The optimal nonpreemptive schedule runs jobs 1 and 2 for a makespan of 14. If preemption is allowed, we can run the same jobs, but we can preempt job 1 at time 4 for a final makespan of 11. The preempted job is not finished, but it runs long enough to prevent job 3 from

starting. Job 1 itself cannot be resumed after job 2 completes, since its deadline has passed. This shows that preemption can be used to improve the makespan of the schedule by delaying other jobs.

As pointed out in [1], there may not be an optimal solution at all when we allow preemption. Consider our example once more. For any positive value of $\epsilon < 1$, we can preempt job 1 at time $3 + \epsilon$ instead of time 4. This is still long enough to prevent job 3 from starting, and the preemptive schedule remains valid, but the makespan can be made arbitrarily close to 10 for sufficiently small values of ϵ. Because of this feature of the Lazy Bureaucrat Scheduling Problem, it seems like the preemptive version is much harder than the nonpreemptive version. The dynamic programming algorithm from [1] does not seem to be applicable here, since there are an infinite number of possible preemption points.

To allow the problem to be algorithmically solvable, we will define a schedule with integral makespan which can be returned by our algorithm. We refer to schedule ϕ as a *representative* preemptive schedule for a family Φ of preemptive schedules if the makespan of ϕ is $T + 1$, the makespan of each schedule in Φ falls in the interval $(T, T + 1]$, and ϕ can be transformed into any member of Φ by decreasing the processing time of exactly one job. Thus, if in the limit the optimal makespan of a preemptive schedule is $T + \epsilon$ for any $\epsilon > 0$, we need only return a representative schedule of makespan $T + 1$.

3.2 Properties of an Optimal Preemptive Solution

The following lemmas provide us with information about the structure of an optimal preemptive schedule that will allow us to build on the nonpreemptive algorithm for makespan minimization given in [1]. The lemmas build on each other; each proof assumes the previous lemmas have been applied.

First we prove that we only need to consider $O(n)$ possible preemptions.

Lemma 1. *Any preemptive schedule for an instance I of $1\,|\,lazy, pmtnII\,|\,C_{\max}$ can be converted to a preemptive schedule of equal makespan in which no job is resumed once it has been preempted.*

Proof Sketch. We rearrange the scheduled pieces so that each job runs during one continuous interval. Let k be the index of the preempted job that occurs last in the schedule, and let t be time at which the last piece of j_k finishes. We reorder the job pieces within the interval $[S_k, t]$ in a way that introduces no idle time or extra processing, so the makespan does not increase. All pieces of j_k move to the end of the interval; the other job pieces run earlier in the interval to make room. We repeat this process until no job is preempted more than once. □

Figure 1 shows a schedule before and after Lemma 1 is applied to job j_k.

Next, we prove that we only need to preempt one job in the schedule and that every other job we start is allowed to finish.

Lemma 2. *Any preemptive schedule for an instance I of $1\,|\,lazy, pmtnII\,|\,C_{\max}$ can be converted to a preemptive schedule of equal makespan in which at most*

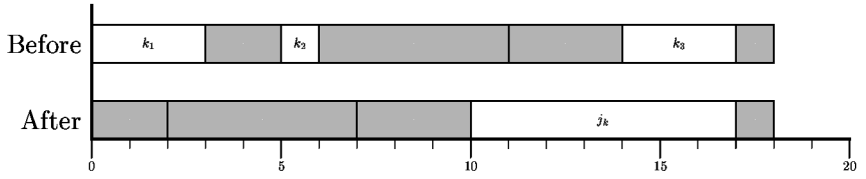


Fig. 1. Application of Lemma 1

one job remains unfinished. Furthermore, if a job is preempted, it is the first job in the schedule.

Proof. We prove this lemma by repeatedly applying a procedure that either reduces the number of unfinished jobs or pushes the last unfinished job earlier in the schedule. Assume ϕ is a preemptive schedule that obeys Lemma 1. Without loss of generality, let the jobs scheduled in ϕ be renumbered in increasing order of their starting times, and let T be the makespan of ϕ.

We start by letting k be the index of the last unfinished job in ϕ and $k-1$ the index of the job that immediately precedes job j_k. If there are no unfinished jobs or $k = 1$, the lemma holds. Otherwise, we either swap j_k and j_{k-1}, or we reassign some or all of the time spent on j_{k-1} to j_k. Let $\Delta = \min\{p_{k-1}, y_{kT}(\phi)\} \geq 0$ be the amount of time we want to take from j_{k-1} to complete j_k. If $T + y_{k-1,T} + \Delta \leq d_{k-1}$, we know that j_{k-1} completes by time T, so we can swap the two jobs. Otherwise, we can reassign the interval $[S_k - \Delta, S_k]$ to j_k. In this case we either complete j_k or we remove j_{k-1} from the schedule entirely. This either reduces the number of unfinished jobs in the schedule, or moves the last unfinished job earlier in the schedule. By repeating this argument, we prove the lemma. □

Figure 2 illustrates Lemma 2 being applied to a schedule.

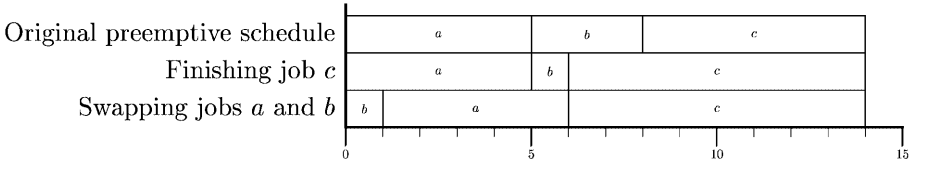


Fig. 2. Application of Lemma 2

Figure 2 shows the first three jobs of a schedule with makespan T during the application of Lemma 2. Job a has $p_a = 5$, $d_a = T + 3$. Job b has $p_b = 3$, $d_b = 15$. Job c has $p_c = 8$, $d_c = 20$. The top line is before Lemma 2 is applied; jobs a and b complete, job c is preempted at time 14. The second line shows job c completing with time taken from job b, since d_b is too early for job b to resume. The third line shows jobs a and b being swapped; if job b takes time from job a, job a remains available at time T

The next lemma further reduces the number of possible preemptive schedules by proving that the completed jobs in a schedule can appear in earliest due date (EDD) order (after the unfinished job, if any). The proof is by a standard pairwise interchange argument and is omitted.

Lemma 3. *For a given job instance I, a preemptive schedule with at most one unfinished job can be converted to a preemptive schedule in which the completed jobs are scheduled in order of nondecreasing deadlines (EDD).*

In the previous section, we claimed we could find a representative preemptive schedule whose makespan could be reduced to the optimal value in the limit.

Lemma 4. *For a given instance I, a preemptive schedule with one preempted one job can be converted to a schedule that runs its unfinished job for an integral amount of time; this change increases the makespan by at most one unit of time.*

Proof. Suppose the unfinished job completes at time $t + \epsilon$ for some $t \in \mathbb{Z}$ and $\epsilon > 0$. Round the time allotted to the unfinished job up to the nearest integer. Since all deadlines and processing times are integral, the set of available jobs does not change between integral times. Therefore, no job is pushed past its deadline by the rounding, and the schedule remains feasible. □

3.3 The Pseudopolynomial Algorithm for $1 \,|\, lazy, pmtnII \,|\, C_{\max}$

To find an optimal preemptive schedule in pseudopolynomial time, we reduce an instance of $1 \,|\, lazy, pmtnII \,|\, C_{\max}$ to a pseudopolynomial number of instances of $1 \,|\, lazy \,|\, C_{\max}$. The preemptive algorithm solves each of the nonpreemptive instances; we will show that the best schedule found is the best preemptive schedule. First, we discuss how to solve the nonpreemptive scheduling problem.

Arkin et al. [1] observe that $1 \,|\, lazy \,|\, C_{\max}$ has a pseudopolynomial-time algorithm based on a dynamic program [4] for minimizing the weighted sum of tardy jobs. We give this algorithm explicitly, as we will refer to the details for our algorithms in the remainder of this paper.

Let the jobs in J be numbered so that so that $d_1 \leq d_2 \leq \ldots \leq d_n$, with ties broken arbitrarily. Define $f(j,t)$ to be the minimum *penalty* of the schedule that completes a subset of jobs 1 through j by time t. The penalty of a schedule is the inverse of the sum of the processing times of the jobs that are not scheduled. The value of $f(j,t)$ is defined recursively as:

$$f(j,\,t) = 0 \qquad j = 0, t = 0 \tag{1}$$

$$f(j,\,t) = +\infty \qquad j = 0, t \neq 0 \tag{2}$$

$$f(j,\,t) = +\infty \qquad j = 1, \ldots, n;\ t < 0 \tag{3}$$

$$f(j,\,t) = \min\{\, f(j{-}1,\, t{-}p_j),\, f(j{-}1, t) - p_j \,\} \quad j = 1, \ldots, n;\ t \geq 0 \tag{4}$$

If an entry $f(j,t)$ is non-positive, it represents a subset of jobs which begins at time zero and ends at exactly time t. Positive entries represent subsets of jobs

that don't exactly fill the interval $[0, t]$. Since the maximum makespan of any feasible schedule is $T = \max\{d_{\max}, \sum_j p_j\} = O(d_{\max})$, we only need to consider values of t up to T. Once the table is filled, each entry $f(j, t)$ for $j = n$ that has non-positive penalty represents a (possibly) feasible schedule whose makespan is t. To ensure that a schedule is feasible, one must check that the unscheduled jobs all have critical times earlier than time t, and that the scheduled jobs complete by their deadlines. If we record the critical point of an unscheduled job and verify that a scheduled job completes on time when the decision is made, we can amortize the time needed for these checks over the time it takes to fill the table. The optimal schedule is the one associated with the table entry with the smallest t index. The running time of the algorithm is $O(n\, d_{\max})$.

Theorem 1. *There is an* $O(n^2\, p_{\max}\, d_{\max})$ *algorithm for* $1 \,|\, lazy, pmtnII \,|\, C_{\max}$.

Proof. Let J be the set of jobs to be scheduled. By Lemmas 1–4, we have established that there is a representative preemptive schedule ϕ for J that runs each scheduled job in one interval of integer length and leaves at most one job unfinished. If all jobs in ϕ complete, it is a nonpreemptive schedule, and the nonpreemptive algorithm can find it. If ϕ contains an unfinished job, we know that it is the first job and it runs for at most $p_{\max}$ units of time. Therefore, a nondeterministic algorithm could guess which job j is the unfinished job and trim its processing time so that ϕ completes it instead of preempting it. In reality, we don't know j or its length, so we enumerate over all the possibilities. We generate $O(n\, p_{\max})$ instances J_{jk} for all $j \in J$ and $k = 1, \ldots, p_{\max}$, where $J_{jk} = J - \{j\} \cup \{j'\}$ and $p'_j = k$. One of these instances is identical to the instance the nondeterministic algorithm creates, and the optimal nonpreemptive schedule of that instance is ϕ. To find an optimal preemptive schedule, we create the $O(n\, p_{\max})$ new instances and solve each instance with a version of the nonpreemptive algorithm, with equations (1) and (2) replaced with $f(j', p'_j) = 0$ and $f(j', t) = +\infty$ for $t \neq p'_j$, respectively. Since each instance J_{jk} can be scheduled in $O(n\, d_{\max})$ time, the lemma is proved. □

It remains open if $1 \,|\, lazy, pmtnII, r_j \,|\, C_{\max}$ is strongly NP-complete.

4 Multiple Bureaucrats

In this section, we discuss extending the Lazy Bureaucrat Scheduling Problem to multiple bureaucrats. In this model, there are m bureaucrats, each of which can process any available job. The greedy requirement still holds, so each bureaucrat must work on a job if one is available. We also require that the bureaucrats obey the *non-interaction principle*, which merely states that the work of one bureaucrat does not affect the work of other bureaucrats. This means that multiple bureaucrats cannot simultaneously work on the same job, nor can one bureaucrat undo the work of another bureaucrat.

The factory example of [1] is a natural example of multiple bureaucrats. While the factory can be viewed as a single bureaucrat that works on jobs one

at a time, it is more realistic to assume that the factory is collection of individual workers and/or groups of workers that can work on multiple jobs simultaneously.

Many of the problems formulated in [1] are NP-complete and hard to approximate, thus they remain so when extended to the multiple-bureaucrat setting. As in the previous section, we will concentrate on minimizing the makespan. We will give pseudopolynomial-time algorithms for both $\mathrm{P}m\,|\,lazy\,|\,C_{\max}$ and $\mathrm{P}m\,|\,lazy, pmtnII\,|\,C_{\max}$. We also show that when moving into the multiple-bureaucrat setting, equivalences among the objective functions that held with a single bureaucrat may no longer hold.

Minimizing Makespan. Because $\mathrm{P}\,|\,lazy\,|\,C_{\max}$ is a generalization of the one machine problem, it is both strongly NP-complete and inapproximable to within any constant factor. Therefore, we focus on problems involving a fixed number of bureaucrats. We represent time with an m-dimensional vector $\boldsymbol{t} = (t_1, \ldots, t_m)$ which stores the time for each of the m bureaucrats separately. When we ask what jobs are scheduled at time $\boldsymbol{t} = (3, 5, 9, 2)$, we are really asking what job the first bureaucrat is running at time 3, the second at time 5, and so on. The makespan of a multiple-bureaucrat schedule is the largest completion time by any bureaucrat. A schedule ending at time $(15, 18, 10, 23)$ has a makespan of 23.

To simplify the use of a time vector, we define the following operator. Let $\ominus_i$ be a subtraction operator that subtracts a value x from only the ith element of a vector. For example, $(10, 12, 8, 11) \ominus_2 3 = (10, 9, 8, 11)$. Formally, $\boldsymbol{t} \ominus_i x \equiv (t_1, \ldots, t_i – x, \ldots, t_m)$. The m-dimensional zero vector is $\mathbf{0}$.

Theorem 2. *$\mathrm{P}m\,|\,lazy\,|\,C_{\max}$ is NP-complete and solvable in $O(n\, d_{\max}^m)$ time.*

Proof. The single-bureaucrat version of this problem is weakly NP-complete; therefore it remains so with multiple bureaucrats. The pseudopolynomial-time algorithm in Section 3.3 can be modified to solve this problem as well by representing time as an m-dimensional vector instead of an integer. This increases the size of the dynamic program's table and therefore the running time.

The dynamic program decides which jobs are scheduled and on which bureaucrats. We define $f(j, \boldsymbol{t})$ to be the minimum penalty of a schedule of a subset of jobs 1 through j in which bureaucrat i finishes by time t_i. The penalty of a schedule is the same as in the single-bureaucrat algorithm. The function f is defined recursively by the following equations:

$$f(j, \boldsymbol{t}) = 0 \qquad (j = 0, \boldsymbol{t} = \mathbf{0}), \tag{5}$$

$$f(j, \boldsymbol{t}) = +\infty \qquad (j = 0,\ \boldsymbol{t} \neq \mathbf{0}), \tag{6}$$

$$f(j, \boldsymbol{t}) = +\infty \qquad (j = 0, 1, \ldots, n;\ \exists i : t_i < 0), \tag{7}$$

$$f(j, \boldsymbol{t}) = \min \left\{ \begin{array}{l} \min\limits_i \{f(j–1, \boldsymbol{t} \ominus_i p_j)\}, \\ f(j–1, \boldsymbol{t}) – p_j \end{array} \right\} \quad (j = 1, 2, \ldots, n;\ t \in \mathbb{N}^m). \tag{8}$$

Whenever we consider adding job j to the schedule now, we consider which is smaller: the penalty of the schedule if j does not run, or the smallest penalty

found by assigning job j to bureaucrat i for each $i = 1, \ldots, m$. As before, each $f(n, \boldsymbol{t})$ entry represents the schedule in which each bureaucrat completes at time t_i. If every bureaucrat does not complete exactly t_i units of processing, $f(n, \boldsymbol{t})$ will have positive infinite value. The optimal schedule is the feasible schedule of $f(n, \boldsymbol{t})$ for which the maximal element of $\boldsymbol{t}$ is minimized. Since there are $O(d_{\max}^m)$ schedules to check, the running time for the entire algorithm is $O(n\, d_{\max}^m)$. □

If a job is required to be processed only by the bureaucrat that begins it, we can use this algorithm to solve $\mathrm{P}m \,|\, lazy, pmtnII \,|\, C_{\max}$ in the same way the single-bureaucrat problem is solved. Treating each bureaucrat separately, Lemmas 1–4 are still applicable to each bureaucrat's jobs. To simplify the analysis, assume that there are m additional jobs with zero processing time to schedule. Since at most m jobs can be preempted (one per bureaucrat), we create $O(n^m p_{\max}^m)$ nonpreemptive instances by enumerating over the $\binom{n+m}{m} = O(n^m)$ sets of jobs we could choose as the preempted jobs. If we modify the above algorithm in the same manner as for the single-bureaucrat preemptive problem, to allow the chosen preempted jobs to run at time 0 on each bureaucrat, we prove the following theorem.

Theorem 3. *There exists an algorithm which runs in* $O(n^{m+1} p_{\max}^m\, d_{\max}^m)$ *time for the problem* $\mathrm{P}m \,|\, lazy, pmtnII \,|\, C_{\max}$.

Difference between Makespan and Total Time Spent Working. We note that with multiple bureaucrats, the makespan and total time spent working objectives are not equivalent when release dates are equal. Suppose two bureaucrats are given three jobs with $p_1 = 5$, $d_1 = 10$; $p_2 = 10$, $d_2 = 10$; and $p_3 = 20$, $d_3 = 30$. There are two schedules which minimize makespan, but only one of those minimizes the total time spent working.

5 Conclusion

We have given a pseudopolynomial-time algorithm for a preemptive Lazy Bureaucrat Scheduling Problem, resolving an open question from [1]. This algorithm uses the the notion of a representative schedule for a set of jobs to represent a family of schedules whose minimal makespan achieves, in the limit, the smallest makespan for that set of jobs. The algorithm also uses the technique of converting a preemptive scheduling problem into a set of nonpreemptive problems; to the best of the authors' knowledge, this technique is new. We also extend the notion of Lazy Bureaucrat Scheduling to the multiple-bureaucrat model, and give two weakly NP-complete multiple-bureaucrat problems, each of which can be solved in pseudopolynomial time with a simple modification to an algorithm for its corresponding single-bureaucrat problem. Open problems include finding hardness results for multiple-bureaucrat problems whose single-bureaucrat versions are polynomially or pseudopolynomially solvable. Other future work may revolve around relaxing the non-interaction constraint on multiple bureaucrats and further exploring other multiple-bureaucrat models.

Acknowledgments. The authors would like to thank Michael Bender for helpful discussions.

References

[1] E. M. Arkin, M. A. Bender, J. S. B. Mitchell, and S. S. Skiena. The lazy bureaucrat scheduling problem. *WADS'99*, 1999.
[2] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.
[3] T. Keneally. Schindler's list. Touchstone Publishers, 1993.
[4] E. L. Lawler and J. M. Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16:77–84, 1969.

A PTAS for the Single Machine Scheduling Problem with Controllable Processing Times

Monaldo Mastrolilli*

IDSIA, Galleria 2, 6928 Manno, Switzerland, monaldo@idsia.ch

Abstract. In a scheduling problem with controllable processing times the job processing time can be compressed through incurring an additional cost. We consider the problem of scheduling n jobs on a single machine with controllable processing times. Each job has a release date, when it becomes available for processing, and, after completing its processing, requires an additional delivery time. Feasible schedules are further restricted by job precedence constraints. We develop a polynomial time approximation scheme whose running time depends only linearly on the input size. This improves and generalize the previous $(3/2+\varepsilon)$-approximation algorithm by Zdrzalka.

1 Introduction

In this paper we consider the following single machine scheduling problem. A set, $J=\{J_1,...,J_n\}$, of n jobs is to be processed without interruption on a single machine. For each job J_j there is an interval $[\ell_j,u_j]$, $0\leq\ell_j\leq u_j$, specifying its possible processing times. The cost for processing job J_j in time ℓ_j is $c_j^\ell\geq 0$ and for processing it in time u_j the cost is $c_j^u\geq 0$. For any value $\delta_j\in[0,1]$ the cost for processing job J_j in time $p_j(\delta_j)=\delta_j\ell_j+(1-\delta_j)u_j$ is $c_j(\delta_j)=\delta_j c_j^\ell+(1-\delta_j)c_j^u$, where δ_j is the *compression parameter*. Additionally, each job J_j has a release date $r_j\geq 0$ when it first becomes available for processing and, after completing its processing on the machine, requires an additional delivery time $q_j\geq 0$; if s_j $(\geq r_j)$ denotes the time J_j starts processing, then it has been delivered at time $s_j+p_j(\delta_j)+q_j$, for compression parameter δ_j. Delivery is a *non-bottleneck* activity, in that all jobs may be simultaneously delivered. Feasible schedules are further restricted by job precedence constraints given by the partial order $\prec$, where $J_j\prec J_k$ means that job J_k must be processed after job J_j. Let η be a permutation of the set J that is consistent with the precedence constraints; η denotes a processing order of jobs. Denote by $Q(\delta,\eta)$ the (earliest) *maximum delivery time* of all the jobs for compression parameters $\delta=(\delta_1,...,\delta_n)$ and processing order η. The *total cost* of compression parameters δ is equal to $\sum_{j\in J}c_j(\delta_j)$, and the *total scheduling cost* for compression parameters δ and processing order η is defined as

* Supported by the "Metaheuristics Network", grant HPRN-CT-1999-00106, and by Swiss National Science Foundation project 20-63733.00/1, "Resource Allocation and Scheduling in Flexible Manufacturing Systems".

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 51–59, 2002.

$$K(\delta, \eta) = Q(\delta, \eta) + \sum_{j \in J} c_j(\delta_j).$$

The problem is to find δ^* and η^* minimizing $K(\delta, \eta)$.

When all processing times are fixed ($\ell_j = u_j$), the problem is equivalent to the well-known sequencing problem denoted as $1|r_j, prec|L_{\max}$ in Graham et al. [1]. Since the special case with fixed processing times and without precedence constraints (noted $1|r_j|L_{\max}$ in [1]) is strongly NP-hard [7], the stated problem is also strongly NP-hard.

Hall and Shmoys [2,4] propose two polynomial time approximation schemes for problem $1|r_j|L_{\max}$, the running time of which are $O((\frac{n}{\varepsilon})^{O(1/\varepsilon)})$ and $O(n \log n + n(1/\varepsilon^{O(1/\varepsilon^2)}))$. For the corresponding problem with controllable processing times, Zdrzalka [9] gives a polynomial time approximation algorithm with a worst-case ratio of $3/2 + \varepsilon$, where $\varepsilon > 0$ can be made arbitrarily small. When the precedence constraints are imposed and the job processing times are fixed ($1|r_j, prec|L_{\max}$), Hall and Shmoys [3] give a PTAS. This consists of executing, for $\log_2 \Delta$ times, an extended version of their previous PTAS for $1|r_j, |L_{\max}$, where Δ denotes an upper bound on the optimal value of any given instance whose data are assumed to be integral. Recently, the author has presented [8] a new PTAS for $1|r_j, prec|L_{\max}$ that runs in $O(n + \ell + 1/\varepsilon^{O(1/\varepsilon)})$ time, where ℓ denotes the number of precedences.

In this paper we provide the first known PTAS for problem $1|r_j, prec|L_{\max}$ with controllable processing times that runs in linear time. This improves and generalizes all the previous results [2,3,4,8,9].

Our algorithm is as follows. We start partitioning jobs into a constant number of subsets (Section 2.1). We show that the precedence graph can be simplified into a more primitive graph (Section 2.2). This simplification depends on the desired precision ε of approximation; the closer ε is to zero, the closer the modified graph will resemble the original one. Then, jobs belonging to the same subset are grouped together into a single compact job to obtain a smaller instance of constant size. The processing times and cost of these compact jobs are constrained to belong to a constant sized set of values; this set is computed by solving a constant number of linear programs (Section 2.3). After this, a non-feasible solution is constructed by allowing preemption. A feasible solution is obtained by processing preempted jobs without interruptions (Section 3).

2 Simplifying the Input

We start by transforming any given instance into a standard form. Let $d_j = \min\{\ell_j + c_j^{\ell}, u_j + c_j^{u}\}$, $D = \sum_{j=1}^{n} d_j$, $r_{\max} = \max_j r_j$ and $q_{\max} = \max_j q_j$. Moreover, let OPT denote the optimal solution value of the given instance.

Lemma 1. *Without loss of generality, we can assume that the following holds:*

- $1 \leq OPT \leq 3$;
- $\max\{D, r_{\max}, q_{\max}\} \leq 1$;
- $0 \leq \ell_j \leq u_j \leq 3$ *and* $0 \leq c_j^u \leq c_j^\ell \leq 3$.

Proof. We begin by bounding the largest number occurring in any given instance. Let $LB = \max\{D, r_{\max}, q_{\max}\}$, we claim that $LB \leq OPT \leq 3LB$. Indeed, since D, $r_{\max}$ and $q_{\max}$ are lower bounds for OPT, LB is also a lower bound for OPT. We show that $3LB$ is an upper bound for OPT by exhibiting a schedule with value at most $3LB$. Starting from time $r_{\max}$ all jobs have been released and they can be scheduled one after the other in any fixed ordering of the jobs that is consistent with the precedence relation; this can be obtained by topologically sorting the precedence graph. Then every job can be completed by time $r_{\max}+D$ and the total scheduling cost is bounded by $r_{\max}+D+q_{\max} \leq 3LB$. By dividing every $\ell_j, u_j, c_j^\ell, c_j^u, r_j$ and q_j by LB, we may (and will) assume, without loss of generality, that $r_{\max}, q_{\max} \leq 1$, $LB = 1$ and $1 \leq OPT \leq 3$.

Furthermore, we can assume, without loss of generality, that $0 \leq \ell_j \leq u_j \leq 3$ and $0 \leq c_j^u \leq c_j^\ell \leq 3$, for all jobs J_j: if $c_j^\ell < c_j^u$, then there exists an optimal solution with $\delta_j = 1$ (i.e., the processing time of job J_j is equal to ℓ_j). Then, we can reset $c_j^u := c_j^\ell$ without affecting the value of the objective function of any feasible schedule. Moreover, in any optimal solution the processing time of any job cannot be larger than 3; therefore, if $u_j > 3$ we can reduce, without loss of generality, the interval of possible processing times and get an equivalent instance by setting $c_j^u := \frac{u_j-3}{u_j-\ell_j}(c_j^\ell - c_j^u) + c_j^u$ and $u_j = 3$. Similar arguments hold if $c_j^\ell > 3$. □

Following Lageweg, Lenstra and Rinnoy Kan [5], if $J_j \prec J_k$ and $r_j > r_k$, then we can reset $r_k := r_j$ and each feasible schedule will remain feasible. Similarly, if $q_j < q_k$ then we can reset $q_j := q_k$ without changing the objective function value of any feasible schedule. Thus, by repeatedly applying these updates we can always obtain an equivalent instance that satisfies

$$J_j \prec J_k \Longrightarrow (r_j \leq r_k \text{ and } q_j \geq q_k) \tag{1}$$

Such a resetting requires $O(\ell)$ time, where ℓ denotes the number of precedence constraints. Thus in the following we assume that (1) holds.

A technique used by Hall and Shmoys [2] allows us to deal with only a constant number of release dates and delivery times. The idea is to round each release and delivery time down to the nearest multiple of $i\varepsilon$, for $i \in \mathbb{N}$. Since $r_{\max} \leq 1$, the number of different release dates and delivery times is now bounded by $1/\varepsilon + 1$. Clearly, the optimal value of this transformed instance cannot be greater than OPT. Every feasible solution for the modified instance can be transformed into a feasible solution for the original instance just by adding ε to each job's starting time, and reintroducing the original delivery times. It is easy to see that the solution value may increase by at most 2ε.

Therefore, we will assume henceforth that the input instance has a constant number of release dates and delivery times, and that condition (1) holds. We shall refer to this instance as I. By the previous arguments, $OPT \geq OPT(I)$, where $OPT(I)$ denotes the optimal value for instance I.

2.1 Partitioning the Set of Jobs

Partition the set of jobs in two subsets:

$$L = \left\{J_j : d_j > \varepsilon^2\right\},$$
$$S = \left\{J_j : d_j \leq \varepsilon^2\right\}.$$

Let us say that L is the set of *large* jobs, while S the set of *small* jobs. Observe that the number of large jobs is bounded by $1/\varepsilon^2$ by Lemma 1. We further partition the set S of small jobs as follows. For each small job $J_j \in S$ consider the following three subsets of L:

$$Pre(j) = \{J_i \in L : J_i \prec J_j\},$$
$$Suc(j) = \{J_i \in L : J_j \prec J_i\},$$
$$Free(j) = L - (Pre(j) \cup Suc(j)).$$

Let us say that $T(j) = \{Pre(j), Suc(j), Free(j)\}$ represents a *3-partition of set* L with respect to job J_j. The number τ of distinct 3-partitions of L is clearly bounded by the number of small jobs and by $3^{|L|} \leq 3^{1/\varepsilon^2}$, therefore $\tau \leq \min\left\{n, 3^{1/\varepsilon^2}\right\}$. Let $\{T_1, ..., T_\tau\}$ denote the set of all distinct 3-partitions. Now, we define the *execution profile* of a small job J_j to be a 3-tuple $< i_1, i_2, i_3 >$ such that $r_j = \varepsilon \cdot i_1$, $q_j = \varepsilon \cdot i_2$ and $T(j) = T_{i_3}$, where $i_1, i_2 = 0, 1, ..., 1/\varepsilon$ and $i_3 = 1, ..., \tau$. For any given instance, the number of distinct execution profiles is clearly bounded by the number of jobs and, by the previous arguments, cannot be larger than $(1 + 1/\varepsilon)^2\tau$.

Corollary 1. *The number π of distinct execution profiles is bounded by* $\pi \leq \min\left\{n, 3^{1/\varepsilon^2}(1 + 1/\varepsilon)^2\right\}$.

Partition the set S of small jobs into π subsets, $S_1, S_2, ..., S_\pi$, such that jobs belonging to the same subset have the same execution profile. Clearly, $S = S_1 \cup S_2, ... \cup S_\pi$ and $S_h \cap S_i = \emptyset$, for $i \neq h$. We illustrate the above by the following.

Example 1. Consider the precedence structure given by the graph in Figure 1. Shaded nodes represent large jobs, while the others denote small jobs. Assume that $r_3 = r_4 = r_9$ and $q_3 = q_4 = q_9$. Since $Pre(3) = Pre(4) = Pre(9) = \{J_1, J_2\}$ and $Suc(3) = Suc(4) = Suc(9) = \{J_7\}$, jobs J_3, J_4 and J_9 establish the same 3-partition of set L and therefore $T(3) = T(4) = T(9)$. Moreover, jobs J_3, J_4 and J_9 have the same execution profile since they have equal release dates and delivery times. Therefore, the set $S = \{J_3, J_4, J_6, J_8, J_9\}$ of small jobs is partitioned into 3 subsets $S_1 = \{J_3, J_4, J_9\}$, $S_2 = \{J_6\}$ and $S_3 = \{J_8\}$.

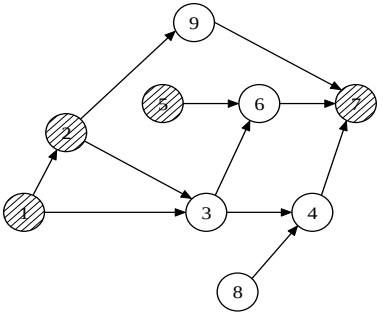


Fig. 1. Graph of Example 1

2.2 Adding New Precedences

Let us say that job J_h is a *neighbor* of set S_i $(i = 1, ..., \pi)$ if:

- J_h is a small job;
- $J_h \notin S_i$;
- there exists a precedence relation between job J_h and some job in S_i.

Moreover, we say that J_h is a *front-neighbor* (*back-neighbor*) of S_i if J_h is a neighbor of S_i and there is a job $J_j \in S_i$ such that $J_j \prec J_h$ ($J_h \prec J_j$).

Let $n_i = |S_i|$ $(i = 1, ..., \pi)$, and let $(J_{1,i}, ..., J_{n_i,i})$ denote any fixed ordering of the jobs from S_i that is consistent with the precedence relation. In the rest of this section we restrict the problem such that the jobs from S_i are processed according to this fixed ordering. Furthermore, every back-neighbor (front-neighbor) J_h of S_i $(i = 1, ..., \pi)$ must be processed before (after) every job from S_i. This can be accomplished by adding a directed arc from $J_{j,i}$ to $J_{j+1,i}$, for $j = 1, ..., n_i - 1$, and by adding a directed arc from J_h to $J_{1,i}$, if J_h is a back-neighbor of S_i, or an arc from $J_{n_i,i}$ to J_h, if J_h is a front-neighbor. Note that the number of added arcs can be bounded by $n + \ell$ (recall that ℓ denotes the number of precedence constraints of the input instance). The above is illustrated by the following.

Example 2. Consider Example 1. Observe that (J_3, J_4, J_9) is an ordering of the jobs from S_1 that is consistent with the precedence relation. Job J_8 is a back-neighbor of S_1, while job J_6 is a front-neighbor of S_1. The new precedence structure is given by the graph in Figure 2, where the new added arcs are emphasized.

We observe that condition (1) is valid also after these changes. Indeed, if J_h is a back-neighbor of S_i then there is a job $J_j \in S_i$ such that $J_h \prec J_j$, and therefore by condition (1) we have $r_h \leq r_j$ and $q_h \geq q_j$. But, the jobs from S_i have the same release dates and delivery times, therefore $r_h \leq r_j$ and $q_h \geq q_j$ for each $J_j \in S_i$. It follows that if we restrict J_h to be processed before the jobs from

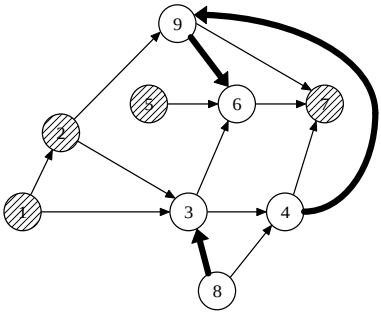


Fig. 2. Graph of Example 2

S_i, condition (1) is still valid. Similar arguments hold if J_h is a front-neighbor. Moreover, all the jobs from S_i have the same release dates and delivery times, therefore condition (1) is still satisfied, if we restrict these jobs to be processed in any fixed ordering that is consistent with the precedence relation.

2.3 Compact Representation of Job Subsets

Consider set S_i $(i = 1, ..., \pi)$. Note that the number of jobs in S_i may be $\Theta(n)$. We replace the jobs from S_i with one *compact* job $J_i^{\#}$. Job $J_i^{\#}$ has the same release $r_i^{\#}$ and delivery time $q_i^{\#}$ as the jobs from S_i. Furthermore, if $J_j \prec J_k$ $(J_k \prec J_j)$, $J_j \in S_i$ and $J_k \notin S_i$, then in the new modified instance we have $J_i^{\#} \prec J_k$ $(J_k \prec J_i^{\#})$. Finally, the processing requirement of $J_i^{\#}$ is specified by a finite set of alternative pairs of processing times and costs determined as follows. Consider the following set $V_S = \left\{\frac{\varepsilon}{\pi}, \frac{\varepsilon}{\pi}(1+\varepsilon), \frac{\varepsilon}{\pi}(1+\varepsilon)^2, ..., 3\right\}$. By simple algebra, we have $|V_S| = O(1/\varepsilon^3)$. Recall that π is the number of distinct execution profiles. Let A_i (and B_i) be the value obtained by rounding $\sum_{j \in S_i} \ell_j$ (and $\sum_{j \in S_i} u_j$) up to the nearest value from set V_S. The possible processing times for $J_i^{\#}$ are specified by set P_i of values from V_S that fall in interval $[A_i, B_i]$, i.e., $P_i := V_S \cap [A_i, B_i]$. For each value $p \in P_i$, we compute the corresponding cost value $C_i(p)$ as follows. Consider the problems (S_i, p) of computing the minimum sum of costs for jobs belonging to S_i, when the total sum of processing times is at most p, for every $p \in P_i$. We can formulate problem (S_i, p) by using the following linear program $LP(S_i, p)$:

$$\begin{array}{ll} \min \sum_{j \in S_i} (\delta_j c_j^{\ell} + (1-\delta_j) c_j^{u}) & \\ \text{s.t.} \ \sum_{j \in S_i} (\delta_j \ell_j + (1-\delta_j) u_j) \leq p & \\ \quad 0 \leq \delta_j \leq 1 & J_j \in S_i \end{array}$$

By setting $\delta_j = 1 - x_j$, it is easy to see that an optimal solution for $LP(S_i, p)$ can be obtained by solving the following linear program:

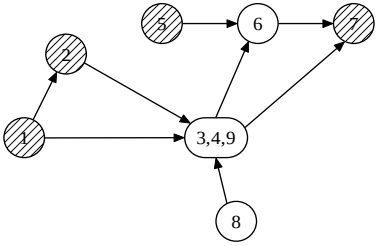


Fig. 3. Graph of Example 3

$$\begin{array}{ll} \max & \sum_{j\in S_i}(c_j^{\ell}-c_j^{u})x_j \\ \text{s.t.} & \sum_{j\in S_i}(u_j-\ell_j)x_j \le p-\sum_{j\in S_i}\ell_j \\ & 0\le x_j\le 1 \qquad\qquad J_j\in S_i \end{array}$$

Note that $p-\sum_{j=1}^{n}\ell_j$ is non-negative, since $p\in P_i$ and the smallest value of P_i cannot be smaller than $\sum_{j=1}^{n}\ell_j$. The previous linear program corresponds to the classical knapsack problem with relaxed integrality constraints. By partially sorting jobs in nonincreasing ratio $(c_j^{\ell}-c_j^{u})/(u_j-\ell_j)$ ratio order, the set $\{LP(S_i,p):p\in P_i\}$ of $O(1/\varepsilon^3)$ many problems can be solved in $O(|S_i|\log\frac{1}{\varepsilon}+(1/\varepsilon^3)\log\frac{1}{\varepsilon})$ time by employing a median-finding routine (we refer to Lawler [6] for details). For each value $p\in P_i$, the corresponding cost value $C_i(p)$ is equal to the optimal solution value of $LP(S_i,p)$ rounded up to the nearest value of set V_S. It follows that the number of alternative pairs of processing times and costs for each compact job $J_i^{\#}$ is bounded by the cardinality of set P_i. Furthermore, since $\sum_{i=1}^{\pi}|S_i|\le n$, it is easy to check that the amortized total time to compute the processing requirements of all compact jobs is $O(n(1/\varepsilon^3)\log\frac{1}{\varepsilon})$. Therefore, every set S_i is transformed into one compact job $J_i^{\#}$ with $O(1/\varepsilon^3)$ alternative pairs of costs and processing times. We use $S^{\#}$ to denote the set of compact jobs.

Example 3. Consider Example 2. We group jobs J_3, J_4 and J_9 together and get a new instance whose precedence structure is given by the graph in Figure 3.

Now, let us consider the modified instance as described so far and turn our attention to the set L of large jobs. We map each large job $J_j\in L$ to a new job $J_j^{\#}$ which has the same release date, delivery time and set of predecessors and successors as job J_j, but a more restricted set of possible processing times and costs. More precisely, let A_j (and B_j) be the value obtained by rounding ℓ_j (and u_j) up to the nearest value from set $V_L=\{\varepsilon^3,\varepsilon^3(1+\varepsilon),\varepsilon^3(1+\varepsilon)^2,...,3\}$. The possible processing times for $J_j^{\#}$ are specified by set $P_j:=V_L\cap[\ell_j,u_j]$. For each value $p\in P_j$, the corresponding cost value $C_j(p)$ is obtained by rounding up to the nearest value of set V_L the cost of job J_j when its processing time is

p. We use $L^{\#}$ to denote the set of jobs obtained by transforming jobs from L as described so far.

Let $I^{\#}$ denote this modified instance. We observe that $I^{\#}$ can be computed in $O(n(1/\varepsilon^3)\log\frac{1}{\varepsilon}+2^{O(1/\varepsilon^2)})$ time: the time required to partition the set of jobs into π subsets can be bounded by $O(n+\ell+2^{O(1/\varepsilon^2)})$; $O(n+\ell)$ is the time to add new precedences; $O(n(1/\varepsilon^3)\log\frac{1}{\varepsilon})$ is the time to compute the alternative pairs of costs and processing times. Moreover, this new instance has at most $\nu = 3^{1/\varepsilon^2}\cdot(1+1/\varepsilon)^2+1/\varepsilon^2$ jobs; each job has a constant number of alternative pairs of costs and processing times. Now let us focus on $I^{\#}$ and consider the problem of finding the schedule for $I^{\#}$ with the minimum scheduling cost such that compact jobs can be preempted, while interruption is not allowed for jobs from $L^{\#}$.

The following lemma shows that the optimal solution value of $I^{\#}$ has value close to $OPT(I)$. Moreover, it gives a bound on the number of preempted jobs. (A proof of the following lemma can be found in the long version of this paper available at: http://www.idsia.ch/˜monaldo/research_papers.html.)

Lemma 2. *For any fixed $\varepsilon > 0$, it is possible to compute in constant time an optimal solution for $I^{\#}$ with at most $1/\varepsilon$ preempted compact jobs. Moreover, $OPT(I^{\#}) \leq (1+4\varepsilon)OPT(I)$.*

3 Generating a Feasible Solution

In this subsection we show how to transform the optimal solution $SOL^{\#}$ for instance $I^{\#}$ into a $(1+O(\varepsilon))$-approximate solution for instance I. This is accomplished as follows.

First, replace the jobs from $L^{\#}$ with the corresponding large jobs. Let $p_j^{\#}$ and $c_j^{\#}$ denote the processing time and cost, respectively, of job $J_j^{\#} \in L^{\#}$ according to solution $SOL^{\#}$, then it is easy to check that the corresponding job $J_j \in L$ can be processed in time and cost at most $p_j^{\#}$ and $c_j^{\#}$, respectively.

Second, we replace each compact job $J_i^{\#}$ with the corresponding small jobs from set S_i as follows. Remove job $J_i^{\#}$, this clearly creates gaps into the schedule. Then, fill in the gaps by inserting the small jobs from set S_i according to any fixed ordering that is consistent with the precedence relation, and by allowing preemption; the processing time and cost of these small jobs are chosen according to the optimal solution of $LP(S_i, p_i^{\#})$ (see Subsection 2.3), where $p_i^{\#}$ denotes the processing time of job $J_i^{\#}$ according to solution $SOL^{\#}$. (Recall that the optimal solution of $LP(S_i, p_i^{\#})$ chooses the processing requirements of jobs from S_i such that the sum of processing times is at most $p_i^{\#}$ and the sum of costs is minimum.) However, these replacements do not yield a feasible solution for I, since there may be a set M of preempted small jobs. By Lemma 2, we have that the number of preempted small jobs is at most $1/\varepsilon$. For each $J_j \in M$ let s_j be the time at which job J_j starts in the preemptive schedule. Remove each $J_j \in M$ and schedule J_j without interruption at time s_j with processing time p_j and cost

c_j, where $p_j + c_j = d_j$. It is easy to see that the maximum delivery time may increase by at most $\sum_{J_j \in M} p_j$ and the cost by at most $\sum_{J_j \in M} c_j$. Therefore, the solution value is increased by at most $\sum_{J_j \in M} d_j \leq |M|\varepsilon^2 \leq \varepsilon \leq \varepsilon \cdot OPT(I)$, since $|M| \leq 1/\varepsilon$, $M \subseteq S$ and $S = \left\{J_j : d_j \leq \varepsilon^2\right\}$.

Finally, we have already observed that every feasible solution for the modified instance with only a constant number of release dates and delivery times can be transformed into a feasible solution for the original instance by simply delaying each job starting time by at most ε, and reintroducing the original delivery times. This may increase the value of the solution by at most 2ε. Therefore, by Lemma 2, the value of the returned solution is at most $(1 + 7\varepsilon) \cdot OPT(I)$, that confirms that this construction does in fact yield an $(1 + O(\varepsilon))$-approximate solution of I. To conclude, we have shown that problem $1|r_j, prec|L_{\max}$ with controllable processing times admits a PTAS.

Theorem 1. *There exists a linear time approximation scheme for problem* $1|r_j, prec|L_{\max}$ *with controllable processing times.*

References

1. R. Graham, E. Lawler, J. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. North–Holland, 1979.
2. L. Hall and D. Shmoys. Approximation algorithms for constrained scheduling problems. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 134–139, 1989.
3. L. Hall and D. Shmoys. Near-optimal sequencing with precedence constraints. In *Proceedings of the 1st Integer Programming and Combinatorial Optimization Conference*, pages 249–260. University of Waterloo Press, 1990.
4. L. Hall and D. Shmoys. Jackson's rule for single-machine scheduling: Making a good heuristic better. *MOR: Mathematics of Operations Research*, 17:22–35, 1992.
5. B. Lageweg, J. Lenstra, and A. R. Kan. Minimizing maximum lateness on one machine: Computational experience and some applications. *Statist. Neerlandica*, 30:25–41, 1976.
6. E. Lawler. Fast approximation algorithms for knapsack problems. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 206–218, 1977.
7. J. Lenstra, A. R. Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Operations Research*, 1:343–362, 1977.
8. M. Mastrolilli. Grouping techniques for one machine scheduling subject to precedence constraints. In *Proceedings of the 21st Foundations of Software Technology and Theoretical Computer Science*, volume LNCS 2245, pages 268–279, 2001.
9. S. Zdrzalka. Scheduling jobs on a single machine with release dates, delivery times, and controllable processing times: worst-case analysis. *Operations Research Letters*, 10:519–532, 1991.

Optimum Inapproximability Results for Finding Minimum Hidden Guard Sets in Polygons and Terrains

Stephan Eidenbenz

Institute of Theoretical Computer Science, ETH Zürich, Switzerland
eidenben@inf.ethz.ch

Abstract. We study the problem Minimum Hidden Guard Set, which consists of positioning a minimum number of guards in a given polygon or terrain such that no two guards see each other and such that every point in the polygon or on the terrain is visible from at least one guard. By constructing a gap-preserving reduction from Maximum 5-Ocurrence-3-Satisfiability, we show that this problem cannot be approximated by a polynomial-time algorithm with an approximation ratio of $n^{1-\epsilon}$ for any $\epsilon > 0$, unless $NP = P$, where n is the number of polygon or terrain vertices. The result even holds for input polygons without holes. This separates the problem from other visibility problems such as guarding and hiding, where strong inapproximability results only hold for polygons with holes. Furthermore, we show that an approximation algorithm achieves a matching approximation ratio of n.

1 Introduction

In the field of visibility problems, guarding and hiding are among the most prominent and most intensely studied problems. In guarding, we are given as input a simple polygon with or without holes and we need to find a minimum number of guard positions in the polygon such that every point in the interior of the polygon is visible from at least one guard. Two points in the polygon are visible from each other, if the straight line segment connecting the two points does not intersect the exterior of the polygon. In hiding, we need to find a maximum number of points in the given input polygon such that no two points see each other.

The combination of these two classic problems has been studied in the literature as well [11]. The problem is called Minimum Hidden Guard Set and is formally defined as follows:

Definition 1. *The problem* Minimum Hidden Guard Set *consists of finding a minimum set of guard positions in the interior of a given simple polygon such that no two guards see each other and such that every point in the interior of the polygon is visible from at least one guard.*

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 60–68, 2002.

We can define variations of this problem by allowing input polygons to contain holes or not or by letting the input be a 2.5 dimensional terrain. A 2.5 dimensional terrain is given as a triangulated set of vertices in the plane together with a height value for each vertex. The linear interpolation inbetween the vertices defines a bivariate continuous function, thus the name 2.5 dimensional terrain (see [10]). In other variations, the guards are restricted to sit on vertices. Problems of this type arise in a variety of applications, most notably in telecommunications, where guards correspond to antennas in a network with a simple line-of-sight wave propagation model (see [4]).

While Minimum Hidden Guard Set is NP-hard for input polygons with or without holes [11], no approximation algorithms or inapproximability results are known. For other visibility problems, such as guarding and hiding, the situation is different: Minimum Vertex/Point/Edge Guard are NP-hard [9] and cannot be approximated with an approximation ratio that is better than logarithmic in the number of polygon or terrain vertices for input polygons with holes or terrains [4]; these problems are APX-hard[1] for input polygons without holes [4]. The best approximation algorithms for these guarding problems achieve a logarithmic approximation ratio for Minimum Vertex/Edge Guard for polygons [8] and terrains [6], which matches the logarithmic inapproximability result upto low-order terms in the case of input polygons with holes and terrains; the best approximation ratio for Minimum Point Guard is $\Theta(n)$, where n is the number of polygon or terrain vertices. The problem Maximum Hidden Set cannot be approximated with an approximation ratio of n^{ϵ} for some $\epsilon > 0$ for input polygons with holes and it is APX-hard for polygons without holes ([5] or [7]). The best approximation algorithms achieve approximation ratios of $\Theta(n)$. Thus, for both, hiding and guarding, the exact inapproximability threshold is still open for input polygons without holes. To get an overview of the multitude of results in visibility problems, consult [12] or [13].

In this paper, we present the first inapproximability result for Minimum Hidden Guard Set: we show that no polynomial-time algorithm can guarantee an approximation ratio of $n^{1-\epsilon}$ for any $\epsilon > 0$, unless $NP = P$, where n is the number of vertices of the input structure. The result holds for terrains, polygons with holes, and even polygons without holes as input structures. We obtain our result by constructing a gap-preserving reduction (see [1] for an introduction to this concept) from Maximum 5-Occurrence-3-Satisfiability, which is the APX-hard satisfiability variation, where each clause consists of at most three literals and each variable occurs at most five times as a literal [2]. We also analyze an approximation algorithm for Minimum Hidden Guard Set proposed in [11] and show that it achieves a matching approximation ratio of n.

[1] A problem is in the class APX, if it can be approximated by a polynomial-time algorithm with an approximation ratio of $1+\delta$, for some constant $\delta \geq 0$. It is APX-hard, if no polynomial-time algorithm can guarantee an approximation ratio of $1+\epsilon$, for some constant $\epsilon > 0$, unless $P = NP$. A problem is APX-complete, if it is in APX and APX-hard. See [2] for more details.

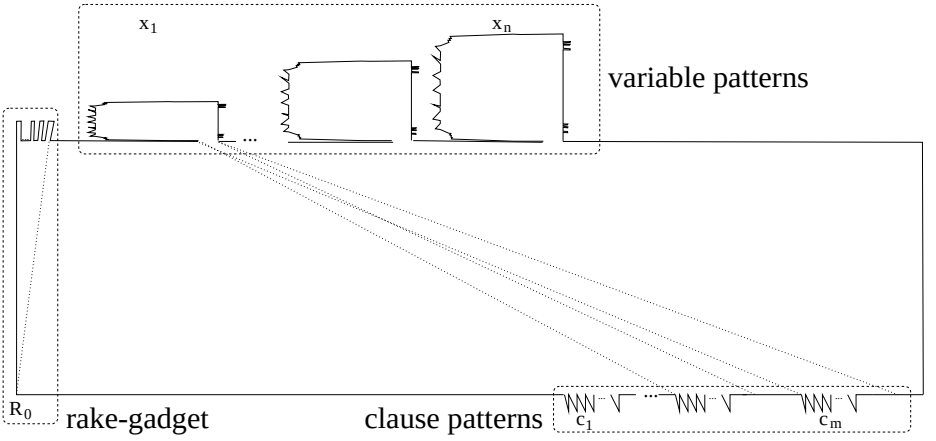


Fig. 1. Overview of construction

In Sect. 2 we present the construction of the reduction. We analyze the reduction and obtain our main result in Sect. 3. We analyze an approximation algorithm in Sect. 4. Section 5 contains some extensions of our results and concluding thoughts.

2 Construction of the Reduction

In this section, we show how to construct in polynomial time from a given instance I of Maximum 5-Occurrence-3-Satisfiability with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$ an instance I' of Minimum Hidden Guard Set, i.e., a simple polygon.

An overview of the construction is given in Fig. 1. The main body of the constructed polygon is of rectangular shape. For each clause c_i, a *clause pattern* is constructed on the lower horizontal line of the rectangle, and for each variable x_i, we construct a *variable pattern* on the upper horizontal line as indicated in Fig. 1.

The construction will be such that a variable assignment that satisfies all clauses of I exists, if and only if the corresponding polygon I' has a hidden guard set with $O(n)$ guards; otherwise, I' has a hidden guard set of size $O(t)$, where t will be defined as part of the *rake-gadget* in the construction. The rake gadget, shown in Fig. 2, enables us to force a guard to a specific point R in the polygon. It consists of t dents, which are small trapezoidal elements that point towards point R. Rakes have the following property:

Lemma 1. *If the t dents of a rake are not covered by a single hidden guard at point R, then t hidden guards (namely one guard for each dent) are necessary to cover the dents.*

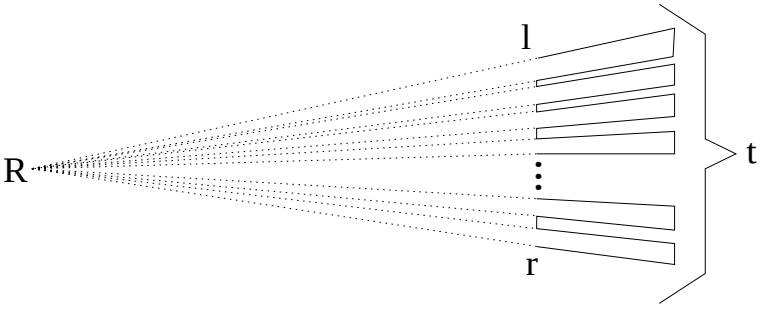


Fig. 2. Rake with t dents

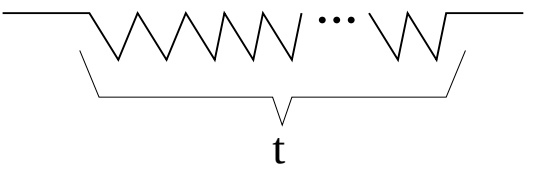


Fig. 3. Clause pattern consisting of t triangles

Proof. Clearly, any guard outside the triangle R, l, and r and outside the dents does not see a single dent completely. A guard in this triangle (but not at R) sees at most one dent completely, but only one such guards can exist as guards must be hidden from each other. Therefore, at least $t-1$ guards must be hidden in the dents. □

In order to benefit from this property of a rake, we must place the rake in the polygon in such a way that the view from point R to the rake dents is not blocked by other polygon edges. As shown in Fig. 1, we place a rake at point R_0 in the lower left corner of the rectangle with the t dents at the top left corner.

A clause pattern, shown in Fig. 3, consists of t triangular-shaped spikes. Clause patterns are placed on the lower horizontal line of the rectangle. They are constructed in such a way that a guard on the upper horizontal line could see all spikes of all clause patterns. (This, however, will never happen, as we have already forced a guard to point R_0 to cover its rake. This guard would see any guard on the upper horizontal line.)

For each variable x_i, we construct a variable pattern, that is placed on top of the horizontal line of the rectangle. Each variable pattern opens the horizontal line for a unit distance. Each variable pattern has constant distance from its neighbors and the right-most variable pattern (for variable x_n) is still to the left of the left-most clause pattern (for clause c_1), as indicated in Fig. 1. The variable patterns will differ in height, with the left-most variable pattern (for x_1) being the smallest and the right-most (for x_n) the tallest. Figure 4 shows the variable pattern of variable x_i.

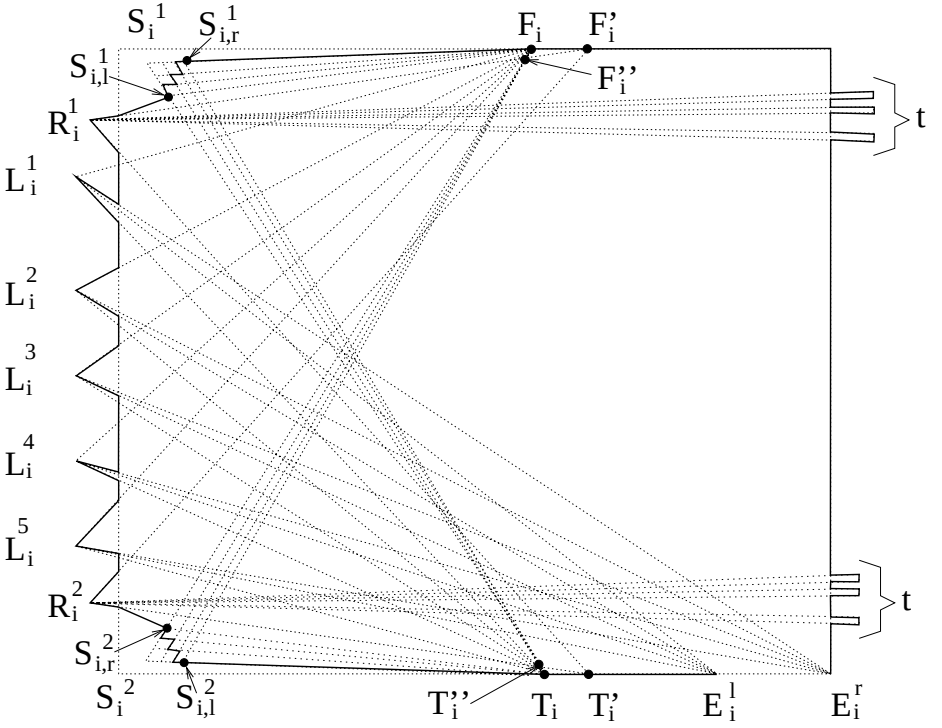


Fig. 4. Variable pattern with three positive and two negative literals

A variable pattern is roughly a rectangular structure with a point F_i on top and a point T_i on the bottom. The construction is such that a guard sits at F_i, if the variable is set to false, and at T_i otherwise. Literals are represented by triangles with tips $L_i^1, \ldots, L_i^5$ for each of the five occurrences of the variable (some may be missing, if a variable occurs less than five times as literal). These triangles are constructed such that – for positive literals – they are completely visible from F_i, but not from T_i, and – for negative literals – they are completely visible from T_i, but not from F_i. A guard that sits at a point L_i^k, for any $k = 1, \ldots, 5$, can see through the exit of the variable pattern between points E_i^l and E_i^r. The construction is such that such a guard sees all spikes of the corresponding clause pattern (but no spikes of other clause patterns). This is shown schematically in Fig. 1.

In order to force a guard to sit at either F_i or T_i, we construct a rake point R_i^1 above L_i^1 and a rake point R_i^2 below L_i^5 with t dents, all of which are on the right vertical line of the variable rectangle. Points R_i^1 and R_i^2 are at the tip of small triangles that point towards points F_i' and T_i', which lie a small distance to the right of F_i and T_i, respectively. In addition, we construct two areas S_i^1 and S_i^2 to the left of T_i and F_i, where we put t triangular spikes, each pointing exactly towards F_i and T_i. For simplicity, we have only drawn three triangular

spikes in Fig. 4 instead of t. Area S_i^1 is the area of all these triangles at the top of the variable rectangle, area S_i^2 is the area of all these triangles at the bottom of the variable rectangle.

This completes our description of the constructed polygon that is an instance of Minimum Hidden Guard Set. The polygon consists of a number of vertices that is polyonmial in the size $|I|$ of the Max 5-Occurrence-3-Satisfiability instance I and in t. The coordinates of each vertex can be computed in time polynomial in $|I|$ and t, and they can be expressed by a polynomial (in $|I|$ and t) number of bits. Thus, the reduction is polynomial, if t is polynomial in $|I|$.

3 Analysis of the Reduction

The following two lemmas describe the reduction as gap-preserving and will allow us to prove our inapproximability result.

Lemma 2. *If the* Maximum 5-Occurrence-3-Satisfiability *instance I with n variables can be satisfied by a variable assignment, then the corresponding* Minimum Hidden Guard Set *instance I' has a solution with at most $8n+1$ guards.*

Proof. In I', we set a guard at each rake point R_0 and R_i^1 and R_i^2, for $i = 1, \ldots, n$, which gives a total of $2n+1$ hidden guards. For each variable x_i, we then place a guard at F_i or T_i depending on the truth value of the variable in a fixed satisfying truth assignment; this yields additional n hidden guards. Finally, we place a guard at each literal L_i^k, if and only if the corresponding literal is true. This yields at most $5n$ hidden guards, as each variable occurs at most five times as a literal. Since the thruth assignment satisfies all clauses, all clause patterns will be covered by at least one guard. The variable patterns and the main body rectangle are covered completely as well. Thus, the solution is feasible and consists of at most $8n+1$ guards. □

Lemma 3. *If the* Maximum 5-Occurrence-3-Satisfiability *instance I with n variables cannot be satisfied by a variable assignment, then any solution of the corresponding* Minimum Hidden Guard Set *instance I' has at least t guards.*

Proof. We prove the following equivalent formulation: If I' has a solution with strictly less than t guards, then I is satisfiable.

Assume we have a solution for I' with less than t guards. Then, there must be a guard at each rake point R_0 and R_i^1 and R_i^2 for $i = 1, \ldots, n$; this already restricts the possible positions for all other guards quite drastically, since they must be hidden from each other.

Observe in this solution, how the triangles of the areas S_i^1 and S_i^2 are covered. Since we have guards at rake points R_i^1 and R_i^2, the guards for S_i^1 and S_i^2 can only lie in the 4-gons $(S_{i,l}^1, S_{i,r}^1, F_i, F_i')$ or $(S_{i,r}^2, S_{i,l}^2, T_i, T_i')$, but only a guard in the smaller triangle of either (F_i, F_i', F_i'') or (T_i, T_i', T_i'') can see both areas S_i^1

and S_i^2 (see Fig. 4). If S_i^1 or S_i^2 is covered by a guard outside these triangles, then the other area can only be covered with t guards inside the S_i^1 or S_i^2 triangles. Therefore, there must be a guard in either one of the two triangles (F_i, F_i', F_i'') or (T_i, T_i', T_i'') in each variable pattern. (Point F_i'' is the intersection point of the line from R_i^1 to F_i' and from $S_{i,r}^2$ to F_i; Point T_i'' is the intersection point of the line from R_i^2 to T_i' and from $S_{i,l}^1$ to T_i). We can move this guard to point F_i or T_i, respectively, without changing which literal triangles it sees.

Now, the only areas in the construction not yet covered are the literal triangles of those literals that are true and the spikes of the clause patterns. Assume for the sake of contradiction that one guard is hidden in a triangle of a clause pattern c_i. This guard sees the triangles of all literals that represent literals from the clause. This, however, implies that the remaining $t-1$ triangles of the clause pattern c_i can only be covered by $t-1$ additional guards in the clause pattern, thus resulting in t guards total. Therefore, all remaining guards must sit in the literal triangles in the variable patterns. W.l.o.g., we assume that there is a guard at each literal point L_i^k that is not yet covered by a guard at points F_i or T_i. If these guards collectively cover all clause patterns, we have a satisfying truth assignment; if they do not, at least t guards are needed to cover the remaining clause patterns. □

Lemmas 2 and 3 immediately imply that we cannot approximate Minimum Hidden Guard Set with an approximation ratio of $\frac{t}{8n+1}$ in polynomial time, because such an algorithm could be used to decide Maximum 5-Occurrence-3-Satisfiability. To get to an inapproximability result, we first observe that

$$|I'| \leq (8t+30)n + 2tm + 4t + 100 \leq 18tn + 30n + 4t + 100$$

by generously counting the constructed polygon vertices and using $m \leq 5n$. We now set

$$t = n^k$$

for an arbitrary but fixed $k > 1$. This implies $|I'| \leq n^{k+2}$ and thus

$$n \geq |I'|^{\frac{1}{k+2}}$$

On the other hand, we cannot approximate Minimum Hidden Guard Set with an approximation ratio of

$$\frac{t}{8n+1} \geq \frac{n^k}{n^2} = n^{k-2} \geq |I'|^{\frac{k-2}{k+2}} = |I'|^{1-\frac{4}{k+2}}.$$

Since k is an arbitrarily large constant, we have shown our main theorem:

Theorem 1. Minimum Hidden Guard Set *on input polygons with or without holes cannot be approximated by any polynomial time approximation algorithm with an approximation ratio of* $|I|^{1-\epsilon}$ *for any* $\epsilon > 0$*, where* $|I|$ *is the number of polygon vertices, unless* $NP = P$.

4 An Approximation Algorithm

The following algorithm to find a feasible solution for Minimum Hidden Guard Set was proposed in [11]: Iteratively add a guard to the solution by placing it in an area of the input polygon (or terrain) that is not yet covered by any other guard that is already in the solution. In terms of an approximation ratio for this algorithm, we have the following

Theorem 2. Minimum Hidden Guard Set *can be approximated in polynomial time with an approximation ratio of* $|I|$*, where* $|I|$ *is the number of polygon vertices.*

Proof. Any triangulation of the input polygon partitions the polygon into $|I|-2$ triangles. Now, fix any triangulation. Any guard that the approximation algorithm places (as described above) lies in at least one of the triangles of the triangulation and thus sees the corresponding triangle completely. Therefore, the solution will contain at most $|I|-2$ guards. Since any solution must consist of at least one guard, the result follows. □

5 Extensions and Conclusion

Theorem 1 extends straight-forward to terrains as input structures by using the following transformation from a polygon to a terrain (see [4]): Given a simple polygon, draw a bounding box around the polygon and then let all the area in the exterior of the polygon have height h (for some $h > 0$) and the interior height zero. This results in a terrain with vertical walls that we then triangulate. Similarly, Theorem 2 extends to terrains as input structures immediately.

Another straight-forward extension of Theorem 1 leads to problem variations, where the guards may only sit at vertices of the input structure. Since we have always placed or moved guards to vertices throughout our construction, Theorem 1 holds for Minimum Hidden Vertex Guard Set for input polygons with or without holes and terrains. Unfortunately, the vertex-restricted problem variations cannot be approximated analogously to Theorem 2, as even the problem of determining whether a feasible solution exists for these problems is NP-hard [11].

If we restrict the problem even more, namely to a variation, where the guards may only sit at vertices and they only need to cover the vertices rather than the whole polygon interior, we arrive at the problem Minimum Independent Dominating Set for visibility graphs. Also in this case, Theorem 1 holds, thus adding the class of visibility graphs to the numerous graph classes for which this problem cannot be approximated with a ratio of $n^{1-\epsilon}$. The approximation algorithm from Sect. 3 can be applied for this variation and achieves a matching ratio of n.

The complementary problem Maximum Hidden Guard Set, where we need to find a maximum number of hidden guards that cover a given polygon, is equivalent to Maximum Hidden Set. Therefore, it cannot be approximated

with an approximation ratio of n^ϵ for some $\epsilon > 0$ for input polygons with holes and it is APX-hard for input polygons without holes [7]. The corresponding vertex-restricted variation cannot be approximated, as it is – again – NP-hard to even find a feasible solution.

We have presented a number of inapproximability and approximability results for Minimum Hidden Guard Set in several variations. Most results are tight upto low-order terms. However, there still exists a large gap regarding the inapproximability of the problem Maximum Hidden Guard Set on input polygons without holes, where only APX-hardness is known and the best approximation algorithms achieve approximation ratios of $\Theta(n)$.

References

1. S. Arora, C. Lund; *Hardness of Approximations*; in: Approximation Algorithms for NP-Hard Problems (ed. Dorit Hochbaum), PWS Publishing Company, pp. 399 - 446, 1996.
2. P. Crescenzi, V. Kann; *A Compendium of NP Optimization Problems*; in the book by G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, *Complexity and Approximation. Combinatorial Optimization Problems and their Approximability Properties*, Springer-Verlag, Berlin, 1999; also available in an online-version at: http://www.nada.kth.se/theory/compendium/compendium.html.
3. S. Eidenbenz, C. Stamm, and P. Widmayer; *Inapproximability of some Art Gallery Problems*; Proc. 10th Canadian Conf. on Computational Geometry, pp. 64 - 65, 1998.
4. S. Eidenbenz, C. Stamm, and P. Widmayer; *Inapproximability Results for Guarding Polygons and Terrains*; Algorithmica, Vol. 31, pp. 79 - 113, 2001.
5. S. Eidenbenz; *Inapproximability of Finding Maximum Hidden Sets on Polygons and Terrains*; Computational Geometry: Theory and Applications (CGTA), Vol. 21, pp. 139 - 153, 2002.
6. S. Eidenbenz; *Approximation Algorithms for Terrain Guarding*; Information Processing Letters (IPL), Vol. 82, pp. 99 - 105, 2002.
7. S. Eidenbenz; *How Many People Can Hide in a Terrain?*; Lecture Notes in Computer Science, Vol. 1741 (ISAAC'99), pp. 184 - 194, 1999.
8. S. Ghosh; *Approximation Algorithms for Art Gallery Problems*; Proc. of the Canadian Information Processing Society Congress, 1987.
9. D. T. Lee and A. K. Lin; *Computational Complexity of Art Gallery Problems*; IEEE Trans. Info. Th, pp. 276 - 282, IT-32, 1986.
10. M. van Kreveld; *Digital Elevation Models and TIN Algorithms*; in: Algorithmic Foundations of Geographic Information Systems (ed. van Kreveld et al.), LNCS tutorial vol. 1340, pp. 37 - 78, Springer, 1997.
11. T. Shermer; *Hiding People in Polygons*; Computing 42, pp. 109 - 131, 1989.
12. T. Shermer; *Recent results in Art Galleries*; Proc. of the IEEE, 1992.
13. J. Urrutia; *Art Gallery and Illumination Problems*; in: Handbook on Computational Geometry, J.R. Sack, J. Urrutia (eds.), North Holland, pp. 973 - 1127, 2000.

Simplex Range Searching and k Nearest Neighbors of a Line Segment in 2D

Partha P. Goswami[1], Sandip Das[2], and Subhas C. Nandy[2]

[1] Computer Center, Calcutta University, Calcutta 700 009, India
[2] Indian Statistical Institute, Calcutta 700 035, India

Abstract. We present an efficient algorithm for finding k nearest neighbours of a query line segment among a set of points distributed arbitrarily on a two dimensional plane. For solving the above, we improved simplex range searching technique in 2D. Given a set of n points, we preprocess them to create a data structure using $O(n^2)$ time and space, which reports the number of points inside a query triangular region Δ in $O(\log n)$ time. The members of P inside Δ can be reported in $O(\log^2 n + \kappa)$ time, where κ is the size of the output. Finally, this technique is used to find k nearest neighbors of a query straight line segment in $O(\log^2 n + k)$ time.

1 Introduction

Given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points arbitrarily distributed on a plane, we study the problem of finding k nearest neighbors of a query line segment σ. On the way of studying this problem, we developed an improved algorithm for the *simplex range searching*, where the objective is to report the subset of points in P that lie inside a query triangle.

A simplex in $\mathcal{R}^d$ is a space bounded by $d+1$ hyperplanes. In the simplex range query problem, a set of points P (in $\mathcal{R}^d$) is given; the objective is to report the number/subset of points which lie inside a simplex query region. We shall refer these two problems as *counting query problem* and *subset reporting problem* respectively. The simplex range search problem was studied extensively [2,6,11]. In $\mathcal{R}^2$, the best result is obtained by Matousek [11]; the preprocessing time and space are $O(n^2 \log^{\epsilon} n)$ and $O(n^2)$ respectively (ϵ is a fixed positive constant), and the counting query takes $O(\log^3 n)$ time. In [6], a quasi-optimal upper bound for the time complexity of simplex range searching problem is presented. The algorithm in [6] can achive $O(\log n)$ time for the *counting query* at an expense of $O(n^{2+\epsilon})$ storage and preprocessing time. For both of these algorithms, the subset reporting problem needs an additional $O(\kappa)$ time, where κ is the size of the output. In [6], the authors agreed that the result in [11] is actually better since the n^{ϵ} factor in the storage is replaced by a polylogarithmic factor in the preprocessing and query time. We improve the *counting query* time to $O(\log n)$ reducing the preprocessing time and space complexities to $O(n^2)$. The subset reporting query requires $(\log^2 n + \kappa)$ time.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 69–79, 2002.

The problem of computing the nearest neighbor of a query line was initially addressed in [7,10]. An algorithm of preprocessing time and space $O(n^2)$ was proposed in that paper which can answer the query in $O(\log n)$ time. In [12], the problem of finding the k nearest/farthest neighbors of a line is proposed along with an algorithm. The preprocessing time and space complexities are both of $O(n^2)$, and the query time complexity is $O(k + \log n)$. The proximity queries regarding line segments are studied very little. The first work appeared in [3] which addresses few restricted cases of the problem of locating the nearest point of a query straight line segment among a set of n points on the plane.

We consider an unrestricted version of the nearest neighbor query problem. The objective is to report the nearest neighbor of an arbitrary query line segment σ among a set of points P. We show that the preprocessed data structure for the simplex range searching problem can answer this query in $O(\log^2 n)$ time. We also show that the following queries can also be answered using our method:

Segment dragging query: Report the first k points (of P) hit by the query line segment σ if σ is dragged in its perpendicular direction. This needs $O(k+\log^2 n)$ time. The preprocessing time and space complexities are $O(n^2)$.

k-nearest neighbors query: This problem has two phases: (i) find k nearest neighbors of the interior of σ, and (ii) find k neraest neighbors of each end point of σ. The first phase can be solved using *segment dragging query* technique. The second phase needs to use order-k Voronoi diagram, provided k is known prior to the preprocessing.

2 Preliminaries

We use geometric duality for solving the problems mentioned in this paper. Here (i) a point $q = (a, b)$ of the primal plane is mapped to the line q^*: $y = ax - b$ in the dual plane, and (ii) a non-vertical line ℓ: $y = cx - d$ of the primal plane is mapped to the point $\ell^* = (c, d)$ in the dual plane. A point q is below (resp., on, above) a line ℓ in the primal plane if and only if the line q^* is above (resp., on, below) the point ℓ^* in the dual plane.

Let Q be a set of m points distributed arbitrarily on a 2D plane, and Q^* be the set of dual lines corresponding to the points in Q. $\mathcal{A}(Q^*)$ denotes the arrangement of the lines in Q^*. As a preprocessing of half-plane range query problem, we construct a data structure for storing the levels of the arrangement $\mathcal{A}(Q^*)$ as defined below. From now onwards, we refer this data structure as *level-structure*.

Definition 1. [8] A point q in the dual plane is at level θ $(0 \leq \theta \leq m)$ if there are exactly θ lines in Q^* that lie strictly below q. The θ-level of $\mathcal{A}(Q^*)$ is the closure of a set of points on the lines of Q^* whose levels are exactly θ in $\mathcal{A}(Q^*)$, and is denoted as λ_θ.

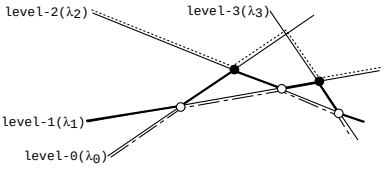


Fig. 1. Demonstration of levels in an arrangement of lines

Clearly, the edges of λ_θ form a monotone polychain from $x = -\infty$ to $x = \infty$. Each vertex of the arrangement $\mathcal{A}(Q^*)$ appears in two consecutive levels, and each edge of $\mathcal{A}(Q^*)$ appears in exactly one level (see Fig. 1).

Definition 2. The *level-structure* is an array $\mathcal{A}$ whose elements correspond to the levels $\{\theta \mid \theta = 1, \ldots, m\}$ of the arrangement $\mathcal{A}(Q^*)$. Each element representing a level θ, is attached with a linear array containing the vertices of λ_θ in a left to right order.

In order to reduce the query time complexity, an augmentation procedure of the *level-structure* is described in [12]. It creates few more vertices and edges by projecting the existing vertices on some existing edges of $\mathcal{A}(Q^*)$. Then it attaches a pair of pointers with each edge. The following theorem abstracts the complexity results of the *half-plane range queries*, using augmented level structure.

Theorem 1. *Given a set of n points in the plane, it can be preprocessed in $O(n^2)$ time and $O(n^2)$ space such that (i) the half-plane counting query can be answered in $O(\log n)$ time, and (ii) the half-plane subset reporting can be performed in $O(\kappa + \log n)$ time, where κ is the size of the output.*

3 Simplex Range Searching

A simplex in 2D is a triangular region obtained as the intersection of three halfplanes, each of them is defined by a straight line. Given a triangular range Δ, our objective is to report the points of P that lie inside Δ. We shall consider both *counting query* and *subset reporting query* for the triangular range searching problem separately.

3.1 Preprocessing

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points arbitrarily distributed in 2D plane. A line splitting the set P into two non-empty subsets is called a *cut*. A cut is said to be *balanced* if it splits P into two subsets P_1 and P_2 such that the size of these two subsets differ by at most one. For a given point set P, the balanced cut

may not be unique. But we may choose any one of them. The point sets P_1 and P_2 are further divided recursively using balanced cuts. This process continues until all the partitions contain exactly one element.

As a preprocessing for the simplex range query, we create a data structure $T(P)$, called *partition tree*, based on the hierarchical balanced bipartitioning of the point set P. Its root node (at layer-0) corresponds to the set P. The cut I_0 splits P into two subsets P_1 and P_2. Thus, the two successors of the root (at layer-1) correspond to the point sets P_1 and P_2 respectively. P_1 and P_2 are further partitioned using balanced cuts to create 4 nodes at layer-2. The splitting continues and the nodes of the tree are defined recursively in this manner.

Each node v of $T(P)$ is attached with (i) the set of points P_v attached to node v, (ii) an integer field χ_v indicating the size of P_v, and (iii) the balanced cut line I_v bipartitioning the points P_v.

Given a set of points P, we uniquely define the partition tree $T(P)$ using *ham-sandwich cuts* [9]. $T(P)$ can be constructed in $O(n\log n)$ time and space.

At each non-leaf node of the partition tree $T(P)$, we attach two secondary structures, namely SS_1 and SS_2. SS_1 is a *level structure* with the dual lines of the points in P_v. It is useful for the simplex range counting query, but is not efficient for subset reporting. SS_2 is created with the same point set P_v in the primal plane, and is used for reporting the members inside the query triangle. It's performence with respect to the counting query is inferior to SS_1.

Secondary structure - SS_1

Consider the node v in $T(P)$. P_v^* is the set of lines corresponding to the duals of the points in P_v. We create the augmented level structure $\mathcal{A}(P_v^*)$ (as defined in Section 2) , and attach it with node v. We refer this secondary structure as $SS_1(v)$. We further augment the data structure using the following lemma. This accelarates the simplex range counting query.

Lemma 1. *Let v be a node in $T(P)$, and w be a successor of node v in $T(P)$. A cell in $\mathcal{A}(P_v^*)$ is completely contained in exactly one cell of $\mathcal{A}(P_w^*)$.*

Proof: Let u be the other successor of node v. P_u^* and P_w^* are the duals of the points attached to nodes u and w. Let C be a cell in $\mathcal{A}(P_v^*)$. It is bounded by the lines of both P_u^* and P_w^*. If the lines of P_u^* are removed, the cell C will be contained in a cell of the arrangement $\mathcal{A}(P_w^*)$. □

With each cell $\mathcal{C} \in \mathcal{A}(P_v^*)$, we attach two pointers, $cell\_ptr_L$ and $cell\_ptr_R$. They point to the cells $\mathcal{C}_L \in \mathcal{A}(P_u^*)$ and $\mathcal{C}_R \in \mathcal{A}(P_w^*)$ respectively in which the cell $\mathcal{C}$ is contained. Basically, this can be done by attaching the pointers with each edge of $e \in \mathcal{A}(P_v^*)$. If the edge e is a part of an edge $e^* \in \mathcal{C}_L$, then its $cell\_ptr_L$ points to e^*. We draw a vertical line at the left end point of e in downward direction. Let it hits the edge $e^{**} \in \mathcal{C}_R$. The $cell\_ptr_{right}$ points to e^{**}. If e is a part of an edge in $\mathcal{C}_R$, the $cell\_ptr_R$ and $cell\_ptr_L$ are set in a similar manner.

Lemma 2. *The time and space required for creating and storing the preprocessed data structure SS_1 for all non-leaf nodes in $T(P)$ are both $O(n^2)$.*

Proof: The initial partition tree can be constructed in $O(n\log n)$ time and space. The number of nodes at level i of $T(P)$ is 2^i, and each of them contains $\frac{n}{2^i}$ points, $i = 0, 1, \ldots, \log n - 1$. For each non-leaf node at level i, the size of SS_1 structure is $O((\frac{n}{2^i})^2)$, and it can be constructed from the point set assigned to that node in $O((\frac{n}{2^i})^2)$ time. So, the total time and space required for constructing the SS_1 data structure for all nodes in $T(P)$ is $O(\sum_{i=0}^{\log n-1} 2^i \times (\frac{n}{2^i})^2) = O(n^2)$.

Finally, we use topological line sweep to set $cell\_ptr_L$ and $cell\_ptr_R$ attached to each edge of $\mathcal{A}(P_v^*)$. This requires an additional $O(n^2)$ amount of time. □

Secondary structure - SS_2

This is another secondary structure attached to each non-leaf node of $T(P)$. It is created with the points attached to each node in the primal plane.

Consider a non-root node v at i-th layer of the tree $T(P)$. The region attached to it is R_v, and the set of points attached to it is P_v. Note that, R_v is a convex polygonal region whose boundaries are defined by the cut lines of its predecessors, i.e., all the nodes on the path from the root of $T(P)$ upto the parent of the current node. Thus, if v is at layer-i, the number of sides of the region R_v is at most i. We store the boundary edges of R_v in an array. Each edge I is attached with the following data structure.

Let π be a point on an edge I of the boundary of R_v. A half-line is rotated around π inside the region R_v by an amount 180^o, and a list L_π is formed with the points in P_v ordered with respect to their appearance during the rotation. For each point of I we get such a list. Note that, we may get an interval on I around the point π such that for all points inside this interval, the list remains same. In order to get these intervals, we join each pair of points in P_v by a straight line. These lines are extended in both sides up to the boundary of R_v. This creates at most $O(|P_v|^2)$ intervals along the boundary of R_v. If we consider any two consecutive intervals on an edge I of the boundary of R_v, the circular order of points only differ by a swap of two members in their respective lists. This indicates that $O(|P_v|^2)$ space is enough to store the circular order of the points of P_v for all the intervals on I. Indeed, we can use the data structure proposed in [5] for storing almost similar lists for this purpose. For the details about this data structure, see [5,7].

Lemma 3. *The time and space required for creating and storing the SS_2 data structure for all nodes in $T(P)$ is $O(n^2)$.*

Proof: Each point appears in exactly one region in each layer of $T(P)$. But a point p inside a region R_v (corresponding to a node v) appears in the data structure attached to all the edges of R_v. For a node v at the i-th layer of $T(P)$, its attached R_v is bounded by at most i cut-lines, and contains $\frac{n}{2^i}$ points. Thus the total space required to store the data structure for each edge I on the

boundary of R_v is at most $i \times (\frac{n}{2^i})^2$. Again, the number of nodes in the i-th layer is 2^i. Thus, the time and space required for creating and storing the SS_2 data structure for all nodes in $T(P)$ in the worst case is
$= 1.2^1.(\frac{n}{2})^2 + 2.2^2.(\frac{n}{2^2})^2 + 3.2^3.(\frac{n}{2^3})^2 + \ldots + \log n.2^{\log n}.(\frac{n}{2^{\log n}})^2$
$= \frac{1}{2}.n^2 + \frac{2}{2^2}.n^2 + \frac{3}{2^3}.n^2 + \ldots + \frac{\log n}{2^{\log n}}.n^2 = O(n^2)$. □

In the next section, we discuss two types of queries, namely (i) counting query, and (ii) subset reporting query separately for a triangular query region Δ.

3.2 Counting Query

Here the objective is to report the number of points of P that lie inside a triangular region Δ. We traverse the preprocessed data structure $T(P)$ from its root with the query region Δ. A global *COUNT* field (initialized with 0) is maintained during the traversal.

During the traversal, if a leaf node is reached with a query region Δ^* ($\in \Delta$), the *COUNT* is incremented by one if the point attached to that node lies inside Δ^*. While processing a non-leaf node v with a query region Δ^*, its attached partition line I_v may or may not split Δ^*. In the former case, v is said to be a *split node*, and in the latter case v is said to be a *non-split node*.

At a *non-split node* v, the traversal proceeds towards one child of v whose corresponding partition contains Δ^*. On the other hand, at a *split node*, Δ^* splits into two query regions; each of them is any one of the following types.

type-0 **:** A region having no corner of Δ,
type-1 **:** A region having one corner of Δ, and
type-2 **:** A region having two corners of Δ.

Lemma 4. *All types of regions obtained by successive splitting of Δ, are convex.*

When Δ splits for the first time, it gives birth to one *type-1* and one *type-2* regions. In the successive splits,

- a *type-2* region may either split into (i) one *type-0* region and one *type-2* region, or (ii) two *type-1* regions. In case (i), the *counting query* inside the *type-0* region is performed among the points in the partition attached to one successor of v, and the traversal proceeds towards the other successor of v containing the *type-2* region. In case (ii), traversal proceeds towards both the successors of v with the corresponding *type-1* region, in recursive manner.
- A *type-1* region splits into one *type-0* and one *type-1* region. The processing of *type-0* region is permormed at one successor of node v. The traversal proceeds towards the other successor with the *type-1* region.

The processing of a *type-0* region at a node v is described below.

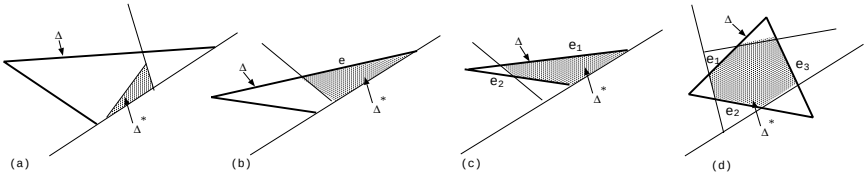


Fig. 2. Different possibilities of *type-0* region

Lemma 5. *The number of type-0 regions generated during the entire traversal with the region Δ may be at most $O(\log n)$.*

Counting Query Inside a Type-0 Region

Lemma 6. *The number of edges of Δ that appear on the boundary of a type-0 region may be at most three.*

Let Δ^* be a *type-0* region, and the *counting query* with respect to Δ^* need to be processed at node v of $T(P)$. We now consider the following four cases depending on the number of sides of Δ that appear on the boundary of Δ^*.

Case 1: Δ^* is not bounded by any edge of Δ (see Fig. 2(a)). Here all the $\chi_v (= |P_v|)$ points lie inside Δ^*.

Case 2: Exactly one edge e of Δ appears on the boundary of Δ^* (see Fig. 2(b)). The edge e cuts the boundary of R_v in exactly two points, and it splits the point set P_v into two disjoint subsets. One of these subsets lies completely outside Δ^*, and the other one completely lies inside Δ^*. The number of points of P_v lying inside Δ^* can be obtained by performing *half-plane range counting query* among the point set P_v with respect to the line containing e.

Case 3: Exactly two edges of Δ appear on the boundary of Δ^* (see Fig. 2(c)). These two edges are mutually non-intersecting inside R_v, and each of them intersects the boundary of R_v in exactly two points. Let these two edges be e_i, $i = 1, 2$. As stated earlier, each of these edges (e_i) partitions the point set P_v into two disjoint subsets, say $P_v(e_i)$ and $\overline{P_v(e_i)}$. $\overline{P_v(e_i)}$ lies completely outside Δ^*, but $P_v(e_i)$ may not completely lie inside Δ^* due to the constraint imposed by the other member $e_j, j \neq i$.

The number of points of P_v inside Δ^* is equal to $|P_v(e_1) \cap P_v(e_2)| = (\chi_v - |\overline{P_v(e_1)}| - |\overline{P_v(e_2)}|)$, and it can be obtained by performing the *half-plane range counting query* with the lines e_1 and e_2 separately.

Case 4: Exactly three edges of Δ appear on the boundary of Δ^* (see Fig. 2(d)). As in Case 3, all these edges are mutually non-intersecting inside R_v, and each of them intersects the boundary of R_v in exactly two points. The number of points of P_v inside Δ^* is equal to $(\chi_v - \sum_{i=1}^{3} |\overline{P_v(e_i)}|)$, where $P_v(e_i)$ is defined as in Case 3. Thus, to report the number of points inside Δ^*, we need to perform *half-plane range counting query* among the points in P_v at most three times.

Time Complexity of Counting Query

The simplex range counting query starts from the root of $T(P)$ and with $COUNT$ equal to zero. Let ℓ_1^*, ℓ_2^* and ℓ_3^* be the duals of the lines containing the three edges e_1, e_2 and e_3 of Δ. We find the cells containing ℓ_1^*, ℓ_2^* and ℓ_3^* in the SS_1 data structure attached to the root node of $T(P)$. During traversal, when search moves from a node v to its children, the cells corresponding to ℓ_1^*, ℓ_2^* and ℓ_3^* in the secondary structure of the children of v are reached using $cell\_ptr_L$ and $cell\_ptr_R$ in $O(1)$ time. At each node on the traversal path, if a *type-0* region is generated, the number of points inside that region is computed, and added with $COUNT$. If a leaf node is reached, the point attached with it is tested to check whether it lies inside Δ in $O(1)$ time. If so, $COUNT$ field is incremented by 1. At the end of traversal, $COUNT$ field indicates the number of points inside Δ.

Theorem 2. *Given a set of n points we can preprocess them in $O(n^2)$ time and space such that the number of points inside a query triangle can be obtained in $O(\log n)$ time.*

Proof: The preprocessing time and space complexity results follow from Lemma 2. During query, $O(\log n)$ time is spent to locate the cells containing ℓ_1^*, ℓ_2^* and ℓ_3^* in the SS_1 data structure attached to the root node. From the next layer of $T(P)$ onwards, the desired cells in the SS_1 structure of each node on the traversal path are reached in $O(1)$ time as mentioned earlier. In each cell containing a *type-0* region, the number of points outside the query region are obtained from the level information attached to the edges of Δ crossing that cell, which may be at most 3 (see Lemma 6). This takes $O(1)$ time. The result follows from the fact that $O(\log n)$ nodes need to be visited during the traversal (see Lemma 5). □

The drawback of using SS_1 secondary structure is that, it is not efficient in reporting the set of points inside Δ. The worst case time complexity of subset reporting query may be $O(\kappa \log n)$, where κ is the number of points inside Δ.

3.3 Subset Reporting Query

The subset reporting for a triangular query region Δ is also done by traversing $T(P)$. At each split node, if the query region is either *type-1* or *type-2*, it splits in a similar manner as described in the counting query. While processing a *type-0* region at a particular node, either of the four cases, as mentioned earlier, may appear. The processing of those cases is described below.

- In case 1, all the points in P_v are reported.
- In case 2, the subset of points in P_v lying in one side of the edge e (of Δ) are reported. These can be easily obtained from SS_1 data structure itself.
- In case 3, the query region at node v is bounded by two edges, say e_1 and e_2, of Δ. We use the secondary structure SS_2 for the reporting in this case. In Fig. 3, edge e_1 (resp. e_2) intersects R_v at α_1 and α_2 (resp. β_1 and β_2).

We split the query region by the diagonal $\alpha_1\beta_2$ (indicated by dotted line). Let the points α_1 and β_2 lie on the edges I and J of R_v respectively. We use binary search with the point α_1 (resp. β_2) to locate its corresponding interval on the edge I (resp. J). Next we use the data structure attached to I (resp. J) to report the points inside the darkly (resp. lightly) shaded angular region. The detailed method is described in [5]. The time complexity of this reporting is $(\log n + \kappa)$, where κ is the size of the output.

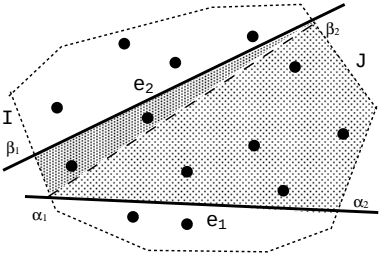


Fig. 3. Reporting in case 3

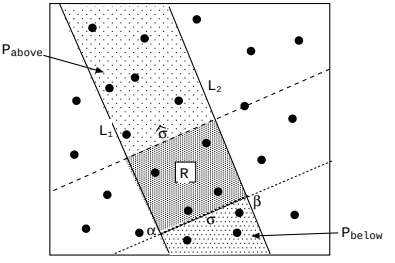


Fig. 4. Segment dragging query

- In case 4, the query region at node v is bounded by three edges e_1, e_2 and e_3 of Δ. Here we need to proceed the traversal to the successor(s) of v in $T(P)$. If the query region splits at v, it may generate at most one query region which is bounded by three edges of Δ. Traversal proceeds with this part. The other part of the split is bounded by either one or two edges of Δ. The points inside this part are reported as in Case 1 or Case 2 or Case 3. This type of split takes place at most $(\log n)$ time. So the reporting time in this case may be $O(\log^2 n + \kappa)$.

Theorem 3. *Given a set of n points we can preprocess them in $O(n^2)$ time and space such that the subset of points inside a query triangle can be reported in $O(\log^2 n + k)$ time.*

4 Segment Dragging Query

We shall use the preprocessed data structure discussed in the earlier section for solving the segment dragging query with respect to a line segment $\sigma = [\alpha, \beta]$.

Let us consider a corridor $\mathcal{C}_\sigma$ defined by two parallel lines L_1 and L_2 drawn through the points α and β respectively, and each of them is perpendicular to σ. The set of points inside the corridor is split into two subsets P_{above} and P_{below} by the segment σ. In the segment dragging query, we need to report k nearest points of σ among the members in P_{above} (P_{below}). Here k may be specified at the query time.

Consider the levels of arrangement $\mathcal{A}(P^*)$. Let ℓ_σ denotes the line containing the segment σ. Let ℓ_σ^* (the dual of ℓ_σ) lies between levels λ and $\lambda+1$ in the dual

plane. If $\lambda < k$, then σ hits no more than $\lambda(< k)$ points if it is dragged above up to infinity. So, all the points in P_{above} need to be reported. If $\lambda > k$, we need to find a line segment $\hat{\sigma}$ parallel to σ and touching the two boundaries of the corridor $\mathcal{C}_\sigma$ such that the number of points inside the region R defined by L_1, L_2, σ and $\hat{\sigma}$ (as shown in Fig. 4) is equal to k (excepting the degenerate cases). Thus, here the segment dragging query consists of two phases: (i) compute $\hat{\sigma}$ appropriately, and (ii) report the points inside R.

We solve the first part as follows: draw a vertical ray from the point ℓ^*_σ downwards. Let $e \in \mathcal{A}(P^*)$ be an edge at level θ $(\theta < \lambda - k)$ whom the ray hits. Let p^* $(p \in P)$ be the line containing the edge e. We draw a line parallel to σ at the point p. This defines a rectangle R_θ bounded by two boundaries of $\mathcal{C}_\sigma$, the given query segment σ, and the portion of the line p^* inside the corridor $\mathcal{C}_\sigma$. Let κ_θ denotes the number of points inside R_θ. We compute κ_θ by splitting R_θ into two triangles, and then applying the triangular range counting method as described in Section 3.

Lemma 7. *Let e_i and e_2 be two edges at level i and j, $i < j$. R_i and R_j denote two rectangles as defined above. If κ_i and κ_j denote the number of points inside R_i and R_j respectively, then $\kappa_i > \kappa_j$.*

Proof: Follows from the fact that the area of R_i is greater than that of R_j, but the three sides of R_i are common with those of R_j. □

Lemma 8. *A rectangle R above the query line segment σ and containing exactly k points can be obtained in $O(\log^2 n)$ time.*

Proof: We consider the subset of edges of $\mathcal{A}(P^*)$ that are hit by the vertically upward ray shot from ℓ^*_σ. These edges are at different levels of $\mathcal{A}(P^*)$. By Lemma 7, the desired $\hat{\sigma}$ can be selected using binary search among these edges. At each step of the binary search, (i) we choose an edge from the above subset and define the corresponding rectangle inside the corridor C_σ (in the primal plane), and then (ii) we need to find the number of points inside that rectangle by applying triangular range counting query. Hence the lemma follows. □

Next, using subset reporting query algorithm, we report the points inside the rectangle R. Thus, we have the main result of this work.

Theorem 4. *Given a set of n points in 2D, it can be preprocessed in $O(n^2)$ time and space such that for an arbitrary query segment, the dragging query can be answered in $O(k + \log^2 n)$ time, where k is an input at the query time.*

5 k Nearest Neighbors of σ

The problem of finding k nearest neighbors of $\sigma = [\alpha, \beta]$ (k is known apriori) has two phases: (i) find k nearest neighbors of α and β using order-k Voronoi

diagram, and (ii) solve segment dragging query with parameter k for both above and below σ. Finally, a merge like pass is executed to report k nearest neighbors of the line segment σ. Thus, the time complexity of creating an order-k Voronoi diagram influences the preprocessing time complexity for finding k nearest neighbors query of a line segment. The time complexity of the best known deterministic and randomized algorithms for creating an order-k Voronoi diagram are $O(nk\log^2 k(\frac{\log n}{\log k})^{O(1)})$ [4] and $O(n\log^3 n + k(n-k))$ [1] respectively. The storage and query time complexities for the k nearest neighbors problem remain same as that of the segment dragging query problem.

Final remark: In case of finding the nearest neighbor of a query line segment σ (i.e., when $k = 1$), the preprocessing time and space complexities are both $O(n^2)$, and query can be answered in $O(\log^2 n)$ time. This is a generalized and improved result of the problem presented in [3]. Here SS_1 data structure supports the segment dragging query; so SS_2 is not needed.

References

1. P. K. Agarwal, M. de Berg, J. Matousek, and O. Schwartzkopf, *Constructing levels in arrangement and higher order Voronoi diagram*, SIAM J. Computing, vol. 27, pp. 654-667, 1998.
2. P. K. Agarwal and J. Matousek, *Dynamic half-space range reporting and its applications*, Algorithmica, vol. 13, pp. 325-345, 1995.
3. S. Bespamyatnikh and J. Snoeyink, *Queries with segments in Voronoi diagram*, Computational Geometry - Theory and Applications, vol. 16, pp. 23-33, 2000.
4. T. M. Chan, *Random sampling, halfspace range reporting, and construction of* $(\leq k)$*-levels in three dimension*, SIAM J. Computing, vol. 30, pp. 561-575, 2000.
5. R. Cole, *Searching and storing similar lists*, Journal of Algorithms, vol. 7, pp. 202-230, 1986.
6. B. Chazelle, M. Sharir and E. Welzl, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, Algorithmica, vol. 8, pp. 407-429, 1992.
7. R. Cole and C. K. Yap, *Geometric retrieval problems*, Information and Control, pp. 39-57, 1984.
8. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer, Berlin, 1987.
9. H.Edelsbrunner and E. Welzl, *Halfplanar range search in linear space and* $O(n^{.695})$ *query time*, Information Processing Letters, vol. 23, pp. 289-293, 1986.
10. D. T. Lee and Y. T. Ching, *The power of geometric duality revisited*, Information Processing Letters, vol. 21, pp. 117-122, 1985.
11. J. Matousek, *Range searching with efficient hierarchical cutting*, Proc. 8th. Annual Symp. on Computational Geometry, pp. 276-285, 1992.
12. S. C. Nandy, *An efficient k-nearest neighbors searching algorithm for a query line*, Proc. 6th. Annual Int. Conf. on Computing and Combinatorics, LNCS-1858, pp. 291-298, 2000.

Adaptive Algorithms for Constructing Convex Hulls and Triangulations of Polygonal Chains

Christos Levcopoulos[1], Andrzej Lingas[1], and Joseph S.B. Mitchell[2⋆]

[1] Dept. of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden
{Christos.Levcopoulos,Andrzej.Lingas}@cs.lth.se
[2] Dept. of Applied Mathematics and Statistics, SUNY, Stony Brook, NY 11794, USA
jsbm@ams.sunysb.edu

Abstract. We study some fundamental computational geometry problems with the goal to exploit structure in input data that is given as a sequence $C = (p_1, p_2, \ldots, p_n)$ of points that are "almost sorted" in the sense that the polygonal chain they define has a possibly small number, k, of self-intersections, or the chain can be partitioned into a small number, χ, of simple subchains. We give results that show adaptive complexity in terms of k or χ: when k or χ is small compared to n, we achieve time bounds that approach the linear-time ($O(n)$) bounds known for the corresponding problems on simple polygonal chains. In particular, we show that the convex hull of C can be computed in $O(n \log(\chi+2))$ time, and prove a matching lower bound of $\Omega(n \log(\chi + 2))$ in the algebraic decision tree model. We also prove a lower bound of $\Omega(n \log(k/n))$ for $k > n$ in the algebraic decision tree model; since $\chi \leq k$, the upper bound of $O(n \log(k + 2))$ follows.
We also show that a polygonal chain with k proper intersections can be transformed into a polygonal chain without proper intersections by adding at most $2k$ new vertices in time $O(n \cdot \min\{\sqrt{k}, \log n\} + k)$. This yields $O(n \cdot \min\{\sqrt{k}, \log n\} + k)$-time algorithms for triangulation, in particular the constrained Delaunay triangulation of a polygonal chain where the proper intersection points are also regarded as vertices.

1 Introduction

A *polygonal chain* in the Euclidean plane is specified by a finite sequence $C = (p_1, p_2, \ldots, p_n)$ of points (*vertices*), which define the chain's edges, p_1p_2, p_2p_3, …, $p_{n-1}p_n$, and possibly p_np_1 (if the chain is *closed*). The chain is *strongly simple* if any two edges, p_ip_{i+1} and p_jp_{j+1}, of C that are not adjacent ($i \neq j$) are disjoint and any two adjacent edges share only their one common vertex. We say that C is *simple* if it is not self-crossing but it is possibly self-touching, with a vertex falling on a non-incident edge or on another vertex; i.e., it is simple if it is strongly simple or it has an infinitesimal perturbation that is strongly simple. If the chain is not strongly simple, but it is simple, we say that it is

⋆ Partially supported by HRL Labs (DARPA subcontract), NASA Ames Research, NSF (CCR-0098172), U.S.-Israel Binational Science Foundation.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 80–89, 2002.

weakly simple. A *crossing witness* for a polygonal chain is a pair of its edges that intersect properly at a *witness point*.

In this paper we consider the problem of efficiently constructing the convex hull of the vertices of a polygonal chain. If the input polygonal chain is simple, in particular only weakly simple, these basic geometric structures, and many others, can be computed in linear time [4,6,7]. If the input chain is arbitrary (with no special structure), then computing the convex hull of C is known to have an $\Omega(n \log n)$ lower bound.

A natural approach to these problems is to compute all k self-crossings of the chain C, cut the chain into subchains at the crossing points, apply known methods (e.g., to compute convex hulls) to each of the resulting subchains, and merge the results. The problem with this approach is that computing all k of the intersections seems too costly. The best known method for reporting segment intersections among an unordered set of n line segments takes time $O(n \log n + k)$ and space $O(n)$ [3], and this is optimal, in general. Since our goal is to replace $O(n \log n)$ solutions for convex hulls and triangulations with $O(n)$ solutions in the case of small k, the $O(n \log n)$ overhead for computing the intersections is already too much for us to spend. There is speculation that if the n segments are given in order as the edges along a chain, as in our problem, then the intersection points can be computed in time $O(n + k)$; this, however, remains a challenging open problem. Also, even if this problem is solved in time $O(n + k)$, the running time is potentially too high when k is super-linear in n.

Thus, the fundamental question we address here is: Can we exploit the fact that the n points or segments are given in an order along a chain, to avoid the $O(n \log n)$ sorting overhead, even when the chain is not simple?

We measure the degree of non-simplicity in terms of two parameters: (1) the number, k, of self-intersections, and (2) the *simple partition number*, χ, defined to be the minimum number of partitioning points at which we need to cut the (open) chain C in order to partition C into simple subchains. If C is simple, then $k = \chi = 0$. Note that $\chi \leq k$, since one can partition C into simple subchains by cutting it at the points of self-intersection.

Our goal is to have a time complexity that matches the simple chain case when k or χ is equal to zero (or a constant), and never goes above the time complexity for the unordered case, in which C is given in arbitrary order, even when k or χ are large (e.g., when k is close to its maximum possible value of $(n-2)(n-3)/2$, or when χ is close to its maximum, $\lceil (n-1)/2 \rceil$).

Questions of the sort "Does simplicity help?" are prominent in computational geometry. Usually the answer is "Yes" – e.g., the convex hull or a triangulation of a simple polygon is found in $O(n)$ time, while, in general, both computations require time $\Omega(n \log n)$ for an unordered set of points. The question we study here is whether we can exploit "approximate simplicity" in designing algorithms whose time complexity is adaptive in the parameter k or χ.

A polygonal chain with a moderate number of self-intersections or a small simple partition number is the geometric analogue to a partially sorted file in the area of sorting algorithms. There exists a vast literature on adaptive algorithms

for sorting partially sorted files and measures of presortedness (e.g., see [10]). There is also work of Aurenhammer [1,2] that studies the problem of computing the convex hull, in linear time, of a special class of self-intersecting polygonal chains that arises in the context of Jordan sorting [8] and on-line sorting of "twisted sequences".

Chazelle [4] shows that his linear-time polygon triangulation algorithm can be used for simplicity testing (determining if $k = \chi = 0$) in linear time; essentially, he says that the triangulation algorithm "fails" in a controlled way on a non-simple input. Chazelle has also observed [5] that, with some additional work, his algorithm can be made to report a crossing witness if the chain is non-simple.

Our first result (Section 2) in this paper is a linear-time method to compute a crossing witness (if one exists), relying solely on a "black box" verifier for polygonal chain weak simplicity. By combining this method of crossing detection with the optimal method for segment intersection [3], we show how a polygonal chain with k proper intersections can be transformed into a weakly simple polygonal chain by adding $2k$ vertices in time $O(n \cdot \min\{\sqrt{k}, \log n\} + k)$. Since the aforementioned linear-time methods work even for weakly simple chains [4,6,7], we obtain $O(n \cdot \min\{\sqrt{k}, \log n\} + k)$-time algorithms for the convex hull of C and for triangulation of C, in particular for the constrained Delaunay triangulation, where the proper intersection points are also regarded as vertices.

Then, in Section 3 we present a more efficient approach to computing the convex hull of a polygonal chain, relying solely on a linear-time weak simplicity test for a polygonal chain, without the crossing witness requirement. This approach enables us to show that the convex hull of a polygonal chain with simple partition number χ can be computed in time $O(n \log(\chi + 2))$. The same approach is applied to sorting the intersection points $\ell \cap C$ along a given line ℓ in time $O(n \log(\chi + 2))$; this generalizes the Jordan sorting result [8] to self-intersecting chains. While our convex hull result relies on the use of the complicated linear-time algorithm of Chazelle [4], we also show that a very simple algorithm solves the convex hull problem in $O(n)$ time for chains having at most one self-intersection ($k \leq 1$).

We complement our algorithms with a proof of lower bounds in the algebraic decision tree model. In particular, we prove a bound of $\Omega(n \log(\chi + 2))$, showing that our upper bound of $O(n \log(\chi + 2))$ is asymptotically tight. We also prove that $\Omega(n \log k/n)$ is a lower bound on computing the convex hull of a polygonal chain with $k > n$ proper intersections.

2 Crossing Witnesses, the Transformation, and Adaptive Triangulation Algorithms

The following fact is shown in the last section of [4].

Fact 1. *A polygonal chain can be tested for weak simplicity in linear time.*

The auxiliary procedure below will be useful in determining a crossing witness of a polygonal chain via Fact 1.

Procedure $FindInt(C', C'')$

Input: two weakly simple polygonal chains C', C'' which properly intersect each other.
Output: an edge e' of C' and an edge e'' of C'' where e' intersects e'' at a proper intersection point of C' and C''.

1. If $|C'| \leq 1$ or $|C''| \leq 1$ then check the single edge of one of the chains for intersection with all the edges of the other chain, report the pair defining the proper intersection point and stop;
2. $C'_1 \leftarrow$ the prefix of C' of length $\lceil |C'|/2 \rceil$; $C'_2 \leftarrow$ the remaining suffix of C';
3. $C''_1 \leftarrow$ the prefix of C'' of length $\lceil |C''|/2 \rceil$; $C''_2 \leftarrow$ the remaining suffix of C'';
4. **for** $i = 1,\ 2$ **do**
   for $j = 1,\ 2$ **do**
   By adding at most $|C'_i| + |C'_j|$ new vertices connect C'_i with C'_j into a polygonal chain $C_{i,j}$ without introducing new proper intersections;
   If $C_{i,j}$ is not weakly simple then $FindInt(C'_i, C''_j)$ and stop.

Lemma 1. *$FindInt(C_1, C_2)$ can be implemented in time $O(|C_1| + |C_2|)$.*

Proof. To form the chain $C_{i,j}$, we simply double the chains C'_i and C''_j so they become cyclic, and draw a minimal subsegment of the straight-line segment connecting, say the leftmost vertex of C'_i with the leftmost one of C''_j, so the two cycles become connected and no new proper intersections are introduced. By testing the segment for intersection with each edge in both chains, this can be done in time $O(|C'| + |C''|)$. Next, we split the two cycles at the endpoints of the subsegment introducing a double vertex at each of them. Now, it is sufficient to make an Euler tour of the resulting figure, possibly deleting the last edge, in order to obtain the polygonal chain $C_{i,j}$. It follows via Fact 1 that the body of the procedure takes time $O(|C'| + |C''|)$. Hence, the recursive calls take time $O(\sum_i (|C'| + |C''|)/2^i)$, i.e., $O(|C'| + |C''|)$. □

The main recursive procedure for reporting a crossing witness, using $FindInt$ as a subroutine, is as follows.

Procedure $Witness(C)$

Input: a self-intersecting polygonal chain C.
Output: two edges e', e'' of C where e' intersects e'' at the proper self-intersection point of C.

1. $C_1 \leftarrow$ the prefix of C of length $\lceil |C|/2 \rceil$; $C_2 \leftarrow$ the remaining suffix of C;
2. If C_1 and C_2 are weakly simple then $FindInt(C_1, C_2)$ and stop;
3. If C_1 is weakly simple then $Witness(C_2)$ else $Witness(C_1)$

Lemma 2. *$Witness(C)$ runs in time $O(|C|)$.*

Proof. The body of the procedure takes time $O(|C|)$ by Fact 1 and Lemma 1. Hence, the recursive calls take time $O(\sum_i |C|/2^i)$, i.e., $O(|C|)$. □

Since the correctness of the procedure $Witness$ is obvious, we obtain the following result by Lemma 2.

Theorem 1. *A crossing witness for a polygonal chain which is not weakly simple can be found in linear time.*

By iterating the method of Theorem 1 and combining it with the optimal method for segment intersection [3], we obtain the following theorem.

Theorem 2. *A polygonal chain properly intersecting itself k times can be transformed into a weakly simple polygonal chain by adding at most $2k$ new vertices in time $O(n \cdot \min\{\sqrt{k}, \log n\} + k)$.*

Proof. Let C be a polygonal chain properly intersecting itself at most k times. Assume first that k is known.

Case: $k < \log^2 n$. We show in this case how to detect all proper intersections of C and transform C into a weakly simple chain by adding at most $2k$ new vertices, in total time $O(n\sqrt{k}))$. We start by partitioning C into $\lceil\sqrt{k}\rceil$ subchains $C_1, C_2, ..., C_{\lceil\sqrt{k}\rceil}$ of about equal length, plus/minus 1. Next, each C_i is tested in isolation to find all of its (internal) proper intersections, and we add vertices to transform it into a weakly simple chain. This is done by repeatedly applying the procedure $Witness(C_i)$ and introducing, after each application, two new vertices at the detected proper intersection point, while reordering the edges of the resulting chain so this proper intersection is eliminated and no new proper intersections are created. Note that the introduction of these two new vertices and the reordering involve deleting and creating $O(1)$ links and thus take $O(1)$ time.

Since $k < \log^2 n$, the size of each subchain remains $O(n/\sqrt{k})$, even with the introduction of the new vertices. By Lemma 2, each intersection is detected and transformed in time $O(|C_i|) = O(n/\sqrt{k})$, so that the total time for transforming all subchains C_i into weakly simple chains is $O(kn/\sqrt{k})) = O(n\sqrt{k}))$.

Next, for each pair of subchains C_i and C_j, with $1 \leq i < j \leq \lceil\sqrt{k}\rceil$, we repeatedly call the procedure $FindInt(C_i, C_j)$ to detect all proper intersections between the subchains and eliminate them by creating two new vertices for each such intersection and rearranging the edge ordering accordingly. Since we may assume without loss of generality that $k < \log^2 n$, the size of both subchains remains $O(n/\sqrt{k})$ throughout the entire algorithm. Thus, the detection of each consecutive proper intersection by $FindInt(C_i, C_j)$ takes time $O(n/\sqrt{k})$, by Lemma 2. Hence, the total time for finding and transforming all intersections in C is $O(n\sqrt{k}))$.

Case: $k \geq \log^2 n$. In this case, the optimal algorithm for reporting segment intersection, running in time $O(n \log n + k)$-time [3], is more efficient than the partitioning method of the previous case. We simply run this algorithm, and, whenever an intersection is reported, we eliminate it again by creating two new vertices and rearranging the edge ordering in the current chain accordingly.

Finally, if k is unknown we use the standard trick of "doubling": we run our method for values of $k = 1, 2, 4, ...$ until all intersections are detected. Clearly, this does not change the asymptotic upper bound on the running time. □

Since the aforementioned linear-time algorithms ([4,6]) for triangulation, in particular the constrained Delaunay triangulation, of a simple polygonal chain work also for weakly simple polygonal chains, we obtain the following corollary of Theorem 2.

Corollary 1. *A triangulation, in particular the constrained Delaunay triangulation, of a polygonal chain with k proper self-intersections can be constructed in time $O(n \cdot \min\{\sqrt{k}, \log n\} + k)$. Here, the proper self-intersection points are regarded as vertices that must appear in the triangulation.*

The method from the proof of Theorem 2 can also be used to decide if the number k of self-intersections of a polygonal chain is at most K, for some specified K. Simply, we run the method for $k = K$ and whenever it detects the $K + 1$st crossing, we stop it. In effect, we obtain an $O(n \cdot \min\{\sqrt{K}, \log n\} + K)$-time algorithm for the decision problem.

3 Adaptive Convex Hull Algorithms

We now present an improved adaptive algorithm for computing the convex hull of the vertices of a polygonal chain. The algorithm uses divide-and-conquer and applies in a more general setting to any construction problem $\mathcal{P}$ associated with a polygonal chain that has the following two properties: (1) $\mathcal{P}$ can be solved in linear time for chains that are (weakly) simple; and, (2) the solutions to $\mathcal{P}$ applied to two distinct chains can be merged in time linear in the sizes of the chains. The algorithm is specified by the following simple recursive procedure *Construct*.

Procedure $Construct(chain, \mathcal{A})$

Input: a polygonal chain C and a subroutine $\mathcal{A}$ for the construction problem $\mathcal{P}$ on weakly simple polygonal chains.
Output: a solution to the construction problem $\mathcal{P}$ for C.

1. If C is weakly simple then return $\mathcal{A}(C)$ and stop;
2. $C_1 \leftarrow$ the prefix of C of length $\lceil |C|/2 \rceil$; $C_2 \leftarrow$ the remaining suffix;
3. $Q_1 \leftarrow Construct(C_1, \mathcal{A})$; $Q_2 \leftarrow Construct(C_2, \mathcal{A})$;
4. merge Q_1 with Q_2 and return the result

We analyze the running time in terms of what we call the *simple partition number*, χ, of a chain. More precisely, let χ denote the minimum number of partitioning points at which we need to cut the (open) chain C in order to partition C into simple subchains. If C is simple, then $\chi = 0$. While it is always the case that $\chi \leq k$, since we can cut at the k self-crossings, note that C may have $k = \Omega(n^2)$ self-intersections, while $\chi = 1$. Thus, χ can be a substantially smaller measure of nonsimplicity than is k. (Note that if C is a closed chain, we can artificially open it by replicating one vertex to serve as the start/end of the new open chain.)

Lemma 3. *Assuming there is a linear-time algorithm $\mathcal{A}$ and a linear-time merge algorithm, Construct runs in time $O(n \log(\chi + 2))$.*

Proof. Let $t(n, \chi)$ be the worst-case time taken by the procedure *Construct* for an input chain of n vertices that has simple partition number χ. Then, $t(n, \chi)$ satisfies the following recursion relation:

$$t(n, \chi) \leq \begin{cases} t(\lceil n/2 \rceil, \chi_1) + t(\lfloor n/2 \rfloor, \chi_2) + O(n) & \text{if } \chi \geq 1 \\ O(n) & \text{if } \chi = 0 \end{cases},$$

where $\chi_1 + \chi_2 \leq \chi$. The solution to this recursion gives $t(n, \chi) = O(n \log(\chi + 2))$. □

We can apply Lemma 3 to the problem of computing the convex hull of a polygonal chain, since the convex hull of a weakly simple polygonal chain can be computed in linear time using any of several known convex hull algorithms (e.g., see Melkman [11] and a survey in [7]), and the merge step (computing the convex hull of two convex polygons) is also readily done in linear time.

Theorem 3. *Let C be a polygonal chain of n vertices that can be partitioned into χ simple subchains. Then the convex hull of C can be computed in time $O(n \log(\chi + 2))$.*

Another application of Lemma 3 yields a similar result about sorting intersection points:

Theorem 4. *Given a line ℓ and a polygonal chain C of n vertices that can be partitioned into χ simple subchains, one can compute the sorted order along ℓ of the intersection points $\ell \cap C$ in time $O(n \log(\chi + 2))$.*

Proof. It is known that if C is simple, the points of intersection $\ell \cap C$ can be found in sorted order along ℓ in linear ($O(n)$) time, via "Jordan sorting" [8]. This provides the algorithm $\mathcal{A}$. Merging of two sorted lists is easily done in linear time. □

3.1 Avoiding the Need for Chazelle's Linear-Time Algorithm

One drawback of the above method is that it relies on the highly complex linear-time algorithm of Chazelle [4]. It would be nice, in general, to give a simple and implementable algorithm for the convex hull that is linear-time for small values of k. While we do not know of such a method in general, we can give the following simple linear-time algorithm for the case of $k \leq 1$: Apply Melkman's online convex hull algorithm to the chain C twice – once going forwards along the chain, once going backwards. If the chain is simple ($k = 0$), the algorithm is guaranteed to compute correctly the convex hull (whether we go forwards or backwards). If $k = 1$, then potentially the algorithm fails to compute the convex hull correctly, as Figure 1 (a) shows, if the chain is traversed either forward or backwards. But, the following theorem shows that the convex hull of the chain C is the convex hull of the results of running the algorithm both forwards and backwards along the chain.

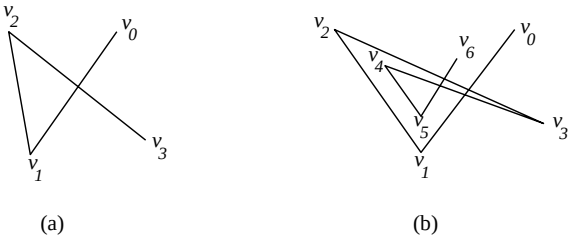


Fig. 1. (a). An example having $k = 1$ of a 4-vertex chain for which Melkman's online convex hull computation fails both for a forwards traversal (which yields triangle $v_0v_1v_2$) and for a backwards traversal (which yields triangle $v_3v_2v_1$); however, the convex hull of the union of the two triangles is the desired output. (b). An example (having $k = 4$) in which Melkman's algorithm applied forwards (resp., backwards) gives triangle $v_0v_1v_2$ (resp., $v_6v_5v_4$), and the convex hull of the union (which is triangle $v_0v_1v_2$) is also not the convex hull of the input chain.

Theorem 5. *The convex hull of a polygonal chain C properly intersecting itself $k \leq 1$ times is given by the convex hull of the union, $P_F \cup P_B$, where P_F (resp., P_B) is the convex polygon produced as the output of Melkman's algorithm while traversing the chain C forwards (resp., backwards).*

Proof. The proof of correctness of Melkman's algorithm establishes that it maintains the convex hull of the sequence of points that have been considered so far: $p_1, p_2, \ldots, p_i$. Suppose that there is exactly one proper self-intersection of the chain C, with edge p_ip_{i+1} crossing p_jp_{j+1}, for $1 \leq i < j \leq n-1$. Then, the subchain $C_F = (p_1, p_2, \ldots, p_j)$ is simple, so the output of Melkman's algorithm, P_F, applied in a forward pass of C is a superset of the convex hull of C_F. Similarly, the backwards subchain $C_B = (p_n, p_{n-1}, \ldots, p_{i+1})$ is simple and the output of Melkman's algorithm, P_B, applied in a backwards pass of C is a superset of the convex hull of C_B. Thus, the convex hull of $P_F \cup P_B$ contains all vertices of C, so, since its vertices are vertices of C, it equals the convex hull of C. □

Corollary 2. *Using two passes of Melkman's online convex hull algorithm, the convex hull of a polygonal chain having at most one self-intersection can be found in $O(n)$ time.*

Remark. If $k > 1$, the two-pass method can fail, as in Figure 1 (b).

3.2 Lower Bounds

We turn now to proving a lower bound on the time required to compute the convex hull of a self-intersecting polygonal chain. (Here, by computing the convex hull, we mean that the output is required to be the vertices of the hull, given in order around the boundary of the hull.)

Theorem 6. *Computing the convex hull of a polygonal chain having simple partition number χ requires $\Omega(n \log(\chi+2))$ time in the algebraic decision tree model of computation.*

Proof. We assume without loss of generality that n is an integral multiple of χ and that $\chi \geq 2$, so that $\log \chi \geq 1$. We use a technique similar to that used in the proof of the $\Omega(n \log n)$ bound for computing the convex hull of a set of n points. The idea is to reduce (in linear time) the problem of sorting n real numbers to the problem of computing the convex hull of a set of points. This is done by producing, for each real number r, a point with (x, y)-coordinates (r, r^2). The resulting set of points is given as input to the convex-hull algorithm (e.g., see [12]).

Let us consider the related problem of sorting χ real numbers that lie within an interval between two consecutive positive integers. In fact, we consider the problem whose instance is an ordered sequence of n/χ copies of this sorting problem, each consisting of an (unordered) subsequence of χ real numbers, in the intervals (1,2), $(3, 4), (5, 6), \ldots, (2n/\chi - 1, 2n/\chi)$. The ith instance of the sorting problem consists of an (unordered) set $S_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,\chi}\}$ of real numbers in the interval $(2i - 1, 2i)$. We know from lower bounds on sorting that it takes $(n/\chi) \times \Omega(\chi \log \chi)$ time, i.e., $\Omega(n \log \chi)$ time, in the worst case to sort the set of n numbers in this instance of the sorting problem.

We reduce this sorting problem in linear time to computing the convex hull of a polygonal chain with n vertices that can be partitioned into χ simple subchains: We map each input real number r to the point $(r, r^2) \in \Re^2$ on the parabola $y = x^2$. We let $p_{i,j} = (x_{i,j}, x_{i,j}^2)$ be the lifted image of $x_{i,j} \in S_i$ on the parabola. We then map, in time $O(n)$, the input instances $S_1, \ldots, S_{n/\chi}$ to the chain $(p_{1,1}, p_{2,1}, \ldots, p_{n/\chi,1}, p_{1,2}, p_{2,2}, \ldots, p_{n/\chi,2}, \ldots, p_{1,\chi}, p_{2,\chi}, \ldots, p_{n/\chi,\chi})$. We note that each of the χ subchains $(p_{1,j}, p_{2,j}, \ldots, p_{n/\chi,j})$ is simple. Thus, the simple partition number of C is at most $\chi - 1$. The proof is concluded by noting that, since the points on the parabola are in convex position, an algorithm that computes the convex hull of C must output the points $p_{i,j}$ in a cyclic order that exactly gives the x-order of the n input real numbers. □

We similarly obtain a lower bound as a function of n and k; for the proof, see the full version of the paper on the authors' web sites.

Theorem 7. *Computing the convex hull of a polygonal chain that properly intersects itself k times requires $\Omega(n \log(k/n))$ time in the algebraic decision tree model of computation.*

Remark. Even if the convex hull algorithm is only required to report which points are vertices of the convex hull, we can obtain the same asymptotic lower bound, using a reduction from other standard problems on sets of real numbers (e.g., the min-gap problem or the element uniqueness problem [12]).

4 Conclusion

We conclude with several interesting open questions:

(1) Can one close the gap between our upper and lower bounds, in terms of n and k, for the adaptive convex hull construction? While our bounds are tight as functions of n and χ, there is a gap when the bounds are written as functions of n and k: $\Omega(n \log(k/n))$ versus $O(n \log(k + 2))$.

(2) Can good adaptive bounds for convex hulls be obtained for large values of χ or k without resorting to the use of Chazelle's complicated linear-time triangulation algorithm?
(3) Can one compute a triangulation of the vertices of a chain C (ignoring the edges of the chain) in time $O(n \log(\chi + 2))$ or time $O(n \log(k + 2))$?
(4) Can one compute the convex hull of a chain C in time $O(n + k)$? (This is interesting in the case that k is super-linear, but $o(n \log n)$.) Can one compute all k self-intersection points of a chain C in time $O(n + k)$?
(5) Can one avoid the use of the complicated algorithm of Chazelle [4] to compute the convex hull of a chain with $\chi \leq 1$ in linear time?
(6) How efficiently can one compute the simple partition number, χ, of a given chain of n vertices? We note that a greedy algorithm that iteratively removes from one end of the chain a maximal length simple subchain can be implemented to run in time $O(n \log n)$. Thus, our goal is to obtain a bound of $o(n \log n)$, at least in the case of small χ.
(7) What bounds can one obtain for the worst-case complexity of computing the convex hull in terms of the input size (n), the degree of non-simplicity (χ and/or k), *and* the output size, h?

References

1. F. Aurenhammer. Jordan sorting via convex hulls of certain non-simple polygons. Proc. 3rd ACM Symposium on Computational Geometry, pp. 21-29, 1987.
2. F. Aurenhammer. On-line sorting of twisted sequences in linear time. BIT 28, pp. 194-204, 1988.
3. I.J. Balaban. An Optimal Algorithm for finding segment intersections. Proc. 11th ACM Symposium on Computational Geometry, pp. 211-219, 1995.
4. B. Chazelle. Triangulating a simple polygon in linear time. Discrete Computational Geometry 6, pp. 485:524, 1991.
5. B. Chazelle. *Personal communication*, 2001.
6. F. Chin and C. Wang. Finding the constrained Delaunay triangulation and constrained Voronoi diagram of a simple polygon in linear time. SIAM J. Comput. Vol. 12, No. 2, pp. 471-486, 1998.
7. J.E. Goodman and J. O'Rourke. Handbook of Discrete and Computational Geometry. CRC Press, Boca Raton, New York 1997.
8. K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. Inform. Control Vol. 68, pp. 170-184, 1986.
9. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? SIAM Journal on Computing 15, pp. 287-299, 1986.
10. C. Levcopoulos and O. Petersson. *Adaptive Heapsort.* Journal of Algorithms 14, pp. 395-413, 1993.
11. A. Melkman. *On-line construction of the convex hull of a simple polyline.* Information Processing Letters 25, pp. 11-12, 1987.
12. F.P. Preparata and M.I. Shamos. Computational Geometry - An Introduction. Texts and Monographs in Computer Science, Springer Verlag, 1985.

Exact Algorithms and Approximation Schemes for Base Station Placement Problems

Nissan Lev-Tov* and David Peleg*

Department of Computer Science and Applied Mathematics,
The Weizmann Institute of Science, Rehovot 76100, Israel
{nissanl,peleg}@wisdom.weizmann.ac.il

Abstract. This paper concerns geometric disk problems motivated by base station placement problems arising in wireless network design. We first study problems that involve maximizing the coverage under various interference-avoidance constraints. A representative problem for this type is the *maximum weight independent set* problem on unit disk graphs, for which we present an exact solution whose complexity is exponential but with a sublinear exponent. Specifically, our algorithm has time complexity $2^{O(\sqrt{m}\log m)}$, where m is the number of disks. We then study the problem of covering all the clients by a collection of disks of variable radii while minimizing the sum of radii, and present a PTAS for this problem.

1 Introduction

1.1 Background

This paper deals with efficient algorithmic solutions for base station placement problems and related problems arising in wireless network design. The input for these problems consists of two sets of points in the Euclidean plane, $X = \{x_1, x_2, \ldots, x_m\}$ representing potential locations for placing base stations, and $Y = \{y_1, y_2, \ldots, y_n\}$ representing the clients. A base station located at x_i has a certain transmission range R_i, which could be either fixed or variable. A client node y_j is *covered* by a base station placed at x_i if it is within its transmission range, namely, if y_j falls within the disk of radius R_i centered at x_i. However, coverage may not be enough; in certain models it is also necessary to avoid interferences between neighboring base stations whose transmission disks partially overlap. Hence our problems concern selecting a placement for the base stations (henceforth referred to in short as *servers*) that will guarantee adequate (interference-free) coverage for the clients while attempting to optimize certain cost functions.

Two design issues affect the nature of the optimization problem at hand. The first concerns the question whether interference avoidance must be enforced. In particular, when interferences are ignored, the problem to be solved is typically a disk-covering problem, namely, finding a minimum cost collection of servers covering all clients. In contrast, when interferences must be avoided, it might be

* Supported in part by a grant from the Israel Ministry of Industry and Commerce.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 90–99, 2002.

impossible to satisfy all the clients simultaneously, hence it is of interest to study also variants of the problem aimed at maximizing the number of clients which are covered by *exactly* one server (henceforth referred to as *supplied* clients). A combined approach which is examined as well is to simultaneously take into account the number of supplied clients and the cost of the servers, by attempting to optimize total *profit*, defined as the gain from supplied clients minus the cost of the servers.

The second design issue is whether transmission radii are fixed or variable. When the radii are variable, we must decide on the transmission range of each server to be built, in addition to choosing its locations. The typical goal is to choose for each server x_i a transmission radius R_i such that all the clients are covered and the sum of radii is minimized. This target arises from the assumption that the cost of choosing a certain radius depends linearly on that radius. Such assumption is often made in various clustering and covering problems [1].

1.2 The Problems

This paper considers problems of two main types. The first involves maximization problems on disks of fixed radius. Given the locations of the clients and servers and a fixed transmission range R, it is required to choose an active set of servers in an optimal way. Several optimization targets can be considered, all taking into account interferences between active servers.

We start with the problem of maximizing the number of supplied clients under the condition that the R-disks around the servers are disjoint. This requirement is perhaps not the most natural in the context of base station placement, as overlaps over client-free regions should cause no problems. On the other hand, this problem can be handled as a special case of the *maximum weight independent set (*MWIS*)* problem on unit disk graphs, which is of independent interest. In the MWIS problem there are weights associated with the points in X, and it is required to choose a maximum weight subset of X such that the R-disks around them are all disjoint. To get our problem as a special case of the MWIS problem, the weight of each vertex $x_i \in X$ is set to the number of clients covered in the R-disk around x_i.

We also consider a number of related problems, which capture the restrictions of the model more adequately. The first problem requires us to maximize the number of supplied clients under the constraint that no client is in the transmission range of more than one chosen server. Again, this problem can be generalized into the *maximum weight collision-free set (*MWCS*)* problem, which is to find a maximum weight subset of servers such that the R-disks around the servers do not contain common clients. Another variant of this problem, named *maximum supplied clients (*MSC*)*, which may be even more useful from practical point of view, is defined as follows. For any subset $\tilde{X}$ of X, let $\texttt{Supp}(\tilde{X})$ denote the set of clients supplied by $\tilde{X}$. The problem is to choose a subset of servers $\tilde{X}$ maximizing the number of supplied clients, $|\texttt{Supp}(\tilde{X})|$. We also consider a variant of the problem named *maximum profit (*MP*)*, in which costs and benefits are treated in a combined manner. The goal is to choose a subset of servers maximizing the *profit* $P(\tilde{X}) = c_1|\texttt{Supp}(\tilde{X})| - c_2|\tilde{X}|$, where $c_1, c_2 > 0$ are given constants.

The MWCS, MSC and MP problems turn out to be hard to manage for arbitrary inputs, and we consider them in a restricted setting referred to as the *grid-based* setting. In this setting, the set X of possible server locations is restricted to grid points of a given spacing, fixed for concreteness to be 1, and moreover, the transmission range R is assumed to be bounded by a constant. Without loss of generality we assume that the underlying unit grid G_1 is aligned so that $(0, 0)$ occurs as a grid point. Throughout, the grid-based version of a problem PROB is denoted PROB_g.

The second type of problems considered in this paper is disk-covering problems, where the goal is to achieve minimum sum of radii. Given a set of clients Y and a set of servers X in the plane, we aim to choose the transmission range R_i of each server x_i such that all the clients are covered and the sum of the transmission ranges chosen, $\varphi = \sum_i R_i$, is minimized. Although any radius R_i can be chosen for a given server, every solution is dominated by a solution in which for each chosen radius $R_i \neq 0$ the corresponding R_i-disk has a client on its border. So the problem is equivalent to choosing from all $n \cdot m$ disks centered at a server and with a client on their border. This problem is referred to as *minimum sum of radii cover (*MSRC*)*.

We also study the variant of this problem where interferences are involved, meaning that the transmission ranges must be chosen such that no two disks intersect on a client. We call this problem *minimum sum of radii cover with interferences (*MSRCI*)*. Its representation is more general as we are given a set of disks of various radii and we have to choose a subset of the given set of disks such that the chosen disks are all disjoint and minimum sum of radii is achieved.

1.3 Previous Work

The disk-covering problem on fixed radius disks is studied in [5], in a model where the server locations are not restricted to a given set of possible locations but rather may be chosen at any point on the plane. A PTAS is given for this problem using a grid-shifting strategy.

Fixed radius covering problems where only potential server locations are considered, with or without interferences, are studied in [4]. They consider optimization problems for cellular telephone networks that arise in a traffic load model which also addresses the positioning of servers on given possible locations with the aim of maximizing the number of supplied clients and minimizing the number of servers to be built.

A technique called *slab dividing* is proposed in [7]. It is used there to give a sub-exponential exact solution of time complexity $O(n^{O(\sqrt{P})})$ for the Euclidean P-center problem. This approach is essentially based on a version of the $\sqrt{n}$-planar separator theorem of [8], suitably adapted to the Euclidean case.

The MWIS problem on unit-disk graphs is shown to be NP-hard in [2], and is given a PTAS in [6]. The MWIS problem on general (arbitrary radii) disk graphs is considerably harder, and was only recently shown to have a PTAS using a sophisticated hierarchical grid-shifting technique [3].

1.4 Our Results

The paper presents exact and approximate solutions for the above problems. We begin in Section 2 by developing a variant of the slab technique of [7] which is suitable for handling maximum independent set and maximum covering set problems. We then apply our method for deriving an $2^{O(\sqrt{m}\log m)}$ time exact solution for the MWIS problem.

Using variations of our method it is possible to obtain similar results for a number of grid-based problems. In particular, the grid-based MWIS_g problem can be given a slightly better $2^{O(\sqrt{m})}$ time solution, and a similar solution exists for the grid-based MWCS_g problem. The grid-based problems MSC_g and MP_g enjoy $2^{O(\sqrt{m}+\log n)}$ time exact solutions. The details of these results are also deferred to the full paper. In the full paper we also provide a PTAS for the grid-based MP_g problem, using a grid-shifting strategy similar to that of [5].

We then turn to the variable radii model. In section 3 we present a PTAS for the MSRC problem (with no interference constraints), based on a modified variant of the hierarchical grid-shifting technique of [3].

Note that all our results can be extended for the case where each client has a certain weight (say, representing the fee paid by this client or the significance of providing it with service), and the optimization targets refer to the sum of weights of the supplied clients instead of merely their number.

While our focus is on the natural 2-dimensional variants of the above problems, we also studied their 1-dimensional variants. Our polynomial time solutions for the 1-dimensional MWIS, MWCS, MSC and MP problems and the MSRC problems with and without interference constraints are deferred to the full paper.

2 Maximization Problems on Fixed Radius Disks

In this section we develop a variant of the slab method of [7] suitable for handling maximum independent set and maximum covering set problems, and then apply this method for giving a $2^{\tilde{O}(\sqrt{m})}$ exact solution for a number of problems, including MWIS, MWCS_g, MSC_g and MP_g.

Let us first describe our variant of the slab method. Subdivide the plane by introducing a grid G_δ of lines at distance $\delta = 2R$ of each other, aligned so that the point $(0,0)$ is a grid point. The vth vertical line, $-\infty < v < \infty$, is at $x = v\cdot\delta$; the index v will be used to identify the line. The same goes for horizontal line h.

Each vertical line v defines a vertical *slab* denoted $\texttt{Slab}(v)$, which is the strip $v\cdot\delta \le x < (v+1)\cdot\delta$. Similarly, $\texttt{Slab}(h)$ is the horizontal strip $h\cdot\delta \le y < (h+1)\cdot\delta$. These vertical and horizontal slabs induce $\delta\times\delta$ squares which are open on their right and upper sides and will be referred to as *grid-squares*. A grid-square is called *occupied* if it contains a server, otherwise it is called *empty*. We use these definitions for vertical and horizontal slabs as well.

Without loss of generality, all the points of the problem instance are contained in a rectangle with boundaries v_1, v_2, h_1, h_2 whose sides are each no larger than $m\cdot\delta$ (otherwise there must be an empty slab which divides the problem instance into two independent problems each contained in a smaller rectangle, and in this

case each of the rectangles could be dealt with separately). Let M denote the number of occupied squares contained in this rectangle.

For a vertical slab $\texttt{Slab}(v)$, $v_1 \leq v < v_2$, denote by $\texttt{In}(v)$ the set of occupied grid-squares contained in $\texttt{Slab}(v)$, by $\texttt{Left}(v)$ the set of occupied grid-squares to the left of $\texttt{Slab}(v)$ and by $\texttt{Right}(v)$ the set of occupied grid-squares to the right of $\texttt{Slab}(v)$. For a horizontal slab $\texttt{Slab}(h)$ define $\texttt{In}(h)$, $\texttt{Above}(h)$ and $\texttt{Below}(h)$ in a similar way. By the above definitions, for every v and h,

$$|\texttt{Left}(v)|+|\texttt{In}(v)|+|\texttt{Right}(v)| = M \quad \text{and} \quad |\texttt{Below}(h)|+|\texttt{In}(h)|+|\texttt{Above}(h)| = M.$$

Definition: For any input instance X,Y, a *dividing slab* is either a vertical slab $\texttt{Slab}(v)$, $v_1 \leq v < v_2$ such that (1) $|\texttt{In}(v)| \leq 5\sqrt{M}$, (2) $|\texttt{Left}(v)| \leq \frac{4}{5} \cdot M$, and (3) $|\texttt{Right}(v)| \leq \frac{4}{5} \cdot M$, or a horizontal slab $\texttt{Slab}(h)$, $h_1 \leq h < h_2$ such that (1) $|\texttt{In}(h)| \leq 5\sqrt{M}$, (2) $|\texttt{Above}(h)| \leq \frac{4}{5} \cdot M$, and (3) $|\texttt{Below}(h)| \leq \frac{4}{5} \cdot M$.

Lemma 1. *For any input instance X,Y, there exists a dividing slab.*

Proof. Let v'_1 be the smallest vertical line index such that $|\texttt{Left}(v'_1)| > M/5$ and let v'_2 be the largest vertical line index such that $|\texttt{Right}(v'_2-1)| > M/5$. Similarly, let h'_1 be the smallest horizontal line index such that $|\texttt{Below}(h'_1)| > M/5$ and let h'_2 be the largest horizontal line index such that $|\texttt{Above}(h'_2 - 1)| > M/5$. Note that $|\texttt{Left}(v'_1 - 1)| \leq M/5$ by choice of v'_1, hence $|\texttt{Right}(v'_1 - 2)| > 4M/5$. This implies that $v'_1 - 1 \leq v'_2$, since otherwise $v'_1 - 1$ should have been chosen instead of v'_2. Moreover, each of the vertical slabs in $L_V = \{\texttt{Slab}(v) \mid v'_1 - 1 \leq v \leq v'_2\}$ satisfies $|\texttt{Left}(v)| \leq 4M/5$ and $|\texttt{Right}(v)| \leq 4M/5$. Analogous statements apply for horizontal slabs in $L_H = \{\texttt{Slab}(h) \mid h'_1 - 1 \leq h \leq h'_2\}$ with $\texttt{Above}$ and $\texttt{Below}$.

We now show that at least one of the slabs in $L_V \cup L_H$ also satisfies the first condition of a dividing slab. Let $\mathcal{R}$ be the rectangle bounded by $v'_1-1, v'_2+1, h'_1-1, h'_2+1$, and let M' be the number of occupied grid-squares in $\mathcal{R}$. Note that $\mathcal{R}$ contains all the occupied rectangles save those in $\texttt{Left}(v'_1 - 1)$, $\texttt{Right}(v'_2 + 1)$, $\texttt{Below}(h'_1 - 1)$ and $\texttt{Above}(h'_2 + 1)$, whose total number is at most $4M/5$, and therefore $M' \geq M/5$.

There are $l_v = |L_V| = v'_2 - v'_1 + 2$ vertical slabs and $l_h = |L_H| = h'_2 - h'_1 + 2$ horizontal slabs in the rectangle $\mathcal{R}$. We now observe that $l_v \cdot l_h < M'$ by the definition of M', and therefore either $l_v \geq \sqrt{M'}$ or $l_h \geq \sqrt{M'}$. Without loss of generality suppose $l_v \geq \sqrt{M'}$. As $M' \geq M/5$, we conclude that $l_v \geq \sqrt{M/5}$.

If $|\texttt{In}(v)| > 5 \cdot \sqrt{M}$ for each vertical slabs $\texttt{Slab}(v) \in L_V$, then we get more than M occupied grid-squares altogether. Therefore there must be a slab $\texttt{Slab}(v) \in L_V$ for which $|\texttt{In}(v)| \leq 5 \cdot \sqrt{M}$. ∎

The above lemma suggests a generic recursive procedure which is at the heart of a number of algorithms for solving some geometric optimization problems. We next discuss the resulting algorithm for the MWIS problem and state the results for the grid-based version of the other problems.

Let $X_{(v_1,v_2,h_1,h_2)}$ and $Y_{(v'_1,v'_2,h'_1,h'_2)}$ denote the sets of servers and clients restricted to the area bounded by the lines v_1, v_2, h_1, h_2 and v'_1, v'_2, h'_1, h'_2, respectively. Let $\text{MWIS}[X_{(v_1,v_2,h_1,h_2)}, Y_{(v_1,v_2,h_1,h_2)}]$ denote the problem restricted to those servers and clients.

The procedure OPTIMIZE(X, Y) recursively finds an optimal set of servers. The bottom level of the recursion is when X is contained in a single disk square, In this case the procedure computes an optimal solution by an exhaustive search, going through all possible choices of three servers in the square.(Note that choosing four servers inside a single $\delta \times \delta$ grid-square will necessarily cause their R-disks to intersect, even if they are symmetrically located on the four corners.) Each step starts with the procedure identifying a dividing slab $\texttt{Slab}(v)$ or $\texttt{Slab}(h)$. This is done by exhaustively examining each slab. The procedure then cycles over all local (not necessarily optimal) partial solutions X_t of the local problem where the servers are contained in squares of $\texttt{In}(v)$ or $\texttt{In}(h)$, respectively. Each such partial solution X_t is composed of all possible choices of up to three servers inside each grid-square of $\texttt{In}(v)$ or $\texttt{In}(h)$. For each such partial solution X_t, the procedure then creates the left and right (resp., above and below) subproblems, in which the servers are restricted to squares of $\texttt{Left}(v)$ and $\texttt{Right}(v)$ (resp., $\texttt{Below}(h)$ and $\texttt{Above}(h)$), while deleting from these subproblems the set of servers $\texttt{Neighbors}(X_t)$ whose disks intersect disks of X_t. We now solve the left and right (resp., below and above) subproblems recursively. The procedure stops when remaining with a single square, in which case it finds a local solution by exhaustively searching all $O(m^3)$ possibilities of choosing a disjoint set of disks with servers inside the grid-square.

Procedure OPTIMIZE(X, Y)

If X is contained in a single grid-square
  then exhaustively compute and return an optimal solution.
Find either a vertical dividing slab $\texttt{Slab}(v)$ or a horizontal dividing slab $\texttt{Slab}(h)$.
If a vertical dividing slab $\texttt{Slab}(v)$ is found then do:
  For each local solution X_t of MWIS$[X_{(v,v+1,h_1,h_2)}, Y]$ do:
    $X_l \leftarrow$ OPTIMIZE$(X_{(v_1,v,h_1,h_2)} - \texttt{Neighbors}(X_t), Y_{(v_1,v+1,h_1,h_2)})$
    $X_r \leftarrow$ OPTIMIZE$(X_{(v+1,v_2,h_1,h_2)} - \texttt{Neighbors}(X_t), Y_{(v,v_2,h_1,h_2)})$
    $X_P \leftarrow X_t \cup X_l \cup X_r$
Else do:
  For each local solution X_t of MWIS$[X_{(v_1,v_2,h,h+1)}, Y]$ do:
    $X_l \leftarrow$ OPTIMIZE$(X_{(v_1,v_2,h_1,h)} - \texttt{Neighbors}(X_t), Y_{(v_1,v_2,h_1,h+1)})$
    $X_r \leftarrow$ OPTIMIZE$(X_{(v_1,v_2,h+1,h_2)} - \texttt{Neighbors}(X_t), Y_{(v_1,v_2,h,h_2)})$
    $X_P \leftarrow X_t \cup X_l \cup X_r$
Return X_P of maximum weight among all the X_ts.

To analyze the algorithm, first note that after deleting the neighbors of X_t we are left with two *independent* subproblems to be solved recursively. This is because the dividing slab is wide enough so that the servers in the two subproblems never cover the same clients. By using an efficient data structure we can find a dividing slab in $O(m \log m)$ time.

By Lemma 1 and the fact that no more than three servers can be chosen inside each grid-square, there are no more than $\left(\sum_{i=1}^{3} \binom{m}{i}\right)^{5\sqrt{M}} \leq m^{15\sqrt{m}}/2$ possible solutions X_t with servers inside the squares of the slab. For each X_t, the process of deleting its neighbors takes at most $O(m)$ time. Thus for $M \geq 2$ we have the

recurrence $T(M) \leq m \log m + (2T(4M/5) + m) \cdot m^{15\sqrt{M}}/2$, so after the kth iteration $T(M) < m^{15\sqrt{M}\sum_{i=0}^{k-1}\sqrt{4/5}^i} \cdot T\left(\left(\frac{4}{5}\right)^k \cdot M\right) + \sum_{i=0}^{k-1} m^{15\sqrt{M}\cdot\sum_{j=0}^{i}\sqrt{4/5}^j + 2}$. The problem on a single grid-square requires choosing at most three servers so $T(1) = O(m^3)$, and as $\sum_{i=0}^{\infty}\sqrt{4/5}^i = \frac{1}{1-\sqrt{4/5}} < 10$ we have that $k = \log_{5/4} M$. So as $M \leq m$, we have $T(M) = 2^{O(\sqrt{m}\log m)}$.

Theorem 1. *The* MWIS *problem has an* $2^{O(\sqrt{m}\log m)}$ *time exact algorithm.*

In the full paper we show the following.

Theorem 2.

1. *The* MWIS_g *and* MWCS_g *problems have an* $2^{O(\sqrt{m})}$ *time exact algorithm.*
2. *The* MSC_g *and* MP_g *problems have an* $2^{O(\sqrt{m}+\log n)}$ *time exact algorithm.*
3. *The* MP_g *problem admits a PTAS which for every* $k \geq 3$ *guarantees an approximation ratio of* $1 - 2/k$ *with time complexity* $O(k^2 \cdot m \cdot 2^{((k-1)\delta)^2})$. *Similar claims hold also for the* MWIS_g, MWCS_g *and* MSC *problems.*

3 Minimum Sum of Radii Problems

In this section we turn to the variable radii model, and consider the MSRC problem. Let $\mathcal{D}$ be the set of $n \cdot m$ disks determined by the sets X and Y as follows. For each $1 \leq p \leq m$ and $1 \leq q \leq n$, the client $y_q \in Y$ and the server $x_p \in X$ determine a disk $D_p^q \in \mathcal{D}$ of radius $R_p^q = dist(x_p, y_q)$ centered at x_p. A weight $\omega_p^q = R_p^q$ is associated with each disk D_p^q, and $\omega(\mathcal{D}') = \sum_{D_p^q \in \mathcal{D}'} \omega_p^q$ is the total weight for a set of disks $\mathcal{D}' \subseteq \mathcal{D}$. Let $\delta_p^q = 2R_p^q$ be the diameter of disk D_p^q and let δ_{max} and δ_{min}, respectively, be the maximal and minimal values of δ_p^q for all possible p and q.

A set of disks $\mathcal{D}' \subseteq \mathcal{D}$ is called a *cover* for a set of clients $Y' \subseteq Y$ if each client $y_p \in Y'$ is contained in some disk of $\mathcal{D}'$. The problem is to find a cover $\mathcal{D}' \subseteq \mathcal{D}$ for Y of minimum total weight $\omega(\mathcal{D}')$. We now present a polynomial time algorithm that approximates the problem with ratio $1 + \frac{6}{k}$ for every given integer $k > 1$.

First, the disks in $\mathcal{D}$ are scaled such that $\delta_{max} = 1$. Given $k > 1$, Let $\ell = \lfloor \lg_{k+1}(\frac{1}{\delta_{min}}) \rfloor$. The set $\mathcal{D}$ is partitioned into $\ell + 1$ *levels* s.t. for $0 \leq j \leq \ell$ the disk D_p^q is on level j if and only if its diameter δ_p^q satisfies $(k+1)^{-(j+1)} < \delta_p^q \leq (k+1)^{-j}$. Disks of level j are called *j-disks*.

Analogously, we subdivide the plane by introducing a hierarchy of increasingly finer grids, s.t. each level $0 \leq j \leq \ell$ imposes a grid G_j of lines at distance $(k+1)^{-j}$ of each other, aligned so there is a grid point at $(0,0)$. We refer to lines of the level j grid G_j as *j-lines*. A vertical j-line is of *index* v, for integer $-\infty < v < \infty$, if its x coordinate is $x = v(k+1)^{-j}$; a similar definition applies to horizontal j-lines.

On top of each grid G_j of this hierarchy, we now construct a coarser supergrid $SG_j(v,h)$ for every $0 \leq v, h < k$ as follows. For every $0 \leq v < k$, let $\texttt{VL}_j(v)$

denote the collection of vertical j-lines of G_j whose index modulo k equals v. Similarly, for every $0 \leq h < k$, let $\mathtt{HL}_j(h)$ denote the collection of horizontal j-lines of G_j whose index modulo k equals h. Now the super-grid $SG_j(v,h)$, for $0 \leq v,h < k$, consists of the line collection $\mathtt{VL}_j(v) \cup \mathtt{HL}_j(h)$.

For fixed (v,h) and a certain level $0 \leq j \leq \ell$, the lines of $SG_j(v,h)$ subdivide the plane into disjoint squares of side $k \cdot (k+1)^{-j}$, called *j-squares*. A j-square J is called *relevant* if $\mathcal{D}$ contains a j-disk that covers a client in J. For a relevant j-square J and a relevant j'-square J' where $j' > j$, J' is called a *child* of J (denoted $J' \prec J$) if it is contained in J and there is no "intermediate" relevant j''-square s.t. $j < j'' < j'$ and $J' \subset J'' \subset J$.

For a fixed choice of v,h and level j, for each j-disk $D_p^q \in \mathcal{D}$ and relevant j-square J imposed by v and h, we define the *disk-sector* $S_p^q(J)$ *induced* by D_p^q and J as their intersection. The disk sector $S_p^q(J)$ is then called *relevant* if it contains a client of J. Note that each disk can induce up to four relevant disk sectors.

Also, for each choice of v,h and level j, let $\mathcal{D}_j(v,h)$ be the set of relevant disk-sectors which are induced by $j-disks$ and $j-squares$ of $SG_j(v,h)$. For given v,h, let $\mathcal{D}(v,h) = \bigcup_{0 \leq j \leq \ell} \mathcal{D}_j(v,h)$. This set is made of disk-sectors that are each completely contained in a relevant square on their level. Note that a disk-sector can be a full disk if the whole disk is contained in a square of the same level. The weight of each disk-sector is defined to be the radius of the original disk and the level and center of the disk-sector remain the same as well.

Two disks $D_{p_1}^{q_1}$ and $D_{p_2}^{q_2}$ are said to be *semi-disjoint* if $D_{p_1}^{q_1}$ does not contain x_{p_2} and $D_{p_2}^{q_2}$ does not contain x_{p_1}. A subset of disk-sectors $\mathcal{S} \in \mathcal{D}(v,h)$ is then said to be semi-disjoint if $D_{p_1}^{q_1}$ and $D_{p_2}^{q_2}$ are semi-disjoint for every $S_{p_1}^{q_1}(J_1), S_{p_2}^{q_2}(J_2) \in \mathcal{S}$.

Let $OPT(\mathcal{D}(v,h))$ be the optimal value of a cover for Y that can be obtained when restricted to the elements of $\mathcal{D}(v,h)$. We then have the following lemma.

Lemma 2. *At least one pair (v,h), for some $0 \leq v,h < k$, satisfies $OPT(\mathcal{D}(v,h)) \leq (1+6/k) \cdot OPT(\mathcal{D})$.*

Proof. Let $\mathcal{C}$ denote the optimal cover by the original disks, i.e., such that $OPT(\mathcal{D}) = \omega(\mathcal{C})$. For each choice of v and h, let $\mathcal{C}(v,h)$ be the set of relevant disk sectors of $\mathcal{C}$ as mentioned above. Because the spacing between j-lines is no smaller than the diameter of a j-disk, it follows that if a j-disk of $\mathcal{C}$ was cut by a vertical j-line in $SG_j(v,h)$ then for all other choices of $v' \neq v$ this disk will not be cut by any vertical j-line of $\mathtt{VL}_j(v')$. Also, if a j-disk of $\mathcal{C}$ was cut by a horizontal j-line in $SG_j(v,h)$ then for all other choices of $h' \neq h$ this disk will not be cut by any horizontal j-line of $\mathtt{HL}_j(h')$.

For $0 \leq v < k$ and a level j, let $\mathcal{C}_j^v$ be the set of all j-disks in $\mathcal{C}$ that intersect a line of $\mathtt{VL}_j(v)$, and let $\mathcal{C}^v = \bigcup_j \mathcal{C}_j^v$. Note that $\mathcal{C} = \bigcup_v \mathcal{C}^v$ and by the above argument the sets $\mathcal{C}^v$ for $0 \leq v < k$ are disjoint, so $\omega(\mathcal{C}) = \sum_v \omega(\mathcal{C}^v)$. Therefore the weight of at least one of these sets must be at most a $\frac{1}{k}$-fraction of the weight of $\mathcal{C}$, i.e., $\omega(\mathcal{C}^v) \leq \omega(\mathcal{C})/k$ for some $0 \leq v \leq k$. Similarly, letting $\mathcal{C}_j^h$ be the set of all j-disks in $\mathcal{C}$ that intersect a line of $\mathtt{HL}_j(h)$ and letting $\mathcal{C}^h = \bigcup_j \mathcal{C}_j^h$, we get that $\omega(\mathcal{C}^h) \leq \omega(\mathcal{C})/k$ for some $0 \leq h < k$. Hence for these v and h, $\omega(\mathcal{C}^v \cup \mathcal{C}^h) \leq$

$\frac{2}{k}\omega(\mathcal{C})$. Each disk D_p^q of $\mathcal{C}^v \cup \mathcal{C}^h$ induces up to four disk-sectors in $\mathcal{C}(v,h)$ with weight $\omega(D_p^q)$. Thus when calculating $\omega(\mathcal{C}(v,h))$, we count the weight of each disk of $\mathcal{C}$ while possibly adding the weight of the disks which belong to $\mathcal{C}^v \cup \mathcal{C}^h$ at most three times more. We therefore have $\omega(\mathcal{C}(v,h)) \leq (1+\frac{6}{k}) \cdot \omega(\mathcal{C})$. ∎

For fixed v, h and j-square J we use the following terminology. For any subset $\mathcal{S} \subseteq \mathcal{D}(v,h)$ and integers $0 \leq a \leq b \leq k$, $\mathcal{S}^J_{[a,b]}$ is the set of all sectors in $\mathcal{S}$ of levels in the range $[a,b]$ that contain a client of J. If $a=b$ we write simply $\mathcal{S}^J_a$. A *full partial cover* of J is a collection $\mathcal{F}$ of semi-disjoint disk-sectors of levels in the range $[0,j]$ that contain a client of J such that $\mathcal{F}$ is a cover for the clients of J that cannot be covered by any of the disk-sectors of levels in $[j+1,l]$. A *partial cover* $\mathcal{P}$ of J is a set of disk-sectors $\mathcal{F}^J_{[0,j-1]}$ for some full-partial cover $\mathcal{F}$ of J. For a partial-cover $\mathcal{P}$ of J, a *(J,P)-completion* is a collection $\mathcal{C}$ of disk-sectors of levels in the range $[j,l]$ that contain a client of J such that $\mathcal{C} \cup \mathcal{P}$ is a cover for the clients of J and $\mathcal{P} \cup \mathcal{C}^J_j$ is a full partial cover for the clients of J.

Lemma 3. *For a j-square J there exists a constant γ (depending only on k), such that for every full-partial cover $\mathcal{F}$ of J, $|\mathcal{F}| \leq \gamma$.*

Proof. The disk sectors of $\mathcal{F}$ are induced by J or by a square containing J, so no two sectors are induced by the same disk. Therefore, letting $\mathcal{Q}'$ be the set of (semi-disjoint) disks inducing the sectors of $\mathcal{F}$ we have $|\mathcal{F}| \leq |\mathcal{Q}'|$. Consequently, it suffices to show that there exists a constant γ such that for a set $\mathcal{Q}$ of semi-disjoint disks of levels in the range $[0,j]$ that intersect J, $|\mathcal{Q}| \leq \gamma$.

Let $\mathcal{Q}$ be such a set of disks and let $\psi = (k+1)^{-j}$ and $d = \frac{\psi}{2\cdot(k+1)}$. Let $d_{\min}(\mathcal{Q})$ denote the minimal distance between the centers of $\mathcal{Q}$. The semi-disjointness property of $\mathcal{Q}$ implies that $d_{\min}(\mathcal{Q}) \geq d$. Let J' be a square consisting of J and a strip of width ψ surrounding J. Let $\mathcal{Q}^{J'}$ be the set of disks in $\mathcal{Q}$ whose centers are contained in J' and let $\overline{\mathcal{Q}^{J'}} = \mathcal{Q} - \mathcal{Q}^{J'}$.

To bound $|\mathcal{Q}^{J'}|$, subdivide J' into a grid GH composed of lines at distance $d/\sqrt{2}$ of each other (the last strip will be narrower, as the side length of J' is not an integral multiple of $d/\sqrt{2}$). As $d_{\min}(\mathcal{Q}) \geq d$, each square of the grid GH contains at most one center of a disk of $\mathcal{Q}^{J'}$, hence the size of $\mathcal{Q}^{J'}$ is bounded by the number of grid squares, i.e., $|\mathcal{Q}^{J'}| \leq \frac{(k+2)\psi^2}{d^2/2} = O(k^4)$.

It remains to bound the size of $\overline{\mathcal{Q}^{J'}}$. Observe that this set consists of disks of levels smaller than j, because the centers of all disks of level j must fall inside J'. Consider the grid G_j and ignore all the j-lines that do not touch the square J. To this grid add two lines which determine the diagonals of J. These lines impose a subdivision of $\mathbb{R}^2 - J'$ into $4(k+2)$ regions. The proof is completed upon verifying that each part of the subdivision can contain at most one center of the disks of $\overline{\mathcal{Q}^{J'}}$, and hence $|\overline{\mathcal{Q}^{J'}}| \leq 4(k+2)$. ∎

The algorithm for computing the table T uses the procedure $\textsc{Proc}(J,\mathcal{F})$ for each j-square J and full partial cover $\mathcal{F}$. This procedure looks up the table entries $T(J', \mathcal{F}^{J'}_{[0,j]})$ of all children J' of J (which were already computed) and outputs their union $\mathcal{U}$.

Procedure PROC is used within the procedure CONS(T) for constructing the table T. Procedure CONS operates on the levels j from l to 0. For each j, the procedure looks at each relevant j-square J and each full partial cover $\mathcal{F}$ of J, computes $\mathcal{P} = \mathcal{F}^J_{[0,j-1]}$ and then uses Procedure PROC to compute $\mathcal{C} = \text{PROC}(J, \mathcal{F}) \cup \mathcal{F}^J_j$. Finally, it updates $T(J, \mathcal{P})$ to $\mathcal{C}$ provided this entry was still undefined or its previous weight was higher than $\omega(\mathcal{C})$.

The main algorithm operates as follows. For each choice of v, h, Let $\mathcal{R}$ be the set of relevant squares and $\mathcal{R}_0$ the set of relevant squares without a parent. Then the algorithm outputs the minimum of $\omega(\bigcup_{J \in \mathcal{R}_0} T(J, \emptyset))$ over all pairs v, h.

In the analysis, deferred to the full paper, we prove that for each relevant j-square J and partial cover $\mathcal{P}$, the table entry $T(J, \mathcal{P})$ is a minimum weight $(J, \mathcal{P})$-completion, and that for a minimum-weight cover $\mathcal{M} \subseteq \mathcal{D}(v, h)$ for Y and a relevant j-square J, $\mathcal{M}^J_{[0,j]}$ is semi-disjoint. Verifying that the algorithm is polynomial in the input size, we have the following result.

Theorem 3. *The* MSRC *problem admits a PTAS.*

References

1. M. Charikar and R. Panigary. Clustering to minimize the sum of cluster diameters. In *Proc. 33rd ACM Symp. on Theory of Computing*, July 2001.
2. B.N. Clark, C.J. Colbourn, and D.S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86:165–177, 1990.
3. T. Erlebach, K. Jansen, and E. Seidel. Polynomial-time approximation schemes for geometric graphs. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms*, 2001.
4. C. Glasser, S. Reith, and H. Vollmer. The complexity of base station positioning in cellular networks. In *Proc. ICALP Workshops*, 167–177. Carleton Press, 2000.
5. D.S. Hochbaum and W. Maas. Approximation schemes for covering and packing problems in image processing and VLSI. *J. ACM*, 32:130–136, 1985.
6. H.B Hunt, S.S. Ravi, M.V.Marathe, D.J.Rosenkrantz, V.Radhakrishnan, and R.E.Stearns. NC-approximation schemes for NP-and PSPACE-hard problems for geometric graphs. *Journal of Algorithms*, 26(2):238–274, 1998.
7. R.Z Hwang, R.C.T Lee, and R.C. Chang. The slab dividing approach to solve the Euclidian p-center problem. *Algorithmica*, 9:1–22, 1993.
8. R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. on Applied Math.*, 36(2):177–189, April 1979.

A Factor-2 Approximation for Labeling Points with Maximum Sliding Labels*

Zhongping Qin[1,2] and Binhai Zhu[2]

[1] Department of Mathematics, Huazhong University of Science and Technology, Wuhan, China. qin@cs.montana.edu.
[2] Department of Computer Science, Montana State University, Bozeman, MT 59717-3880, USA. bhz@cs.montana.edu.

Abstract. In this paper we present a simple approximation algorithm for the following NP-hard map labeling problem: Given a set S of n distinct sites in the plane, one needs to place at each site an axis-parallel sliding square of maximum possible size (i.e., a site can be anywhere on the boundary of its labeling square) such that no two squares overlap and all the squares have the same size. By exploiting the geometric properties of the problem, we reduce a potential 4SAT problem to a 2SAT problem. We obtain a factor-2 approximation which runs in $O(n^2 \log n)$ time using discrete labels. This greatly improves the previous factor of 4.

1 Introduction

Map labeling is an old art in cartography and finds new applications in recent years in GIS, graphics and graph drawing [1,4,5,12,14,15,19,17,18,21,25,26,27]. About a decade ago, an interesting relation between 2SAT and map labeling was found by Formann and Wagner [12]. This finding leads to a series of exact and approximate solutions for different map labeling problems; among them, the first factor-2 approximation for labeling points with maximum discrete squares [12], the first polynomial time solution for labeling a set of disjoint rectilinear line segments [21], a factor-3.6 approximation for labeling points with maximum circles [7], and a factor-2 approximation for labeling points with maximum square pairs [22]. (Recently, we obtain a very simple factor-3 approximation for labeling points with maximum circles [23].)

The idea of using 2SAT in map labeling is very simple. Suppose that we have two points p, q and somehow we want to label p, q using two candidate labels each, one upper and one lower (Figure 1), we need to pick one label for each point. Assume that we use a binary variable X (Y) to encode the labeling of p (q) in Figure 1, i.e., if we pick the upper label for p (q) then $X = 1$ ($Y = 1$) else $X = 0$ ($Y = 0$), then in the example of Figure 1 we need to satisfy the following formula $\neg(X \wedge Y) \wedge \neg(\neg X \wedge Y) \wedge \neg(\neg X \wedge \neg Y)$, where $\neg(u \wedge v)$ means 'we do not want u, v to be true at the same time'. Simplifying the above formula, we

* This research is partially supported by Hong Kong RGC CERG grant CityU1103/99E, NSF CARGO grant DMS-0138065 and a MONTS grant.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 100–109, 2002.

have $(\neg X \vee \neg Y) \wedge (X \vee \neg Y) \wedge (X \vee Y)$. It is easy to find a truth assignment $X = 1, Y = 0$ for this 2SAT formula, which implies that we should choose the upper label for p and lower label for q.

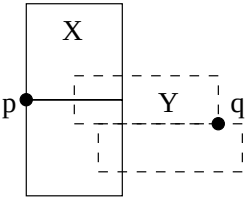


Fig. 1. An example of using 2SAT in map labeling.

At this point let us say a few words about the discrete/sliding models used in map labeling. In [12], for each site one has 4 candidate labels (axis-parallel squares each of which has a vertex anchored at the site) and the problem is to select one out of the 4 candidates for each site so as to maximize the labels' size yet make sure that no two labels intersect. (We simply call each of the 4 candidate labels a *discrete label*.) In the past several years more generalized models have been proposed. The basic idea is to allow each site to have an infinite number of possible candidate labels (see [8,13,16,24,26]). This model is more natural than the previous discrete model (like the one in [12]) and has been coined as the *sliding model* in [16]. Certainly, designing efficient algorithms for map labeling under the sliding model becomes a new challenge.

In this paper, we investigate the following map labeling problem under the sliding model. *Given a set S of n sites in the plane, label sites in S with maximum uniform axis-parallel squares, namely how to place n axis-parallel squares of the same size such that no two squares intersect except at their boundary; each square is associated with a point p_i in S such that p_i lies on the boundary of that square; and the size of these uniform squares is maximum.*

In [16] van Kreveld et al. proved the NP-hardness of this problem, i.e., it is NP-hard to decide whether a set of points can all be labeled with axis-parallel unit squares under the sliding model. A careful look at their proof results in a better hardness-result, i.e., it is NP-hard to decide whether a set of points can all be labeled with axis-parallel squares of size greater than 0.75. (Notice that van Kreveld et al. tried to maximize the number of sites labeled instead of the size of the labels in [16].) Clearly, it is meaningful to study approximation algorithms for this problem. In [27], two approximation algorithms, with factor-$5\sqrt{2}$ and factor-4 respectively, were proposed.

Our idea is as follows. We first compute the minimum diameter $D_{5,\infty}(S)$, under the L_∞ metric, of a 5-subset of S and use it to bound the optimal solution l^*. We then design a decision procedure, which, for $L \leq D_{5,\infty}(S)$, decides whether a labeling of S using discrete labels of size $L/2$ exists or not and if $L \leq l^*$ then the answer is always yes. This can be done as follows. We identify all the feasible

regions (to be defined formally) to place a label of size L for p_i. (There are at most 4 such feasible regions for p_i.) We prove that if we shrink all of the optimal sliding labels by a half then we can label points in S with discrete labels at least half of the optimal size at 2 stages: some of them can be labeled at a unique discrete position and will not interact with other labels, others can be labeled at one of the 2 discrete positions such that either one of the candidates of such a point p_i is in the optimal label for p_i or one of the candidates of p_i intersects at most one candidate of another point p_j.

Our detailed algorithm again uses 2SAT. For each point p_i, let $C_i(L)$ be the square that has p_i as its center and is of edge length $L \leq D_{5,\infty}(S)$. We thus have an $O(n)$-size intersection graph $G(L)$ for all $C_i(L), 1 \leq i \leq n$. ($G(L)$ can be constructed in $O(n \log n)$ time.) Consequently, for fixed L whether we can label S using sliding labels of size at least $L/2$ can be determined in $O(n)$ time and moreover; if $L \leq l^*$ then the answer is always positive and the corresponding approximate labeling can be determined also in $O(n)$ time. With this decision procedure, we can either have an $O(n^2 \log n)$ time approximation.

2 Preliminaries

In this section we make some necessary definitions related to our algorithm. The *decision version* of the MLUS-AP problem is defined as follows:

Instance: Given a set S of points (sites) p_1, p_2, ..., p_n in the plane, a real number $l > 0$.
Problem: Does there exist a set of n uniform axis-parallel squares of edge length l, each of which is placed at each input site $p_i \in S$ such that no two squares intersect, a site can be anywhere on the boundary of its labeling square and no site is contained in any square.

This problem is NP-hard [16]. From now on we will focus on the *maximization version* of this problem (i.e., to compute/approximate the optimal solution with size l^*). We say that an approximation algorithm for a (maximization) optimization problem Π provides a *performance guarantee* of ρ if for every instance I of Π, the solution value returned by the approximation algorithm is at least $1/\rho$ of the optimal value for I. (For the simplicity of description, we simply say that this is a factor-ρ approximation algorithm for Π.)

If we allow a small number of sites to be unlabeled then it is possible to obtain a bicriteria polynomial time approximation scheme (PTAS) for this problem [8]. The best known approximation factor is 4 and the running time of the corresponding algorithm is $O(n \log n)$ [27]. In this paper we present an $O(n^2 \log n)$ time, factor-2 approximation algorithm for this problem.

Let $k \geq 2$ be an integer. Given a set S of k points (sites) in the plane, the k-diameter of S under the L_∞-metric is defined as the maximum L_∞-distance between any two points in S. Given a set S of at least k sites in the plane, the

min-k-diameter of S under the L_∞ metric, denoted as $D_{k,\infty}(S)$, is the minimum k-diameter over all possible subsets of S of size k.

In the following section we present an approximation solution for MLUS-AP. We use $D_{5,\infty}(S)$, the min-5-diameter of the set S under L_∞, to bound the optimal solution l^*. Given a set of n sites S, $D_{5,\infty}(S)$ can be computed in $O(n \log n)$ time [6,10].

3 Algorithm

In this section we present the details of an approximation algorithm for the MLUS-AP problem. Let l^* denote the size of each square in the optimal solution of the problem MLUS-AP. For any two points $p_i, p_j \in S$, let $d_\infty(p_i, p_j)$ denote the L_∞-distance between them. We first refer the following fundamental lemma in [27].

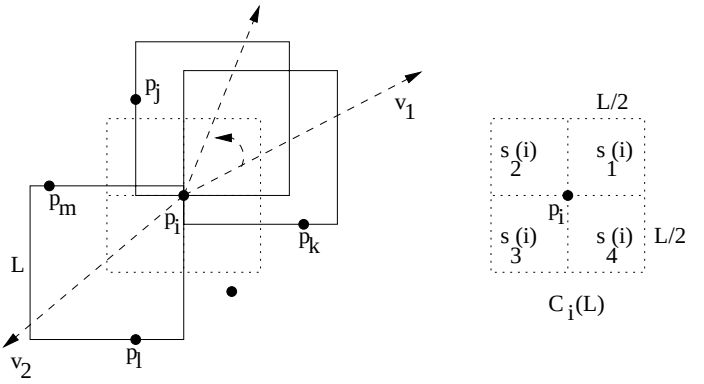


Fig. 2. Maximal feasible regions for p_i.

Lemma 1. *If $|S| \leq 4$, then l^* is unbounded and if $|S| \geq 5$, then $l^* \leq D_{5,\infty}(S)$.*

From now on we assume that $|S| \geq 5$. Let $L \leq D_{5,\infty}(S)$ and let $C_i(L)$ denote the open L_∞-circle centered at point $p_i \in S$ with radius $L/2$. Clearly, the circle $C_i(L)$ contains at most four points from the input set S, including its center. (Otherwise, the five points inside $C_i(L)$ would have a diameter smaller than $L \leq D_{5,\infty}(S)$.) We partition each circle $C_i(L)$ into 4 L_∞-circles with radius $L/4$, which are geometrically squares (see Figure 2). We loosely call them *sub-squares* $s_1(i), s_2(i), s_3(i), s_4(i)$ of $C_i(L)$ which are corresponding to the quadrants originated at p_i in which they lie in. When labeling S with labels of size $L/2$, if a sub-square of $C_i(L)$ has no intersection with any other label then we call such a sub-square *free*. Clearly, a free sub-square of $C_i(L)$ can be a discrete label for p_i.

We define the *maximal feasible region* for labeling $p_i \in S$ using a sliding label of size L as follows. (If $L \leq l^*$ this definition is always valid.) A *maximal feasible region* for a site $p_i \in S$ whose corresponding L_∞-circle $C_i(L)$ contains at most one other point p_j in S is the set of all vectors originated at p_i which pass through the center of a sliding label of size L for p_i. Each such vector corresponds to a maximal feasible *box* with length and height at least L, e.g., in Figure 2 v_1 is corresponding to the box through p_i and p_k. Clearly a maximal feasible region for p_i is determined by two points in S, e.g., p_j and p_k in Figure 2. We call p_j, p_k the *anchor* points of this region. (Note that there are degenerate cases, e.g., in Figure 2 the feasible region for p_i anchored by p_l, p_m contains only one vector v_2.)

In [7] it is shown that in the case of labeling points with uniform approximate circles, every point has at most two maximal feasible regions. This makes it possible to apply 2SAT directly to obtain a factor-3.6 approximation [7]. However, in our case a point $p_i \in S$ can have four maximal feasible regions; therefore, a different method has to be used.

Given $L \leq D_{5,\infty}(S)$ we compute the number of free sub-squares in $C_i(L)$ and the maximal feasible regions for every point $p_i \in S$. (This can be done by computing the points which are within $D_{5,\infty}(S)$ distance to p_i. By the property of $D_{5,\infty}(S)$, for each p_i we have at most 24 such points and they can be computed in $O(n \log n)$ time using the algorithms in [10,6]. If some point p_i has no maximal feasible region then $L > l^*$ and we have to search for a smaller L.) If a point p_i has at most two maximal regions or if $C_i(L)$ has at most two non-empty sub-squares then we can always choose at most two discrete candidates of size $L/2$ for p_i such that at least one of the candidate is contained in an optimal label for p_i with fixed size L.

In a solution which we can label all input sites with labels of size $L \leq l^*$, we call the corresponding label for p_i the *optimal label* for p_i (with size L). The nontrivial part of our algorithm, which is different from [7], is as follows. If p_i has more than two maximal feasible regions, then we can still pick up a pair of discrete labels with size $L/2$ for p_i such that either one of them appears in the optimal label for p_i with size L, or any one of them intersects at most one candidate of another point p_j. This reduces a potential 4SAT problem to a 2SAT problem; in other words, if we simply list all four discrete labels of size $L/2$ for p_i then we would have four candidates for each p_i [27]. The following lemma shows the correctness of the above method.

Lemma 2. *Let p_i have more than two maximal feasible regions and let $L \leq l^*$. We can compute two discrete candidate labels with size $L/2$, $c_1(p_i)$ and $c_2(p_i)$, for p_i such that either one of $c_1(p_i)$ and $c_2(p_i)$ is contained in the optimal label for p_i with size L or $c_1(p_i)$ ($c_2(p_i)$) intersects at most one candidate of another point p_j.*

Proof: We refer to Figure 3, in which p_i has three maximal feasible regions and Figure 3 (a) shows the points around p_i. By the definition of a maximal feasible region, we can have at most four of them and each region can contribute

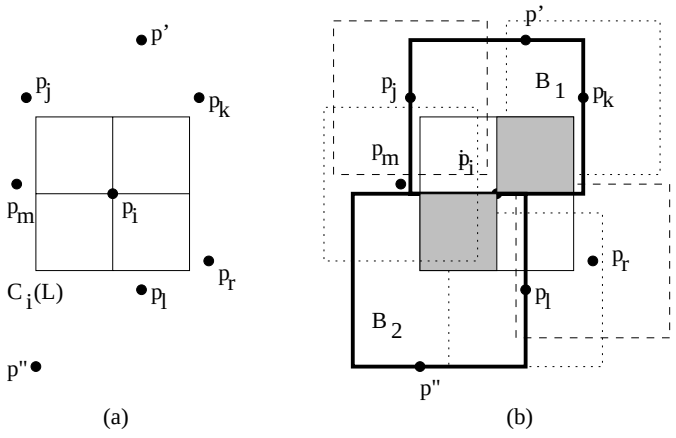


Fig. 3. Illustration for the proof of Lemma 2.

a square of size L. This implies that we must have two maximal feasible boxes B_1, B_2 which do not intersect (Figure 3 (b)). Although we do not know what direction the optimal label for p_i actually goes — this is determined by the whole set of points, not just those local points of p_i, we claim that either one of the two diagonal sub-squares of $C_i(L)$ which are completely contained in $B_1 \cup B_2$ is contained in the optimal label of p_i with size L or one of the candidates of p_i intersects at most one candidate of another point which shares the same maximal feasible box with p_i (e.g., a candidate of p_l in Figure 3). If neither of these two conditions is true then either we cannot label p_i with a label of size L or some point is contained in a maximal feasible box of p_i, both leads to a contradiction. □

In the above lemma, note that once having all of the maximal feasible boxes (regions) of p_i we can easily construct two such labels with size $L/2$, $c_1(p_i)$ and $c_2(p_i)$ (which can be encoded with a binary variable), in constant time. The above lemma, though simple, has a strong implication. We label a point p_i with one of its selected candidates, which might force one of its neighboring points to be labeled using a unique candidate. If a candidate of p_i is inside the optimal label of p_i (with twice of the size) then certainly this candidate can intersect at most one candidate of any other point p_j. Otherwise, i.e., when p_i has 4 maximal feasible regions and we make a wrong decision in choosing p_i's candidates, we still have this property. Of course, we might still have a problem: When p_i has 4 maximal feasible regions and we fail to choose the right candidates for p_i, there might be two cycles through p_i' candidates in the intersection graph $G'(L)$ of all candidates of input sites. (The vertices in $G'(L)$ are candidates of input sites, there is an edge between $c_y(p_i)$ and $c_z(p_j)$ if $c_y(p_i)$ intersects $c_z(p_j)$, $y, z \in \{1, 2\}$.) This can be handled with the following lemma.

Lemma 3. *Let p_i have four maximal feasible regions and let $L \leq l^*$. Let the two discrete candidate labels for p_i with size $L/2$ be $s_1(i)$ and $s_3(i)$ ($s_2(i)$ and $s_4(i)$).*

In $G'(L)$, if we have two cycles associated with the candidates of p_i, then we can update the candidates of p_i as $s_2(i)$ and $s_4(i)$ ($s_1(i)$ and $s_3(i)$) to eliminate both of the cycles.

Therefore, for the 2SAT formula we eventually construct, if $L \leq l^*$, every literal X can only intersect one of Y and $\neg Y$; moreover, there is at most one cycle associated with X and $\neg X$. Clearly, such a 2SAT formula is always satisfiable. The following lemma contains other details we have not described so far and is the basis for our approximation algorithm. For each input site we only need to consider at most two discrete labels as its labeling candidates! This naturally connects this problem, for one more time, to 2SAT.

Lemma 4. *For any $L \leq l^*$, we can label S using discrete labels of size at least $L/2$; moreover, for each site $p_i \in S$ we only need to consider at most two candidates.*

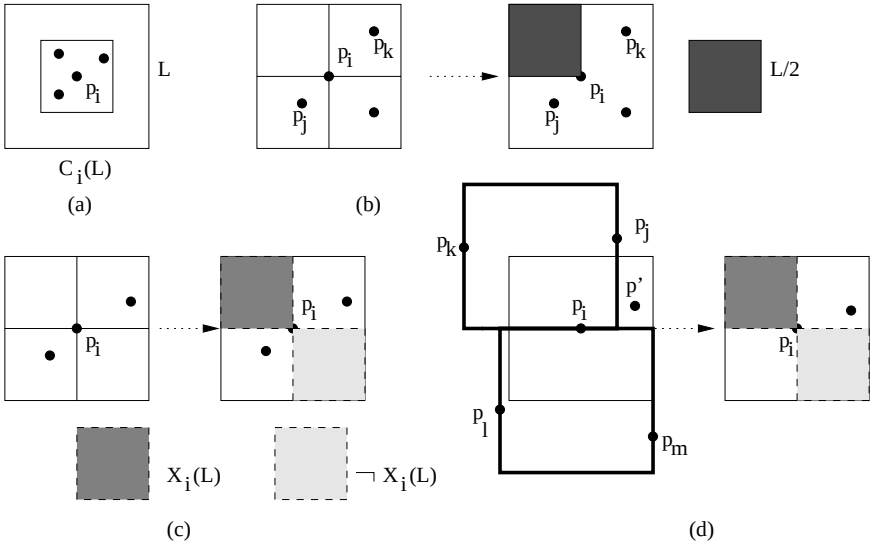


Fig. 4. Illustration for the proof of Lemma 3.

Proof: It is easy to see that given a labeling of S with sliding labels of size L, we can always label S using discrete labels of size $L/2$ for every $p_i \in S$ (i.e., shrinking all sliding labels by a half to a discrete label with size $L/2$). In the following, we exploit the geometric properties of the problem to show that to actually label p_i using a label of size $L/2$ we only need to consider at most two discrete candidates for every p_i.

First of all, remember that $C_i(L)$ contains at most four sites in S including its center p_i. If all four sites are within the L_∞-circle $C_i(L/2)$ then we can label

the 4 sites in a unique way using discrete labels of size $L/2$ without any conflict with other labels (Figure 4 (a)).

Now assume that the above situations have been handled. Because $C_i(L)$ has 4 sub-squares and it contains only 3 other sites different from p_i; by the Pigeonhole Principle, one of the sub-squares of $C_i(L)$ must be empty of other sites in S. If exactly one of the sub-squares of $C_i(L)$ is empty (e.g., the upper-left sub-square $s_2(i)$ in Figure 4 (b)), then it must be free as any label with size L for p_i must contain this sub-square. In this case, we can simply label p_i with $s_2(i)$ as the discrete label.

If exactly two of the sub-squares of $C_i(L)$ are empty (e.g., the upper-left and bottom-right sub-squares $s_2(i), s_4(i)$ in Figure 4 (c)), then one of them must be free. We therefore use two literals $X_i(L)$ and $\neg X_i(L)$ to encode these two sub-squares. (If two adjacent sub-squares are empty, we can handle the situation similarly.)

If either three or four of the sub-squares of $C_i(L)$ are empty (e.g., the case in Figure 4 (d)), then we identify the maximal feasible regions around p_i and by definition if $L \leq l^*$ then such a region must exist.

If p_i has three maximal feasible regions, then we can handle this case using Lemma 2 (Figure 4 (d)). If all four sub-squares of $C_i(L)$ are empty and if p_i has four maximal regions, then we can handle this case using the algorithm summarized in Lemma 2 and Lemma 3. □

We now present our decision procedure which tests whether a labeling of S using squares of size $L/2, L \leq D_{5,\infty}$, under the discrete model exists or not. First we consider the intersection graph $G(L)$ of all $C_i(L)$'s (i.e., $C_i(L)$, $1 \leq i \leq n$, are the vertices for $G(L)$ and there is an edge between $C_i(L), C_j(L)$ if they intersect.) This graph $G(L)$ has two properties: (1) $G(L)$ is planar; and (2) $G(L)$ has maximum vertex degree 7.

We can construct the graph $G(L)$ in $O(n \log n)$ time using standard algorithm, like plane sweep [20]. ($G'(L)$ can be computed from $G(L)$ in $O(n)$ time.) If the sub-square corresponding to the literal $X_i(L)$ intersects another sub-square corresponding to the literal $X_j(L)$, then we will build a 2SAT clause

$$\neg(X_i(L) \wedge X_j(L)) = (\neg X_i(L) \vee \neg X_j(L)).$$

As $G(L)$ is of linear size the eventual 2SAT formula contains at most $O(n)$ such clauses. Following Lemma 4, if $L \leq l^*$ then this 2SAT formula is always satisfiable and finding a truth assignment can be done in $O(n)$ time [11]. We thus have the following lemma.

Lemma 5. *Deciding whether a labeling of S using discrete labels of size $L/2$ exists, $L \leq D_{5,\infty}(S)$, can be done in $O(n)$ time.*

By runing a binary search on $L \in (0, D_{5,\infty}(S)]$ and stop when the eventual search interval is small enough, say δ, we can compute a labeling of S using discrete labels of size at least $l^*/(2+\delta)$ in $O(n \log n + n \log \frac{D_{5,\infty}(S)}{\delta})$ time. We thus have the following theorem.

Theorem 1. *There is an* $O(n \log n + n \log \frac{D_{5,\infty}(S)}{\delta})$ *time, factor-*$(2+\delta)$ *approximation for MLUS-AP.*

Notice that the running time of this algorithm is not fully polynomial. Nevertheless, it is conceptually simple. In practice, we could make the algorithm completely practical by using a heuristic algorithm to approximate $D_{5,\infty}(S)$. If we want to have a fully polynomial time approximation algorithm, we need to identify the possible candidates for l^* which is proved in the following lemma.

Lemma 6. *The candidates for* l^* *include* $d_\infty(p_i, p_j)/K$, $1 \leq K \leq n-1$, *provided that they are bounded above by* $D_{5,\infty}(S)$.

Lemma 6 implies that the number of candidates for l^* is $O(n^3)$. If we compute them out explicitly, sort them and run a binary search of this list (using the decision procedure summarized in Lemma 5), then we would have an $O(n^3 \log n + n \log n) = O(n^3 \log n)$ time approximation. Using a technique reminiscent of Blum et al. [2] on finding a median in linear time and a technique on computing weighted median (page 193 of [3]), we do not have to try all the combinations of i, j, K and we obtain an $O(n^2 \log n)$ time approximation algorithm for MLUS-AP. (An identical technique is used in [9].) Therefore, we have the main theorem of this paper.

Theorem 2. *For any given set* S *of* n *sites in the plane, there is an* $O(n^2 \log n)$ *time, factor-2 approximation for the MLUS-AP problem.*

Notice that in the NP-completeness proof of [12] the optimal solutions of using sliding labels and discrete labels to label the constructed input points are the same. Therefore, if we use discrete labels to approximate MLUS-AP then we cannot obtain a factor better than 2.

Corollary 1. *It is NP-hard to obtain an approximation solution with a factor better than 2 using discrete labels for the MLUS-AP problem.*

Notice that how to reduce the gap between $1.33 - \epsilon$ and 2 for approximating MLUS-AP remains an open problem. From Corollary 1, apparently we have to use sliding labels to improve the factor-2 approximation.

References

1. P. Agarwal, M. van Kreveld and S. Suri. Label placement by maximum independent set in rectangles. *Comp. Geom. Theory and Appl.,* 11:209–218, 1998.
2. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest and R. E. Tarjan. Time Bounds for Selection. *J. of Computer and System Sciences*, 7(4):448–461, 1973.
3. T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to algorithms.* MIT Press, 1990.
4. J. Christensen, J. Marks, and S. Shieber. An Empirical Study of Algorithms for Point-Feature Label Placement, *ACM Transactions on Graphics,* 14:203–222, 1995.

5. J. Doerschler and H. Freeman. A rule-based system for cartographic name placement. *CACM*, 35:68–79, 1992.
6. A. Datta, II.-P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems, *J. Algorithms,* 19:474–503, 1995.
7. S. Doddi, M. Marathe and B. Moret, Point set labeling with specified positions. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 182–190, June, 2000.
8. S. Doddi, M. Marathe, A. Mirzaian, B. Moret and B. Zhu, Map labeling and its generalizations. In *Proc. 8th ACM-SIAM Symp on Discrete Algorithms (SODA'97)*, New Orleans, LA, pages 148–157, Jan, 1997.
9. R. Duncan, J. Qian, A. Vigneron and B. Zhu. Polynomial time algorithms for three-label point labeling, *Invited paper accepted by TCS, a special issue for COCOON'01*, March, 2002.
10. D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes, *Discrete & Comput. Geom.,* 11:321–350, 1994.
11. S. Even, A. Itai and A. Shamir. On the complexity of timetable and multicommodity flow problem. *SIAM J. Comput.*, 5:691–703, 1976.
12. M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 281–288, 1991.
13. C. Iturriaga and A. Lubiw. Elastic labels: the two-axis case. In *Proc. Graph Drawing'97*, pages 181–192, 1997.
14. E. Imhof. Positioning names on maps. *The American Cartographer*, 2:128–144, 1975.
15. C. Jones. Cartographic name placement with Prolog. *Proc. IEEE Computer Graphics and Applications*, 5:36–47, 1989.
16. M. van Kreveld, T. Strijk and A. Wolff. Point set labeling with sliding labels. *Comp. Geom. Theory and Appl.,* 13:21–47, 1999.
17. K. Kakoulis and I. Tollis. An algorithm for labeling edges of hierarchical drawings. In *Proc. Graph Drawing'97*, pages 169–180, 1997.
18. K. Kakoulis and I. Tollis. A unified approach to labeling graphical features. *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 347–356, 1998.
19. D. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM J. Disc. Math.*, 5:422–427, 1992.
20. F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.
21. C.K. Poon, B. Zhu and F. Chin, A polynomial time solution for labeling a rectilinear map. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 451–453, 1997.
22. Z.P. Qin, A. Wolff, Y. Xu and B. Zhu. New algorithms for two-label point labeling, In *Proc. 8th European Symp. on Algorithms (ESA'00),* pages 368–379, Springer-Verlag, LNCS series, 2000.
23. Z.P. Qin, B. Zhu and R. Cimikowski. A simple factor-3 approximation for labeling points with circles, *Submitted to IPL*, April, 2002.
24. T. Strijk and A. Wolff. Labeling points with circles. *Intl. J. Computational Geometry and Applications*, 11(2):181–195, 2001.
25. F. Wagner and A. Wolff. Map labeling heuristics: Provably good and practically useful. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 109–118, 1995.
26. B. Zhu and C.K. Poon. Efficient approximation algorithms for two-label point labeling, *Intl. J. Computational Geometry and Applications*, 11(4):455–464, 2001.
27. B. Zhu and Z.P. Qin. New approximation algorithms for map labeling with sliding labels, *J. Combinatorial Optimization*, 6(1):99–110, 2002.

Optimal Algorithm for a Special Point-Labeling Problem

Sasanka Roy[1], Partha P. Goswami[2], Sandip Das[1], and Subhas C. Nandy[1]

[1] Indian Statistical Institute, Calcutta 700 035, India
[2] Computer Center, Calcutta University, Calcutta 700 009, India

Abstract. We investigate a special class of map labeling problem. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of point sites distributed on a 2D map. A label associated with each point is a axis-parallel rectangle of a constant height but of variable width. Here height of a label indicates the font size and width indicates the number of characters in that label. For a point p_i, its label contains the point p_i at its top-left or bottom-left corner, and it does not obscure any other point in P. Width of the label for each point in P is known in advance. The objective is to label the maximum number of points on the map so that the placed labels are mutually non-overlapping. We first consider a simple model for this problem. Here, for each point p_i, the corner specification (i.e., whether the point p_i would appear at the top-left or bottom-left corner of the label) is known. We formulate this problem as finding the maximum independent set of a chordal graph, and propose an $O(n\log n)$ time algorithm for producing the optimal solution. If the corner specification of the points in P is not known, our algorithm is a 2-approximation algorithm. Next, we develop a good heuristic algorithm that is observed to produce optimal solutions for most of the randomly generated instances and for all the standard benchmarks available in [13].

1 Introduction

Labeling a point set is a classical problem in the geographic information systems, where the points represent cities on a map which need to be labeled with city names. The point set labeling problem finds many important statistical applications, e.g., scatter plot of principal component analysis [6], in spatial statistics where the aim is to post the field measures against the points, etc. The ACM Computational Geometry Impact Task Force report [2] lists label placement as an important research area.

In general, the label placement problem includes positioning labels for area, line and point features on a 2D map. A good labeling algorithm has two basic requirements: the label of a site should touch the site at its boundary, and the labels of two sites must not overlap. Another important requirement is that the label of one site should not obscure the other sites on the map. Many other aesthetic

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 110–120, 2002.

requirements for map labeling are listed in [7]. Given the basic requirements, two major types of problems are considered: (1) label as many sites as possible, and (2) find the largest possible size of the label such that all the sites can be labeled in a non-overlapping manner. In general, both of these problems are NP-hard [4]. In this paper, we shall consider a special case of the first variation of the point-site labeling problem.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the plane. For each point $p_i \in P$, we have a rectangular label r_i of specified size, and a set π_i of marked positions on the boundary of r_i. The label of a point p_i must be placed parallel to the coordinate axes, and must contain p_i on one of the marked positions on π_i. A feasible configuration is a family of axis-parallel rectangles (labels) $R = \{r_{1'}, r_{2'}, \ldots, r_{k'}\}$, where all the $i' \in \{1', 2', \ldots, k'\}$ are different and $r_{i'}$ is represented by a tuple $\{(p_{i'}, x_{i'}) \mid p_{i'} \in P,\ x_{i'} \in \pi_{i'}$ and $r_{i'}$ is placed with $p_{i'}$ at the position $x_{i'}$ on its boundary$\}$, such that the members in R are mutually non-overlapping. The label placement problem is to find the largest feasible configuration [1]. Typical choices of π_i include (i) the end points of the left edge of r_i, (ii) the four corners of r_i, or (iii) the four corners and the center points of four edges of r_i, etc. In [1], an $O(\log n)$-approximation algorithm is proposed for this problem which runs in $O(n\log n)$ time. An α-approximation algorithm produces a solution of size at least $\frac{\Delta}{\alpha}$, where Δ is the size of the optimal solution. In particular, if the labels are of the same height, a dynamic programming approach is adopted to get a $(1 + \frac{1}{k})$-approximation algorithm which runs in $O(n\log n + n^{2k-1})$ time [1]. This case is of particular importance since it models the label placement problem when all labels have the same font size. In [15], a simple heuristic algorithm for the point labeling problem is proposed which is easy to implement and produces near-optimal solution, but the running time is $O(n\log n + k)$, where k may be $O(n^2)$ in the worst case. The point labeling with sliding labels is introduced in [8], where the point p_i can assume any position on the boundary of r_i. The problem has shown to be NP-hard, and using plane sweep paradigm, a 2-approximation algorithm has been presented whose time complexity is $O(n\log n)$. The label placement problem in the slider model has been extended for the map containing several polygonal obstacles, and the objective is to label a set of n point sites avoiding those obstacles [10]. The time complexity of this algorithm is $O((n + m)\log(n + m))$, where m and n are respectively the total number of vertices of all the polygons, and the number of point sites to be labeled. In [11], a decision theoretic version of the map labeling problem is introduced where the sites are horizontal and vertical line segments. Each label has unit height and is as long as the segment it labels. The problem is to decide whether it is possible to label all the sites. The problem is transformed to the well known 2-SATISFIABILITY problem, and an algorithm for this decision problem is proposed which runs in $O(n^2)$ time. Later, in [9], the time complexity was improved to $O(n\log n)$. Good heuristics are proposed for labeling arbitrary line segments and polygonal areas [3].

In our model, the labels of the points on the map are axis-parallel rectangles of a constant height (h) but of variable width. The width (w_i) of the label of

a point p_i is pre-specified. The point p_i appears either on the top-left or the bottom-left corner of its label. A label said to be *valid* if it does not contain any other point(s) of P. Thus, for each point, it may not have any valid label, or it can have one valid label or it can have two valid labels. We consider the following two variations of the problem:

P1: For each point $p_i \in P$, the corner specification (i.e., whether p_i would appear at the top-left or bottom-left corner of the label) is known.

P2: The corner specifications of the points in P are not known.

Problem P1 is modeled using maximum independent set of a chordal graph, and an $O(n\log n)$ time algorithm is proposed which produces optimum solution. A minor modification of this algorithm is proved to be a 2-approximation algorithm for problem P2. Finally, we propose an efficient heuristic algorithm for the problem P2. This heuristic is tested on several randomly generated examples and the standard benchmarks [13]. In most of the randomly generated examples, and for all the benchmarks it produces optimum solution. Surely, we have encountered few random instances where it fails to produce optimum solution. However, for all instances, we have tried, our algorithm outputs better result than the algorithm presented in [1].

2 Problem P1

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane. Each point p_i is associated with a label r_i which is a closed region bounded by an axis-parallel rectangle. We assume that the heights of all r_i are same, but their width may vary. The placement of label r_i must coincide with the point p_i at either of its top-left and bottom-left corners which is specified. A label r_i is said to be *valid* if it does not contain any point $p_j \in P(j \neq i)$ in its interior. In Fig. 1, we demonstrate the valid and invalid labels (using solid and dashed rectangles, respectively). We construct a graph, called *label graph* $LG = (V, E)$, whose set of vertices (V) correspond to the valid labels of the points in P. An edge $e \in E$ connects a pair of nodes v_i, v_j if their corresponding labels have a non-empty intersection. In the worst case, $|V| = n$ and $|E| = O(n^2)$. Our problem is to find the largest subset $P' \in P$ such that valid labels corresponding to the members of P' are mutually non-overlapping. We show that the above problem reduces to finding the maximum independent set of the label graph LG. In the next section, we mention some important characterization of the label graph.

2.1 Some Useful Results

Definition 1. *[5] An undirected graph is a chordal graph if and only if every cycle of length greater than or equal to 4 possesses a chord.*

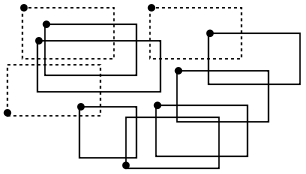


Fig. 1. Examples of valid and invalid labels

Definition 2. *[5] A graph $G = (V, E)$ is said to have a transitive orientation property if each edge $e \in E$ can be assigned a one-way direction in such a way that the resulting oriented graph $G' = (V, F)$ satisfies the following condition: $ab \in F$ and $bc \in F$ imply $ac \in F$ for all $a, b, c \in V$. Here F denotes the set of oriented edges obtained from the set of edges E of the graph G.*

Definition 3. *[5] The intersection graph of a family of intervals on a real line is called an interval graph.*

Fact 1 *[5] A chordal graph is an interval graph if and only if the complement of the graph has a transitive orientation.* □

Fact 2 *[5] Every chordal graph is a perfect graph.* □

From now onwards, we use $\mathcal{IG}$, $\mathcal{LG}$ and $\mathcal{CG}$ to denote the classes of interval graphs, label graphs and chordal graphs, respectively.

Lemma 1. *The intersection graph of a set of valid labels is a chordal graph, but the converse may not always be true.*

Proof: Let LG be a label graph which contains a chordless cycle C of length greater than or equal to 4 (see Fig. 2(a)). Since the left edge of r is rightmost, either s or t must contain the point that r labels (since labels are of same height). This contradicts the validity of r. For the second part, consider the chordal graph in Fig. 2(b). It can be easily shown that it is impossible to place points corresponding to all the vertices of that graph such that the labels (with any arbitrary size and the corner specification) of all the points are valid. □

It can be easily shown that any interval graph is a label graph, but the converse is not true. This leads to the following theorem.

Theorem 1. $\mathcal{IG} \subset \mathcal{LG} \subset \mathcal{CG}$. □

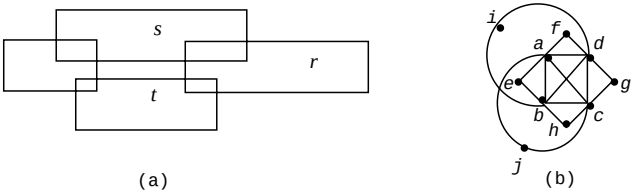


Fig. 2. Proof of Lemma 1

2.2 Algorithm

We propose an efficient algorithm for finding the placement of maximum number of labels for the points in P. For each point $p_i \in P$, the size of its label r_i, and the corner specification (top-left/bottom-left) of point p_i on r_i is already known. For each label, we check its *validity* (whether it obscures any other points or not) by searching a 2-d tree [12] with the set of points in P. This step requires $O(n\log n)$ time in total for all the labels.

Let $R = \{r_1, r_2, \ldots, r_N\}$ $(N \leq n)$ be a set of valid labels placed on the plane. The traditional line sweep technique may be used to construct the label graph LG in $O(n\log n + |E|)$ time. Our objective is to find the maximum independent set of LG, denoted by $MIS(LG)$. As LG is a perfect graph (see Lemma 1 and Fact 2), we define the perfect elimination order (PEO) among the vertices of the graph as follows:

Definition 4. *[5] A vertex v of a graph LG is a simplicial vertex if its adjacency set $Adj(v)$ induces a complete subgraph of LG.*

Definition 5. *[5] Let $\sigma = \{v_1, v_2, \ldots v_n\}$ be an ordering of vertices of LG. We say that σ is a perfect elimination order (PEO) if each v_i is a simplicial vertex of the induced subgraph with the set of vertices $\{v_i, \ldots, v_n\}$. In other words, each set $X_i = \{v_j \in Adj(v_i) | j > i\}$ is a complete graph.*

Lemma 2. *If v_i (corresponding to point p_i) is a simplicial vertex in LG, then there exists an optimal solution of the problem P1 containing the label r_i.*

Proof: [By contradiction] Let v_i be a simplicial vertex, and it does not appear in the optimum solution. Let $V_i = \{v_{i1}, v_{i2}, \ldots, v_{ik}\}$ be the set of vertices adjacent to v_i. Now we need to consider two cases: (i) none of the vertices in V_i appears in $MIS(LG)$, and (ii) one member, say v_{ij} of V_i appears in $MIS(LG)$. Case (i) is impossible since we can include v_i in $MIS(LG)$ as it does not intersect with any one of the existing members of $MIS(LG)$. In case (ii), no member of V_i except v_{ij} is present in $MIS(LG)$. Thus, we can replace v_{ij} by v_i in $MIS(LG)$; the updated set ($MIS(LG)$) will remain maximum independent set of LG. □

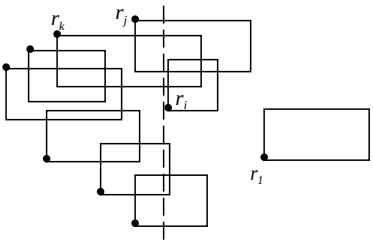


Fig. 3. Proof of Lemma 3

Lemma 3. *Let R be a set of valid labels. The sorted order of the left boundaries of R from the right to the left, gives a PEO of the graph LG.*

Proof. Let $L = \{\lambda_1, \lambda_2, \ldots, \lambda_N\}$ be the left boundaries of the labels in R, and let L^* denote the sorted sequence of the members in L in a right-to-left order. Consider the left boundary λ_i of a label r_i. Let R' be a subset of R such that the left boundaries of all the members in R' appear after λ_i in L^*, and all of them intersect r_i. We need to prove that $R' \cup \{r_i\}$ forms a clique. As the placement of r_i is valid, either all the members of R' contain the top-left corner of r_i or all of them enclose the bottom-left corner of r_i (in Fig. 3, r_j and r_k encloses the top-left corner of r_i). In other words, the corresponding point $p_i \in P$ is present either at the bottom-left corner of r_i or at the top-left corner of r_i. Hence all the members in $R' \cup \{r_i\}$ have a common region of intersection. □

Theorem 2. *The PEO of a label graph with n vertices can be obtained in $O(n\log n)$ time.*

Proof. Follows from Lemma 3. □

Let σ be a PEO for the graph $LG = (V, E)$. We define inductively a sequence of vertices $y_1, \ldots, y_t$ in the following manner: $y_1 = \sigma(1)$; y_i is the first vertex in σ which follows y_{i-1}, and which is not in $\{X_{y_1} \cup X_{y_2} \cup \ldots \cup X_{y_{i-1}}\}$, where $X_v = \{x \in \text{Adj}(v) \mid \sigma^{-1}(v) < \sigma^{-1}(x)\}$. Hence, all the vertices following y_t are in $X_{y_1} \cup X_{y_2} \cup \ldots \cup X_{y_t}$, and $V = \{y_1, \ldots, y_t\} \cup X_{y_1} \cup X_{y_2} \cup \ldots \cup X_{y_t}$.

Theorem 3. *The vertices $\{y_1, .., y_t\}$ forms a maximum independent set in LG.*

Proof. Follows from Theorem 1 and Theorem 4.18 of [5]. □

The maximum independent set of LG can be obtained in $O(|V| + |E|)$ time [5]. We now show that if the placement of the labels corresponding to the vertices of LG is available in the plane, then a maximum independent set of LG can be determined in a faster way by simply sweeping the plane with a vertical line from right to left.

Algorithm MIS (* for finding the maximum independent set of LG *)

Input: An array L containing the line segments corresponding to the left and the right boundaries of all the valid labels $r_i \in R$. An element representing a right boundary of a label has a pointer to its left boundary. With each element of L, a *flag* is stored and is initialized to 0. During execution, the *flag* of an element is set to 1 if it is selected as a member of $MIS(LG)$ or if its corresponding label overlaps on a label of $MIS(LG)$ under construction.
Output: A maximum independent set (*MIS*) of the intersection graph of R.
Preprocessing: We initialize an interval tree $\mathcal{T}$ [12] with the y-coordinates of the top and bottom boundaries of the labels in R. $\mathcal{T}$ is used for storing the vertical intervals of those labels that the sweep-line currently intersects, and does not overlap with any of the existing members in *MIS*.
Step 1: Sort the array L with respect to the x-coordinates of its elements in decreasing order.
Step 2: Process the elements of the array L in order.
Let $I \in L$ be the current element.
If *flag* of I is 1, then ignore I; otherwise perform the following steps.
2.1: If I corresponds to the right boundary of a label, then insert I in $\mathcal{T}$.
2.2: Otherwise (* if it corresponds to the left boundary of a label *)
2.2.1: Insert the label r (corresponding to I) in *MIS*.
2.2.2: (* Note that I is currently in $\mathcal{T}$ *)
Search $\mathcal{T}$ to find the set $X_r = \{J \mid J \in \mathcal{T}$ and J overlaps with $I\}$.
2.2.3: (* Note that $I \cup X_r$ form a clique. In other words, the set of labels corresponding to $I \cup X_r$ are mutually overlapping *)
For each member in X_r (* representing the right boundary of a label *) the *flag* bit of its corresponding left boundary element is set to 1.
2.2.4: Finally, remove all the intervals in $I \cup X_r$ from $\mathcal{T}$.
Step 3: Report the elements of the array *MIS*.

Theorem 4. *Algorithm* **MIS** *computes the maximum independent set of the label graph LG in* $O(n\log n)$ *time.*

Proof. The PEO of the graph LG is obtained from the right to left sweep on the plane (see Theorem 2). When a label is selected as a member in the MIS, its adjacent labels are discarded by setting 1 in their *flag* bit. Now by Theorem 3, the correctness of the algorithm follows. Next, we discuss the time complexity of the algorithm.

The initial sorting requires $O(n\log n)$ time. A vertical interval corresponding to each label is inserted once in $\mathcal{T}$. After placing a label, say r_i, in the *MIS* array, it is deleted from $\mathcal{T}$. The set of labels X_{r_i}, whose corresponding vertical intervals are in $\mathcal{T}$ and which overlap on r_i, are recognized in $O(k_i + \log n)$ time. These intervals are deleted from $\mathcal{T}$, so that none of them will be recognized further. So, for every pair of elements $r_i, r_j \in MIS$, $X_{r_i} \bigcap X_{r_j} = \phi$, and if *MIS* contains t elements then $\sum_{i=1}^{t} k_i < n$. Each insertion/deletion in $\mathcal{T}$ requires $O(\log n)$ time. Thus the proof of the result. □

3 Problem P2

As in the problem P1, here also the point p_i may appear either of the top-left and bottom-right corners of r_i, and for each point $p_i \in P$, the size of its label r_i is given, but unlike problem P1, the corner specification of the label r_i is not known in advance. Thus, a point p_i may not have any valid label, or it may have one or two valid labels. If a point p_i has two valid labels, say r_i and r'_i, they have an edge in the label graph LG. Fig. 4 shows that, in this case, the label graph may contain cycle(s) of length ≥ 5; so it may not always be a perfect graph. Thus, the earlier algorithm may not produce optimum result. We first prove that a minor modification of our algorithm MIS produces a 2-approximate solution of problem P2. Next, we present an efficient heuristic algorithm for the problem P2.

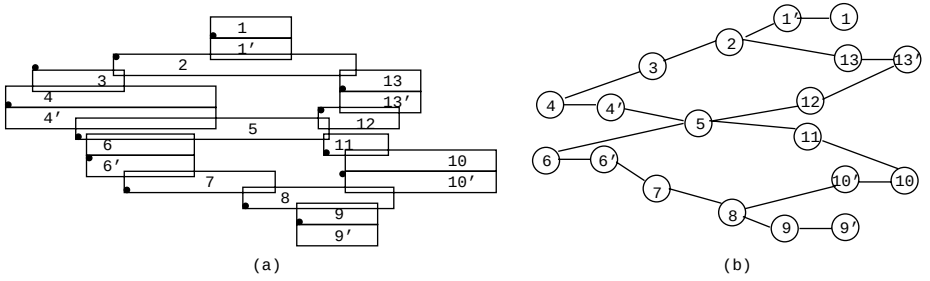


Fig. 4. The label graph corresponding to problem P2 is not perfect

3.1 2-Approximation Result

Let R be the set of all valid labels. During the right to left scan, let r_i and r'_i be a pair of valid labels (corresponding to point p_i) which are currently encountered by the sweep line. Prior to this instant of time some labels are already selected for solution, and for these selections, some labels are removed from the set of valid labels by setting their *flag* bit to 1. Let R^* denote the set of valid labels whose *flag* bit contain 0 at the current instant of time, and LG^* denotes the corresponding label graph, and $OPT(R^*)$ is the set of valid labels corresponding to $MIS(LG^*)$. It is easy to show that there must exist an optimal solution containing either of r_i and r'_i. We select any one (say r_i) of them arbitrarily in our modified algorithm. Let R_i and R'_i denote the set of labels adjacent to r_i and r'_i respectively.

Lemma 4. $1 + \#(OPT(R^* \setminus R_i)) \leq \#(OPT(R^*)) \leq 2 + \#(OPT(R^* \setminus R_i))$, *where* $\#(A)$ *indicates the size of set* A.

Proof: The first part of the lemma is trivial. For the second part, consider the following argument.

If $r_i \in OPT(R^*)$, then $OPT(R^*) = \{r_i\} \bigcup OPT(R^* \setminus R_i)$. So the lemma follows in this case.

If $r'_i \in OPT(R^*)$ then $OPT(R^*) = \{r'_i\} \bigcup OPT(R^* \setminus R'_i)$.
Again, $(R^* \setminus R'_i) = (R^* \setminus \{R_i \bigcup R'_i\}) \bigcup (R_i \setminus \{r_i, r'_i\})$.
Thus, $\#(OPT((R^* \setminus \{R_i \bigcup R'_i\}) \bigcup (R_i \setminus \{r_i, r'_i\}))) \leq \#(OPT(R^* \setminus \{R_i \bigcup R'_i\})) + \#(OPT(R_i \setminus \{r_i, r'_i\}))$.

Again, $\#(OPT(R_i \setminus \{r_i, r'_i\})) = 1$ since $R_i \setminus \{r_i, r'_i\}$ forms a clique. Hence the lemma follows. □

If $r_i \in OPT(R)$, and we choose r'_i we lead to a non-optimal solution. But Lemma 4 says that the maximum penalty for doing a wrong choice is at most one. If k choices are made during the entire execution of MIS, and all the choices are wrong, the size of $OPT(R)$ is at most $2k$. Thus we have the following theorem:

Theorem 5. *If the ties are resolved arbitrarily, then the size of the solution obtained by the algorithm MIS is no worse than* $\frac{1}{2} \times OPT(R)$.

3.2 Heuristic Algorithm

The key idea of our heuristic is as follows:

1. At each step, locate a simplicial vertex of the label graph.
2. If such a vertex found then
   Select it as a member of the maximum independent set.
3. else (* the rightmost point p_i has a pair of valid labels, say r_i and r'_i *)
   Select the vertex corresponding to any one label (say r_i) of p_i arbitrarily.
4. Remove the selected vertex and all its adjacent vertices from the graph by setting their *flag* bit.
5. Repeat steps 1 to 4 until the *flag* bit of all the vertices are set to 1.

The algorithm is implemented by sweeping two horizontal lines I_1 and I_2 and two vertical lines J_1 and J_2 simultaneously. I_1 (resp. I_2) is swept from the top (resp. bottom) boundary to the bottom (resp. top) boundary, and J_1 (resp. J_2) is swept from the left (resp. right) boundary to the right (resp. left) boundary. We explain the sweeping of the vertical line J_2. The sweeping of the other lines are done in a similar manner.

We maintain an interval tree $\mathcal{T}$ with the y-coordinates of the end points of the vertical boundaries of n valid rectangles. The secondary structure, attached with each node of the interval tree, is a pair of height balanced trees, say AVL_t and AVL_b. They store the set of intervals, which are attached to this node during the sweep. To insert an interval ℓ in the interval tree, we start traversing from the root with the interval ℓ. As soon as we reach a node whose discriminant value

lies inside ℓ, we attch ℓ with that node. In other words, its top (resp. bottom) end point is inserted in AVL_t (resp. AVL_b).

During the sweep, when J_2 encounters a right boundary of a valid label, the corresponding interval is stored in the appropriate node of the interval tree $\mathcal{T}$ and sweep continues. When a left boundary is faced by J_2, the sweep halts for searching with the corresponding interval, say ℓ, in the interval tree to find the set of other intervals (present in the interval tree) which overlap on ℓ. If all these intervals are mutually overlapping, then the vertex of LG corresponding to the label having the above left boundary, is simplicial.

At each step I_1 (resp. I_2) proceeds until a bottom (resp. top) boundary of a valid rectangle is faced, and J_1 (resp. J_2) proceeds until a left (resp. right) boundary is faced. If any of this scan returns a simplicial vertex v_i, the corresponding label r_i is inserted in MIS. Otherwise, the label which is obtained by J_2 (in right to left scan) is selected for insertion in MIS. The vertex v_i, and all the vertices whose labels overlap on r_i are marked by setting their *flag* bit to 1. The corresponding intervals are removed from all the four interval trees if they are present there. The process is repeated until the *flag* bit of all the vertices are set to 1.

The insertion and deletion of an interval in the interval tree takes $O(\log n)$ time. Consider the processing of a left boundary ℓ. Let Π, the path from root to the node π whose discriminant is contained in ℓ, and Π_1 (resp. Π_2) be the path from π to the leaf node containing the left (resp. right) end point of ℓ. We spend $O(\log n)$ time for finding the paths Π, Π_1 and Π_2. All the intervals stored with the node π overlap on ℓ. We first find the common intersection region among the intervals stored with π. Next, we inspect the secondary structure of each node on Π, Π_1 and Π_2, to find the intervals which are attached to that node and which overlap on ℓ. If all these intervals have a common intersection region, then ℓ contributes a simplicial vertex. Thus, the overall time complexity is $O(n \log n)$. Surely, the space complexity is $O(n)$.

3.3 Experimental Results

We executed this algorithm on many randomly generated examples and on all the benchmark examples available in [13]. In most of the examples, at each step we could locate a simplicial vertex, which leads to an optimum solution. It needs to mention that we have also encountered few random instances where our algorithm could not produce optimum solution. For example, in Fig. 4, the optimum solution is $\{1, 3, 4', 6, 7, 9, 10', 11, 12, 13\}$, whereas our algorithm returns $\{1, 3, 4', 6', 8, 9', 10, 12, 13\}$. We have also compared our result with the labeling algorithm suggested in [1] (see Table 1). The algorithm of [1] assumes that the label of a point p_i may contain p_i at one of its four corner. Thus, each point may have at most four valid labels. In spite of this flexibility, it is observed that our proposed algorithm can label more points than the algorithm proposed in [1].

Table 1. Experimental results on the benchmarks cited in [13]

Examples	No. of sites	height (pixels)	Optimum solution	Our algorithm		Algorithm [1]	
				No. of valid labels	No. of points labeled	No. of valid labels	No. of points labeled
Tourist shops in Berlin	357	4	216	401	216	799	165
		5	206	389	206	769	166
German railway stations	366	4	304	569	304	1133	258
		5	274	513	274	1030	243
American cities	1041	4	1036	2048	1036	4095	859
		5	1031	2027	1031	4053	878
Drill holes in Munich	19461	1000	***	27156	13895	54325	13730
		5000	***	10107	4737	20197	4678

Acknowledgment. We are thankful to Dr. Alexander Wolff for providing us the benchmark examples.

References

1. P. K. Agarwal, M. van Kreveld and S. Suri, *Label placement by maximum independent set in rectangles*, Computational Geometry: Theory and Applications, vol. 11, pp. 209-218, 1998.
2. B. Chazelle et at, Application challenges to computational geometry: CG impact task force report, http://www.cs.princeton.edu/čhazelle/taskforce/CGreport.ps, 1996.
3. S. Edmondson, J. Christensen, J. Marks, and S. Shieber, *A general cartographic labeling algorithm*, Cartographica, vol. 33, no. 4, pp. 13-23, 1997.
4. M. Formann and F. Wagner, *A packing problem with applications to lettering of maps*, Proc. 7th. Annual ACM Symp. on Computational Geometry, pp. 281-288, 1991.
5. M. C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, NY, 1980.
6. E. H. Isaaks and R. M. Srivastava, An Introduction to Applied Geostatistics, Oxford University Press, New York, 1989.
7. E. Imhof, *Positioning names on maps*, The American Cartographer, vol. 2, no. 2, pp. 128-144, 1975.
8. M. van Kreveld, T. Strijk and A. Wolff, *Point labeling with sliding labels*, Computational Geometry: Theory and Applications, vol. 13, pp. 21-47, 1999.
9. T. Strijk and M. van Kreveld, *Labeling a rectilinear map more efficiently*, Information Processing Letters, vol. 69, pp. 25-30, 1999.
10. T. Strijk and M. van Kreveld, *Practical extension of point labeling in the slider model*, 7th. Int. Symp. on Advances in Geographical Information Systems, (ACM-GIS '99), pp. 47-52, 1999.
11. C. K. Poon, B. Zhu and F. Chin, *A polynomial time solution for labeling a rectilinear map*, Proc. 13th. ACM Symp. on Computational Geometry, pp. 451-453, 1997.
12. F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, Springer, Berlin, 1985.
13. A. Wolff, General Map Labeling Webpage, http://www.math-inf.uni-greifswald.de/map-labeling/general/.
14. F. Wagner, A. Wolff, *A practical map labeling algorithm*, Computational Geometry: Theory and Applications, vol. 7, pp. 387-404, 1997.
15. F. Wagner, A. Wolff, V. Kapoor and T. Strijk, *Three rules suffice for good label placement*, Algorithmica, vol. 30, pp. 334-349, 2001.

Random Arc Allocation and Applications*

Peter Sanders and Berthold Vöcking

Max Planck Institut für Informatik
Saarbrücken, Germany
[sanders,voecking]@mpi-sb.mpg.de

Abstract. The paper considers a generalization of the well known random placement of balls into bins. Given n circular arcs of lengths α_1, ... ,α_n we study the maximum number of overlapping arcs on a circle if the starting points of the arcs are chosen randomly. We give almost exact tail bounds on the maximum overlap of the arcs. These tail bounds yield a characterization of the expected maximum overlap that is tight up to constant factors in the lower order terms. We illustrate the strength of our results by presenting new performance guarantees for several application: Minimizing rotational delays of disks, scheduling accesses to parallel disks and allocating memory to limit cache interference misses.

1 Introduction

Randomly assigning tasks or data to computational resources has proved an important load balancing technique for many algorithmic problems. The model we study here was motivated by three different applications one of the authors became involved with where the lack of an accurate analysis of such a simple system became an obstacle. Section 3 gives more details on these applications: minimizing rotational delays in disk scheduling, scheduling parallel disks, and memory allocation to limit cache conflicts. The common theme there is that jobs (disk blocks, strings of disk blocks, memory segments) have to be allocated to a contiguous resource that wraps around (disk tracks, striped disks, memory locations mod cache size). In all three applications randomization is used to make worst case situations unlikely.

The following model describes all three applications. It is so simple that we expect further applications. An *arc allocation* describes the arrangement of n circular arcs that are pieces of a *unit circle*, i.e., a circle with circumference 1. We represent points on this circle by numbers from the half open interval $[0, 1)$. Let $0 \leq a_i < 1$ denote the starting point of *left endpoint* of arc i and α_i the *arc length* of arc i. Arc i spans the half open interval $[a_i, a_i + \alpha_i \bmod 1)$ where $x \bmod 1$ denotes the fractional part of x and where $[a, b)$ for $a > b$ denotes the set $[a, 1] \cup [0, b)$ in this paper. Let $\alpha = \sum_i \alpha_i / n$ denote the *average arc length*. If all arc length are identical, $\alpha_i = \alpha$ for all $0 \leq i < n$, we have a *uniform* arc allocation. In a *random* arc allocation, the starting points are chosen

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 121–130, 2002.

independently and uniformly at random. Let $L(x)$ denote the number of arcs containing x. Let $L = \sup_{x\in[0,1]} L(x)$ denote the *maximum overlap* of an arc allocation. In this paper, we estimate the expectation $\mathbf{E}[L]$ of the maximum overlap and derive tail bounds for L.

Let us go back to the interpretation of arcs as jobs to be executed/allocated. The maximum overlap L is important because all jobs can be executed using between L and $L+1$ trips around the circle if all jobs are known in advance (see Section 3.1). Furthermore, there is a natural online allocation algorithm that needs at most $2L$ trips around the circle (see Section 3.3).

1.1 New Results for Random Arc Allocation

In this paper we present almost exact tail bounds on the maximum overlap for random arc allocation. These tail bounds yield a complete characterization of the expected maximum overlap. Let $\Delta = L - \alpha n$ denote the difference between maximum and average load. We are able to describe $\mathbf{E}[\Delta]$ almost exactly in terms of Lambert's W function [5]. This function is discussed in more detail below. The tail bounds imply the following estimates for $\mathbf{E}[\Delta]$.

A) If $\alpha \leq \frac{\ln n}{n}$ then $\mathbf{E}[\Delta] = \mathcal{O}\left(\alpha n \exp\left(\mathrm{W}\left(\frac{\ln(1/\alpha)}{\alpha n}\right)\right)\right)$.

B) For $\alpha \in [\frac{\ln n}{n}, \frac{1}{2}]$, $\mathbf{E}[\Delta] = \mathcal{O}\left(\sqrt{\alpha n \ln\left(\frac{1}{\alpha}\right)}\right)$.

C) For $\alpha \in [\frac{1}{2}, 1 - \frac{\ln n}{n}]$, $\mathbf{E}[\Delta] = \mathcal{O}\left(\sqrt{(1-\alpha) n \ln\left(\frac{1}{1-\alpha}\right)}\right)$.

D) If $\alpha \geq 1 - \frac{\ln n}{n}$ then $\mathbf{E}[\Delta] = \mathcal{O}\left((1-\alpha)n \exp\left(\mathrm{W}\left(\frac{\ln(1/(1-\alpha))}{(1-\alpha)n}\right)\right)\right)$.

Our estimates on $\mathbf{E}[\Delta]$ in all four cases are essentially tight in the sense that they describe the case of uniform arc length exactly up to constant factors. Observe that the cases D) and C) are symmetric to the cases A) and B) in α and $1-\alpha$, resp. In fact, these bounds are derive using a simple symmetry arguments treating holes (i.e., the uncovered pieces of the circle) like arcs. In case A) it holds $\mathbf{E}[L] = \mathbf{E}[\Delta]$. Exact estimates for this case can be derived relatively easily using Chernoff bounds. More interesting is case D. To obtain tight estimates in this case, we need to combine Chernoff bounds with a random walk analysis.

Now let us come to a discussion of Lambert's W function. This function is defined to be the unique positive solution to the equation $\mathrm{W}(x) \cdot \exp(\mathrm{W}(x)) = x$ for $x > 0$. Of particular interest for us is the function $\exp(\mathrm{W}(x))$. Asymptotically, this function can be estimated by $\lim_{x\to\infty} \exp(\mathrm{W}(x)) = x/\ln(x)$. For example, consider the estimate of $\mathbf{E}[\Delta]$ for the subcase that arcs are very short, say $\alpha = \mathcal{O}(1/n)$. In this case, the characterization above gives $\mathbf{E}[L] = \mathbf{E}[\Delta] = \mathcal{O}((\ln n)/\ln\ln n)$. Furthermore, if $\alpha = \Theta(\frac{\log n}{n})$ then we obtain $\mathbf{E}[L] = \mathbf{E}[\Delta] = \mathcal{O}((\ln n)/\ln\ln n)$.

Finally, in some applications there might be arcs wrapping around the circle several times, i.e., $\alpha_i > 1$. Clearly, in this case our bounds for $\mathbf{E}[\Delta]$ transfer

immediately when using $\alpha' = \frac{1}{n}\sum_i \alpha_i \bmod 1$ instead of α. In Section 2 we prove these bounds for the uniform case. The generalization to variable arc lengths is deferred to the full paper.

1.2 Results for Chains-into-Bins

Many applications in computer science also require a discrete variant of arc allocation where the circle is subdivided into M equal bins and where arc end points are multiples of a bin size. We note that our proof techniques and hence our upper bounds directly transfer to this discrete model. Observe that discrete arc allocation is equivalent to the following *chains-into-bins* problem: $N = \sum_i \alpha_i$ balls connected into n chains are allocated to M bins that are arranged in a circle. A chain is allocated by throwing its first ball into a bin chosen independently, uniformly at random and by putting its remaining balls into adjacent bins in a round robin fashion. In this notation, the bounds in A) and B) become

$$\mathbf{E}[L^{(\mathrm{cb})}] = \Theta\left(\frac{N}{M}\, W^*\!\left(\frac{\ln\left(M\frac{n}{N}\right)}{N/M}\right)\right) \text{ if } N \geq M \ , \tag{1}$$

$$\mathbf{E}[\Delta^{(\mathrm{cb})}] = \Theta\left(\sqrt{\frac{N}{M}\ln\left(M\frac{n}{N}\right)}\right) \text{ if } N = \Omega(M\ln(Mn/N)) \text{ and } \alpha \leq M/2 \tag{2}$$

where $L^{(\mathrm{cb})}$ is the number of balls in the fullest bin and $\Delta^{(\mathrm{cb})} = L^{(\mathrm{cb})} - N/M$. In the way one can translate the results in the cases C) and D).

Let us compare our results for chains-into-bins to the well known results for balls-into-bins processes. These processes are among of the most intensively studied stochastic processes in the context of algorithm analysis (e.g., [10,17,12]). The simplest balls-into-bins process assumes that N balls are placed at random into M bins [10,17]. Balls-into-bins are the special case of chains-into-bins where all chains consist of a single ball, i.e., $n = N$. We get

$$\mathbf{E}[L^{(\mathrm{bb})}] = \Theta\left(\frac{N}{M}\, W^*\!\left(\frac{\ln M}{N/M}\right)\right) \text{ if } N \geq M \ , \tag{3}$$

$$\mathbf{E}[\Delta^{(\mathrm{bb})}] = \Theta\left(\sqrt{\frac{N}{M}\ln M}\right) \text{ if } N = \Omega(M\ln M) \ . \tag{4}$$

The Bounds (3) and (4) are well known although other papers [10,17,13] use a different, slightly more complicated notation that yields more information about constant factors.

Another instructive perspective is that arc allocations are related to balls-into-bins systems with $1/\alpha$ bins. Our analyses for L and Δ will give further insights into the relationship between the two different random processes.

1.3 Previous Results

Barve et al. [2] introduce the chains-into-bins problem and show why several tail bounds for the case $N = n$ also apply to the general case. Apparently, $\mathbf{E}[L^{(\mathrm{cb})}]$

can only grow if chains are atomized into individual balls (although this is not proven yet). Our bounds improve these results by showing that $\Delta^{(\mathrm{cb})}$ can be much smaller if $n \ll N$, i.e., if chains are long.

Chains-into-bins have been analyzed asking what is the expected number of bins with at least a balls [11]. This measure was needed to estimate the number of cache misses in executing a class of cache-efficient algorithms. Refer to Section 3.3 for more details.

Arc allocations have been studied in mathematics under the aspect of when the arcs cover the circle (e.g., [15]). This is related to the minimum overlap which seems to be more important for most computer science applications. We have adopted the convention from these papers to measure arc lengths between 0 and 1 rather than 0 and 2π in order to avoid notational overhead.

An arc allocation defines a *circular arc graph* [9,8] with n nodes where there is an edge between nodes i and j if the corresponding arcs overlap. A set of overlapping arcs defines a clique of the circular arc graph. In this terminology, we are studying the size of the maximum overlap clique of a random circular arc graph. But note that the maximum overlap clique is not necessarily maximum clique of a circular arc graph [3].

2 Uniform Arcs

In this section we assume that all arcs have the same length. The following tail bound imply the expectation Bounds A) and B) respectively.

Theorem 1. *Suppose n arcs of length $\alpha \leq \frac{1}{2}$ are placed at random onto the unit circle. Let $\mu \geq \alpha n$ denote an upper bound on the average overlap. Then, for every $\epsilon > 0$,*

$$\mathbf{Pr}[\Delta \geq \epsilon\mu + 1] \leq n \left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}} \right)^{\mu} \tag{5}$$

$$\mathbf{Pr}[\Delta \geq 5\epsilon\mu] \leq \frac{6}{\alpha} \left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}} \right)^{\mu} . \tag{6}$$

Bound (5) that is best suited for short arcs is derived by bounding the maximum overlap by the overlap at the discrete set of starting positions of arcs. We defer the analysis to the full paper since simple Chernoff bound arguments are sufficient in this case.

Perhaps the most interesting case are rather long arcs with $\alpha < 1/2$. Section 2.1 derives Bound (6) that is a up to a factor $\Theta(n)$ more tight in this case. The proof combines Chernoff bound arguments with random walk arguments that may be interesting for other applications too. Bounds C) and D) for even longer arcs can be proven using an almost symmetric argument on the *minimum* overlap of non-arcs or *holes*. The proof is deferred to the full paper. In the full paper we furthermore argue that our results are essentially tight by giving lower bounds in terms of a balls-into-bins process considering $1/\alpha$ equally spaced positions on the circle. Section 2.2 reports simulation results that even give some hints as to what the constant factors in Bounds B) and C) might be.

2.1 Proof of Bound (5)

Proof. Define $\kappa = \lceil 1/\alpha \rceil \geq 2$. Let $x_0, \ldots, x_{\kappa-1}$ denote κ points on the circle that decompose the circle into κ intervals $X_i = [x_i, x_{i+1})$ of identical length $\lceil \frac{1}{\kappa} \rceil \approx \alpha$. (Here and in the following $i+1$ abbreviates $(i+1) \bmod \kappa$.) Observe that every arc has at most one endpoint in each interval. Define $\Delta_i = L(x_i) - \alpha n$ and $\Delta'_i = \sup_{x \in X_i}(L(x) - L(x_i))$. In this way, the maximum overlap in interval X_i is exactly $\alpha n + \Delta_i + \Delta'_i$. Our argument is based on the following two claims:

$$\forall \epsilon > 0 : \mathbf{Pr}[\Delta_i \geq \epsilon\mu] \;\leq\; \left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{\mu} \tag{7}$$

$$\forall \epsilon > 0 : \mathbf{Pr}[\Delta'_i \geq \epsilon(2+2\epsilon)\mu \mid \Delta_i + \Delta_{i+1} \leq 2\epsilon\mu] \;\leq\; 2\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{(1+\epsilon)\mu} \tag{8}$$

Let us show that, in fact, these claims imply the theorem. First suppose $\epsilon \geq 1$. The maximum overlap in interval X_i is bounded above by $L(x_i) + L(x_{i+1}) = 2\alpha n + \Delta_i + \Delta_{i+1}$ because every arc overlapping with interval X_i covers x_i or x_{i+1}. This implies the inequality a1: $\Delta \leq \max_i\{\alpha n + \Delta_i + \Delta_{i+1}\}$ so that we obtain

$$\begin{aligned}\mathbf{Pr}[\Delta \geq 3\epsilon\mu] &\overset{(a1)}{\leq} \mathbf{Pr}[\exists_{i=0}^{\kappa-1} : \alpha n + \Delta_i + \Delta_{i+1} \geq 3\epsilon\mu] \overset{(a2)}{\leq} \mathbf{Pr}\left[\exists_{i=0}^{\kappa-1} : \Delta_i \geq \epsilon\mu\right] \\ &\overset{(a3)}{\leq} \frac{1}{\kappa}\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{\mu} \overset{(a4)}{\leq} \frac{2}{\alpha}\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{\mu},\end{aligned}$$

where inequality (a2) follows from $\alpha n \leq \epsilon\mu$, (a3) follows from Claim (7), and (a4) follows from $\kappa = \lceil \frac{1}{\alpha} \rceil$.

Now assume $\epsilon < 1$. Observe that this implies b1: $\epsilon \geq \epsilon(3+2\epsilon)/5$. Furthermore, we apply b2: $\Delta = \max_i\{\Delta_i + \Delta'_i\}$ and obtain

$$\begin{aligned}\mathbf{Pr}[\Delta \geq 5\epsilon\mu] &\overset{(b1)}{\leq} \mathbf{Pr}[\Delta \geq \epsilon(3+2\epsilon)\mu] \\ &\overset{(b2)}{\leq} \mathbf{Pr}[\exists_{i=0}^{\kappa-1} : \Delta_i \geq \epsilon\mu \vee \Delta'_i \geq \epsilon(2+2\epsilon)\mu] \\ &\overset{(b3)}{\leq} \sum_{i=0}^{\kappa-1} \mathbf{Pr}[\Delta_i \geq \epsilon\mu] + \mathbf{Pr}[\Delta'_i \geq \epsilon(2+2\epsilon)\mu \,|\, \Delta_i + \Delta_{i+1} \leq 2\epsilon\mu] \\ &\overset{(b4)}{\leq} \kappa\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{\mu} + 2\kappa\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{(1+\epsilon)\mu} \leq \frac{6}{\alpha}\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{\mu}.\end{aligned}$$

The following basic fact from probability theory implies inequality (b3). For events X, X', and Y with $X' \subseteq X$ it holds $\mathbf{Pr}[X \vee Y] \leq \mathbf{Pr}[X] + \mathbf{Pr}[Y \setminus X'] \leq \mathbf{Pr}[X] + \mathbf{Pr}[Y|X']$. Furthermore, inequality (b4) follows from the Claims (7,8).

It remains to prove the two claims. Observe that $\mathbf{E}[L(x_i)] \leq \mu$ so that the bound in Claim (7) follows directly from a Chernoff bound. Hence it only remains to show the Claim (8). We will estimate Δ'_i by investigating the following random

walk. Recall that each arc has at most one endpoint in interval X_i. Let m denote the number of those arcs that have an endpoint in X_i. As we condition on $\Delta_i + \Delta_{i+1} \leq 2\epsilon\mu$, we can assume

$$m \;\leq\; L(x_i) + L(x_{i+1}) \;=\; 2\mu + \Delta_i + \Delta_{i+1} \;\leq\; (2+2\epsilon)\mu \;.$$

For the time being, assume that m is fixed. Let $y_1, \ldots, y_m$ denote the endpoints in X_i, sorted from left to right. If we are given these endpoints without further information then the *orientation* of the corresponding arcs (i.e., whether y_j is a left or right endpoint) defines a random experiment that can be described in terms of binary random variables as follows. Let $s_1, \ldots, s_m$ denote random variables with

$$s_j \;=\; \begin{cases} +1 \text{ if } y_j \text{ is a left endpoint, and} \\ -1 \text{ if } y_j \text{ is a right endpoint.} \end{cases}$$

The only assumption that we made about the allocation of the arcs is $\Delta_i + \Delta_{i+1} \leq 2\epsilon\mu$. As this assumption does not affect the arc's orientation, the variables $s_1, \ldots, s_m$ are independent and $\mathbf{Pr}[s_j = 1] = \mathbf{Pr}[s_j = -1] = \frac{1}{2}$, for $1 \leq j \leq m$.

Now let us define $S_j = \sum_{k=1}^{j} s_j$, for $0 \leq j \leq m$. Notice that the sequence $S_0, S_1, \ldots, S_m$ corresponds to a random walk in which a particle starts at position 0 and goes up or down by one position in each step with probability $\frac{1}{2}$ each, that is, $\Delta_i' = \max_{0 \leq i \leq m}(S_i)$. Hence, we can estimate Δ_i' by analyzing this random walk. Applying Theorem III.7.1 from [6] we can derive the following probability bound. For every $r \geq 0$,

$$\mathbf{Pr}[\exists j\{1, \ldots, m\} : S_j \geq r] = \sum_{k=r+1}^{\infty} \mathbf{Pr}[S_m = k] + \mathbf{Pr}[S_m = k+1] \leq 2\mathbf{Pr}[S_m > r].$$

Next we observe that the random variable $X_m = S_m/2 + m/2$ follows the binomial distribution and, hence, can be estimated using a Chernoff bound. In this way, we obtain

$$\mathbf{Pr}[S_m > \epsilon m] \;=\; \mathbf{Pr}[X_m > (1+\epsilon)m/2] \;\leq\; \left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{m/2} \;,$$

for every $\epsilon > 0$. As a consequence,

$$\mathbf{Pr}[\Delta_i' > \epsilon m] \;\leq\; 2\mathbf{Pr}[S_m > \epsilon m] \;\leq\; 2\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{m/2} \;.$$

Clearly, Δ_i' is monotonically increasing in m. Therefore, the worst-case choice for m is $m = (2+2\epsilon)\mu$, and we obtain,

$$\mathbf{Pr}[\Delta_i' > \epsilon(2+2\epsilon)\mu] \;\leq\; 2\left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{(1+\epsilon)\mu} \;,$$

which proves Claim (8). Thus Theorem 6 is shown.

2.2 Simulations

Fig. 1 shows simulations for three different values of n and compares them with our analytic bound from the case B) and C) of our characterization for $\mathbf{E}[\Delta]$. Merging the bounds in these two cases we obtain the elegant estimate $\mathbf{E}[\Delta] \approx \sqrt{\alpha(1-\alpha)n \ln\left(\frac{1}{\alpha(1-\alpha)}\right)}$.

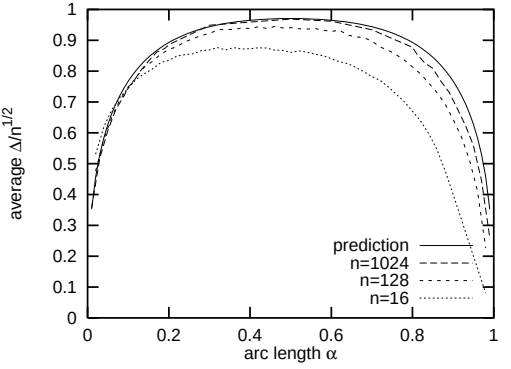


Fig. 1. Comparison of the theoretical prediction $\Delta = \sqrt{en \cdot \alpha(1-\alpha) \ln \frac{1}{\alpha(1-\alpha)}}$ with simulations for different n. The measurements are averages of 10000 repetitions for $n \leq 128$ and 1000 repetitions for $n = 1024$.

In fact, the measured and predicted curves are quite close together when we choose an appropriate constant factor (namely $\sqrt{e}$) in front of this estimate. Even for $n = 16$, where a significant influence of lower order terms can be expected, Δ is fairly well approximated. An interesting phenomenon is that the measured graphs for $\Delta(\alpha)$ are not completely symmetric around $\alpha = 1/2$. This asymmetry is not reflected by our theoretical analysis since we use the same Chernoff bounds to estimate the overlap of arcs in case of $\alpha \leq \frac{1}{2}$ and the overlap of holes in case of $\alpha > \frac{1}{2}$. In fact, however, the deviation of holes below the mean can be expected to be slightly less than the deviation of arcs above the mean, which explains the asymmetry that can be observed in the experiments.

3 Applications

We now present three examples, where our bounds on arc allocation yield performance guarantees. In Section 3.1, Bound (2) guarantees that rotational delays in accessing a single disks eventually become small compared to data access times. Section 3.2 gives a different disk scheduling application where load balancing data accesses over several disks is the objective. Whereas the first two example concern execution time, the last example in Section 3.3 bounds the memory consumption of a technique for reducing cache interference misses.

In all three applications very bad worst case behavior is avoided using randomization. The price paid is that the best case behavior gets worse. Since best case behavior may sometimes not be far from real inputs, it is crucial for our

performance bounds to demonstrate that this possible penalty is very small. This is a quite strong incentive to study lower order terms in the performance bounds rather than bounds that leave the constant factors open.

3.1 Disks and Drums

One of our main motivations for studying arc allocations was the desire to find disk scheduling algorithms that take rotational delays into account. For example, consider a data base query selecting the set $S = \{x \in r : x.a = y\}$ from a relation r. Assume we have an index of r with respect to attribute a that tells us a set of small disk blocks where we can find S. In this situation, the access time for retrieving S is dominated by rotational delays and seek times rather than the access or transmission time of the data [14]. Unfortunately, simultaneously minimizing seek times and rotational delays is NP-hard [1]. On the other hand, the explosive growth of disk storage densities in the last years suggests to consider the case where the accessed file fits into a narrow zone on the disk. In this case, seek times can be bounded by a constant that is only slightly larger than the overhead for request initiation, data transmission, and settling the head into a stable state. Such constant overheads can be absorbed into the size of the blocks and we end up with a problem where only block lengths and rotational delays matter. Interestingly, this reasoning leads to a model logically identical to *drums* — rotating storage devices from the 50s [4] that are technologically outdated since the 70s. Drums have a separate read/write head for every data track and hence suffer no seek delays.

To read a block on a drum one just has to wait until its start rotates to the position of the read head and then read until the end of the block. Suppose we want to read a batch of n blocks efficiently. Each block can be modeled as an arc in an arc allocation problem in the obvious way. Obviously, L is a lower bound for the number of drum rotations needed to retrieve all n blocks. Fuller [7,16] found optimal and near optimal drum scheduling algorithms that can retrieve the blocks in at most $L+1$ drum rotations. One such algorithm, *Shortest Latency Time First (SLTF)*, is very simple: When the drum is at point x, pick the unread block i whose starting point a_i is closest, read it, set $x = a_i+\alpha_i \bmod 1$ and iterate until all blocks are read.

The question arises, how good is an optimal schedule. In the worst case, n rotations may be needed. For example, if all blocks have the same starting point. In this case, even a good scheduling algorithm is of little help. Our results provide us with very attractive performance guarantees if the starting points are randomized. We need time $n\alpha + O(\sqrt{n})$ rotations and hence for large n, almost all of the schedule time is spent productively reading data.

Performance guarantees for random starting points need not merely be predictions for the average case if we randomize the mapping of logical blocks to physical positions. Here is one scheme with particular practical appeal: Start with a straightforward non-randomized mapping of logical blocks to physical positions by filling one track after the other. Now rotate the mapping on each track by a random amount. This way, accesses to consecutive blocks mapped to the same track remain consecutive. A technical problem is that starting points

are not completely independent so that our analysis does not strictly apply. However, starting points of two blocks in a batch of blocks to be read are either independent or the two blocks do not overlap (we merge consecutive blocks on the same track). Therefore it seems likely that our upper bounds still apply.

3.2 Parallel Disk Striping

Assume a file is to be allocated to M parallel disks numbered 0 through $M-1$ so that accesses to any consecutive range of data in the file can be done efficiently. A common allocation approach is *striping* — block i of the data stream is allocated to disk $i \bmod M$. The situation gets more complicated if n files are accessed concurrently. Barve et al. [2] propose *simple randomized striping (SR)* where block i of a file f is mapped to disk $r_f + i \bmod M$ where r_f is a random offset between 0 and $M-1$. The number of I/O steps needed to access N blocks from the n data streams and M disks is the variable $L^{(\mathrm{cb})}$ in the corresponding chains-into-bins problem. Our results improve the performance bounds shown in [2] for the case that $n \ll N$.

3.3 Cache Efficient Allocation of DRAM Memory

Many algorithms that are efficient on memory hierarchies are based on accessing n arrays in such a way that accesses in each array are sequential but where it is unpredictable how accesses to different arrays are interleaved. In [11] it is shown that these algorithms even work well on hardware caches where we have no direct control over the cache content provided that the starting points of the arrays modulo the cache size M are chosen at random. (Essentially the SR disk striping from [2] is applied to words in the cache.) Now the question arises how to allocate the arrays. Assume we have a large contiguous amount of memory available starting at address $x \bmod M$. A naive approach allocates one array after the other in arbitrary order by picking a random location $y \bmod M$ and allocating the array so that it starts at point $x + (y - x \bmod M)$. On the average this strategy wastes $nM/2$ of storage. A better way is by applying the simple SLTF algorithm we have seen for drum scheduling. This way we need space ML on the average wasting only $M\Delta$ of memory.

This bound for the offline algorithm also translates into a performance guarantee for an online allocation algorithm where requests for memory segments arrive one at a time. The following greedy algorithm can be easily proven to be 2-competitive: Keep a list of free memory intervals sorted by starting address. Allocate a new segment in the first free interval that has room at the right address offsets. Hence, space $2M(L + \Delta)$ memory suffices to fulfill all requests. In the final version of [11] citing our result replaces a lengthy derivation that shows a bound of $4eML$ for the online algorithm. No result on the offline case was available there.

Acknowledgements. We would like to thank Ashish Gupta and Ashish Rastogi for fruitful cooperation on disk scheduling algorithms.

References

1. M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In IEEE, editor, *37th Annual Symposium on Foundations of Computer Science*, pages 550–559. IEEE Computer Society Press, 1996.
2. R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.
3. B. Bhattacharya, P. Hell, and J. Huang. A linear algorithm for maximum weight cliques in proper circular arc graphs. *SIAM Journal on Discrete Mathematics*, 9(2):274–289, May 1996.
4. A. A. Cohen. Technical developments: Magnetic drum storage for digital information processing systems (in Automatic Computing Machinery). *Mathematical Tables and Other Aids to Computation*, 4(29):31–39, January 1950.
5. R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth. On the lambert W function. *Advances in Computational Mathematics*, 5:329–359, 1996.
6. W. Feller. *An Introduction to Probability Theory and its Applications*. Wiley, 3rd edition, 1968.
7. S. H. Fuller. An optimal drum scheduling algorithm. *IEEE Trans. on Computers*, 21(11):1153, November 1972.
8. M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic and Discrete Methods*, 1(2):216–227, June 1980.
9. V. Klee. What are the intersections graphs of arcs in a circle? *Amer. Math. Monthly*, 76:810–813, 1969.
10. V. F. Kolchin, B. A. Sevatyanov, and V. P. Chistiakov. *Random Allocations*. V. H. Winston, 1978.
11. K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 2002. to appear.
12. M. Mitzenmacher, A. Richa, and R. Sitaraman. The power of two random choices: A survey of the techniques and results. In P. Pardalos, S. Rajasekaran, and J. Rolim, editors, *Handbook of Randomized Computing*. Kluwer, 2000.
13. M. Raab and A. Steger. "balls into bins" – A simple and tight analysis. In *RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science*. LNCS, 1998.
14. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.
15. A. F. Siegel and L. Holst. Covering the circle with random arcs of random sizes. *J. Appl. Probab.*, 19:373–381, 1982.
16. H. S. Stone and S. F. Fuller. On the near-optimality of the shortest-access-time-first drum scheduling discipline. *Communications of the ACM*, 16(6), June 1973. Also published in/as: Technical Note No.12, DSL.
17. J. S. Vitter and P. Flajolet. Average case analysis of algorithms and data structures. In *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 9, pages 431–524. Elsevier, 1990.

On Neighbors in Geometric Permutations*

Micha Sharir[1,2] and Shakhar Smorodinsky[1**]

[1] School of Computer Science,
Tel Aviv University, Tel Aviv 69978, Israel,
smoro@tau.ac.il
sharir@cs.tau.ac.il
[2] Courant Institute of Mathematical Sciences,
New York University, New York, NY 10012, USA;

Abstract. We introduce a new notion of 'neighbors' in geometric permutations. We conjecture that the maximum number of neighbors in a set S of n pairwise disjoint convex bodies in $\mathbb{R}^d$ is $O(n)$, and we prove this conjecture for $d = 2$. We show that if the set of pairs of neighbors in a set S is of size N, then S admits at most $O(N^{d-1})$ geometric permutations. Hence we obtain an alternative proof of a linear upper bound on the number of geometric permutations for any finite family of pairwise disjoint convex bodies in the plane.

1 Background and Motivation

Let S be a finite family of convex bodies in $\mathbb{R}^d$. A line ℓ that intersects every member of S is called a *line transversal* of S.

If the bodies in S are pairwise disjoint, then a line transversal ℓ of S induces a pair of linear orderings on S (one order being the reverse of the other order), which are the orders in which the members of S are met by ℓ, corresponding to the two orientations of ℓ. Katchalski et al. [6] were the first to study such pairs of orderings and called them *geometric permutations*. We refer to [3,4,5,6,7,8] for the recent study of this concept, its applications and its generalizations.

Line transversals have practical applications in computing visibility information for efficient rendering of scenes, e.g., in 3-dimensional computer games and in architectural walkthroughs. See [2,12] for the algorithmic aspects of line transversals.

Let $g_d(n)$ denote the maximum number of geometric permutations in $\mathbb{R}^d$, where the maximum is taken over all such families S of size n. A major open

* Work on this paper has been supported by NSF Grants CCR-97-32101 and CCR-00-98246, by a grant from the U.S.-Israeli Binational Science Foundation, by a grant from the Israeli Academy of Sciences for a Center of Excellence in Geometric Computing at Tel Aviv University, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

** The research by this author was done while the author was a Ph.D. student under the supervision of Micha Sharir.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 131–139, 2002.

problem in transversal theory is to give sharp asymptotic bounds on $g_d(n)$. Edelsbrunner and Sharir [3] have shown that $g_2(n) = 2n - 2$ (for $n > 3$). Katchalski et al. [7] showed that $g_d(n) = \Omega(n^{d-1})$. The only known general upper bound on $g_d(n)$ is $O(n^{2d-2})$ and is due to Wenger [13]. Hence, for $d \geq 3$ there still exists a wide gap between the known upper and lower bounds. Recently, Smorodinsky et al. [10,11], obtained a tight bound of $\Theta(n^{d-1})$ on the maximum number of geometric permutations, in the special case where S consists of pairwise disjoint balls in $\mathbb{R}^d$. (For the case of congruent or nearly congruent balls, the number of geometric permutations is only $O(1)$, as shown recently by Zhou and Suri [14].) This result was followed by an extension of the same bound to the case of "fat" convex bodies, by Katz and Varadarajan [8]. A very recent result by Koltun and Sharir [9] implies that the number of geometric permutations of a set S of n pairwise disjoint convex semialgebraic sets of constant description complexity in $\mathbb{R}^3$ is $O(n^{3+\epsilon})$, for any $\epsilon > 0$; this is an intermediate bound between the upper bound $O(n^4)$ of [13] and the lower bound $\Omega(n^2)$ of [7]. It has been conjectured that $g_d(n) = O(n^{d-1})$.

1.1 Separation Sets and Neighbors

Wenger [13] introduced the notion of *separation set*, which was later generalized in [11]:

Definition 1. *Let S be a family of pairwise disjoint convex sets in $\mathbb{R}^d$, and let P be a set of hyperplanes in $\mathbb{R}^d$ passing through the origin. We say that P is a* separation set *for S if for each pair $s_i, s_j \in S$ there exists a hyperplane h, parallel to a hyperplane in P, such that s_i and s_j are contained in different open half-spaces bounded by h.*

For a family S of pairwise disjoint convex sets, let $GP(S)$ denote the number of geometric permutations of S.

Lemma 2. *(see [10,13]). Let S be a collection of pairwise disjoint convex sets in $\mathbb{R}^d$ and let P be a separation set for S. Then $GP(S) = O(|P|^{d-1})$.*

Proof (sketch): Consider the cross section $\mathcal{A}^*(P)$ of the arrangement $\mathcal{A}(P)$ within the unit sphere $\mathbb{S}^{d-1}$; it is an arrangement of $|P|$ great spheres, each consisting of all orientations parallel to some hyperplane in P. It is easy to show that, for each cell C of $\mathcal{A}^*(P)$, there is at most one possible geometric permutation that is induced by lines with orientation in C, and this implies the asserted bound. □

Since any set of n pairwise disjoint convex bodies admits a separation set of size $\binom{n}{2}$, one obtains an upper bound of $O(n^{2d-2})$ on the number of geometric permutations in d-space. (This is Wenger's proof.)

It was shown in [11] that, if S is a collection of pairwise disjoint balls in $\mathbb{R}^d$, then there exists a separation set for S of size $O(n)$, where the constant of proportionality depends on the dimension d. Hence, combined with Lemma

2, one immediately gets an $O(n^{d-1})$ upper bound on the number of geometric permutations for S. The fact that, in any dimension, balls can be separated with only $O(n)$ hyperplane directions seems to depend on their fatness. Indeed, Katz and Varadarajan [8] showed later that any set of pairwise disjoint α-*fat* convex bodies in $\mathbb{R}^d$ (i.e., the ratio between the circumradius and inradius of any input body is at most α, where α is a constant), admits a separation set of size $O(n)$, where the constant of proportionality depends on α and on the dimension d.

The problem with the notion of separation set is that, already in three dimensions, one cannot hope to get a separation set of linear size for general convex bodies. In fact, there exist collections S of n pairwise disjoint convex bodies in $\mathbb{R}^3$, for arbitrarily large n, such that any separation set for S is of size $\Omega(n^2)$. For example, one can take S to be the collection of Voronoi cells of the Voronoi diagram of a set consisting of $n/2$ points on the line $l_1 : x = 0; z = 1$ and of $n/2$ points on the line $l_2 : y = 0; z = 0$. It is easily seen that every pair of points (p, q), such that $p \in l_1$ and $q \in l_2$, have touching Voronoi cells, and that the separating facets of these pairs of cells have different orientations. Shrink the bodies in S by a small amount, to make them pairwise disjoint. Hence any separation set for S must contain a distinct plane for each of these pairs of formerly touching cells, so its size must be $\Theta(n^2)$. Therefore, Lemma 2 is useless in the general case.

In this paper, we introduce a weaker notion of separation, and show that it can be used, in a manner similar to that in Lemma 2, to derive bounds on the number of geometric permutations.

Definition 3. *Let S be a family of n pairwise disjoint convex sets in $\mathbb{R}^d$. Two objects $a, b \in S$ are called* neighbors *if there exists a line tranversal l of S such that a and b are consecutive elements of the geometric permutation induced by l.*

Denote by $N(S)$ the set of all neighbors in S. Note that if $GP(S) = 0$ then $N(S) = \emptyset$ and if $GP(S) \geq 1$ then $|N(S)| \geq n - 1$.

Conjecture 4. *If S is a family of pairwise disjoint convex bodies in $\mathbb{R}^d$ then $|N(S)| = O(n)$.*

In Section 2 we establish the conjecture for the planar case.

It can be shown that, in the above example involving Voronoi cells, we have $GP(S) \geq 1$, and there are indeed only $O(n)$ neighbors. The following lemma shows that, if the conjecture is true, it leads to sharp bounds on the number of geometric permutations in the general case.

Lemma 5. *Let S be a family of n pairwise disjoint convex bodies in $\mathbb{R}^d$. Then $GP(S) = O(|N(S)|^{d-1})$.*

Proof: For each pair of bodies $a, b \in N(S)$, let $h_{a,b}$ be any hyperplane separating a and b, and let P be the set of these $|N(S)|$ hyperplanes. We proceed as in the proof of Lemma 2, by considering the arrangement $\mathcal{A}^*(P)$ of the great spheres in $\mathbb{S}^{d-1}$ associated with the hyperplanes in P. Fix a (full-dimensional) cell C of $\mathcal{A}^*(P)$ and let ℓ be an oriented line transversal to S with orientation in C.

ℓ induces a geometric permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ on S. For each $i = 1, \ldots, n-1$, the elements π_i, π_{i+1} are neighbors (by definition), so there exists a hyperplane $h_i \in P$ that separates them. h_i corresponds to one of the great spheres in the arrangement $\mathcal{A}^*(P)$, with C lying in one of the two corresponding hemispheres. This means that every oriented line that intersects both π_i and π_{i+1} and has orientation in C, must intersect these two sets in a fixed order. This implies that in any geometric permutation induced by a line transversal with orientation in C, π_i must precede π_{i+1}, for each $i = 1, \ldots, n-1$. Clearly, exactly one geometric permutation (namely, π) has this property. We conclude that there is at most one geometric permutation that can be induced by lines with orientation in C. Hence the number of geometric permutations is at most the number of cells in $\mathcal{A}^*(P)$, which is $O(|N(S)|^{d-1})$. □

2 Linear Bound on the Number of Neighbors in the Plane

In this section we prove our main result, showing that our conjecture is true in the plane.

Theorem 6. *Let S be a set of n pairwise disjoint convex bodies in the plane. Then the number of neighbor pairs in S is $O(n)$.*

Proof: We define a graph G on S, whose edges connect each of the neighbor pairs. We will show that G is a *quasi-planar* graph, i.e., G can be drawn in the plane so that no three edges with distinct endpoints are pairwise crossing. The theorem then follows from the result of Agarwal et al. [1], that quasi-planar graphs have linear size.

We draw G as follows. For each object $a \in S$ we fix a point v_a inside a; these are the vertices of (the drawing of) G. If (a, b) are neighbors, we choose one transversal line ℓ of S along which $\ell \cap a$ and $\ell \cap b$ appear consecutively. Let q_a and q_b be the two nearest endpoints of these two respective intervals. Then we draw the edge ab of G as the polygonal arc $v_a q_a q_b v_b$. See Figure 1.

To prove that G is quasi-planar, assume to the contrary that there exist six distinct objects a, b, c, d, e, f in S, such that the edges (a, b), (c, d), (e, f) are pairwise crossing. Any of these edges, say, (a, b), is drawn as a polygonal arc, whose portion outside $a \cup b$ is a straight segment s_{ab}, connecting a point $q_a \in \partial a$ to a point $q_b \in \partial b$, such that s_{ab} is contained in a line transversal ℓ_{ab} of S, along which a and b are neighbors, and so that the relative interior of s_{ab} is disjoint from a and b (and from all other objects in S as well). Similar constructions hold for the edges (c, d) and (e, f). Note that the crossing between, say (a, b) and (c, d) must be at an interior point of s_{ab} and of s_{cd}, and similarly for the two other crossings.

The removal of s_{ab} from ℓ_{ab} partitions this line into two rays, one of which meets a and is denoted as ρ_a, and the other meets b, and is denoted as ρ_b. A similar notation holds for the two other lines.

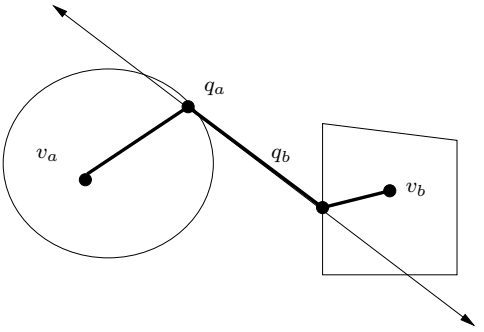


Fig. 1. Drawing an edge of G.

Note that the lines ℓ_{ab}, ℓ_{cd}, ℓ_{ef} must be distinct transversals of S. Let Δ be the triangle whose vertices are the crossing points of the segments s_{ab}, s_{cd}, s_{ef}. Order the six objects a, b, c, d, e, f according to the counterclockwise angular order of the endpoints $q_a, q_b, q_c, q_d, q_e, q_f$ about (any point of) Δ. Without loss of generality, assume that this circular order is (a, c, e, b, d, f); note that the three pairs a and b, c and d, and e and f must be 'antipodal' elements of this order. See Figure 2.

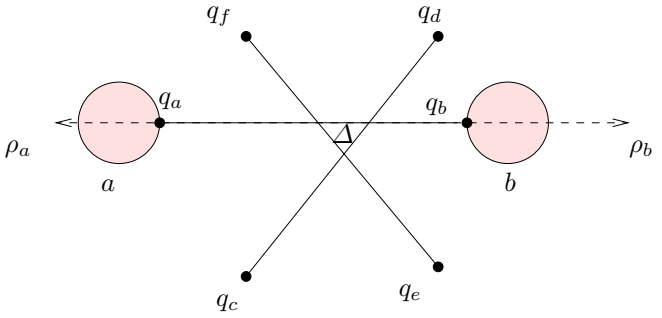


Fig. 2. The structure of a triple edge-crossing in G.

Since each of the lines ℓ_{ab}, ℓ_{cd}, ℓ_{ef} is transversal to S, it meets each of the six sets a, b, c, d, e, f. Consider, say, the line ℓ_{ab}. Since the relative interior of the contained segment s_{ab} is disjoint from all sets in S, each of the four other objects c, d, e, f meets ℓ_{ab} at one (and only one) of the two complementary rays ρ_a, ρ_b.

We make the following simplification of the configuration, by replacing each of these six objects by a straight segment. Consider one of the objects, say a. Pick a point $q'_a \in a \cap \ell_{cd}$, and a point $q''_a \in a \cap \ell_{ef}$. Then $q'_a \in \rho_c \cup \rho_d$ and $q''_a \in \rho_e \cup \rho_f$. It is impossible that $q'_a \in \rho_d$ and $q''_a \in \rho_e$, for then the segment s_{ab}

would not be disjoint from a, as is easily seen (see Figure 2). Hence there are three possible cases (the mnemonics refer to the positioning of q_a in the triple $\{q_a, q'_a, q''_a\}$):

(i) $q'_a \in \rho_c$ and $q''_a \in \rho_f$. In this case we refer to a as a *middle* object, and replace it by the straight segment $q'_a q''_a$; see Figure 3(a).
(ii) $q'_a \in \rho_c$ and $q''_a \in \rho_e$. In this case we refer to a as a *clockwise* object, and replace it by the straight segment $q_a q''_a$; see Figure 3(b).
(iii) $q'_a \in \rho_d$ and $q''_a \in \rho_f$. In this case we refer to a as a *counterclockwise* object, and replace it by the straight segment $q_a q'_a$; see Figure 3(c).

Similar 'straightenings' are applied to the five other objects. Each straightened object is contained in the corresponding original object, and thus the resulting six segments are pairwise disjoint. Note that the straightened a need not contain the point q_a (in case (i)). In this case we replace q_a by the intersection of the straightened a with ℓ_{ab}. This step, applied to each of the objects if needed, may cause some of the segments s_{ab}, s_{cd}, s_{ef} to expand, but it is easily checked that the relative interiors of the new segments are still all disjoint from all the new six objects; see Figure 3(a).

Lemma 7. *If one of the six objects is middle or clockwise (resp., middle or counterclockwise) then the object immediately succeeding (resp., preceding) it in the counterclockwise circular order about Δ must be clockwise (resp., counterclockwise).*

Proof: Without loss of generality, assume that the first object is a and that it is middle or clockwise. The succeeding object, c, must meet ℓ_{ab} within ρ_b, or else a and c would have to intersect (because any segment connecting q_c to a point on ρ_a has to cross $q_a q'_a \subseteq a$). Then c must be clockwise. □

Corollary 8. *Either all six objects are clockwise or all are counterclockwise.*

Without loss of generality, we may assume that all objects are clockwise. Start at the a-endpoint $v_1 = q_a$ of s_{ab}, and draw through it a line that is parallel to ℓ_{ef} and intersects ℓ_{cd} at a point v_2, which lies counterclockwise from v_1 with respect to Δ. Then draw from v_2 a line parallel to ℓ_{ab} which intersects ℓ_{ef} at a point v_3, again, lying counterclockwise from v_2. Keep 'turning' like this to trace a closed hexagon H. (It is not hard to show that this process does indeed always produce a closed hexagon.) See Figure 4.

We claim that the vertices of H satisfy

$$v_1 = q_a,\ v_2 \in \rho_c,\ v_3 \in \rho_e,\ v_4 \in \rho_b,\ v_5 \in \rho_d,\ v_6 \in \rho_f.$$

Indeed, since the segment a is clockwise, it meets ρ_e, and thus must lie counterclockwise to the ray $\overline{q_a v_2}$. Since $a \cap \ell_{cd} \in \rho_c$, we have $v_2 \in \rho_c$ as well. A similar argument, proceeding inductively counterclockwise around H, holds for all other vertices.

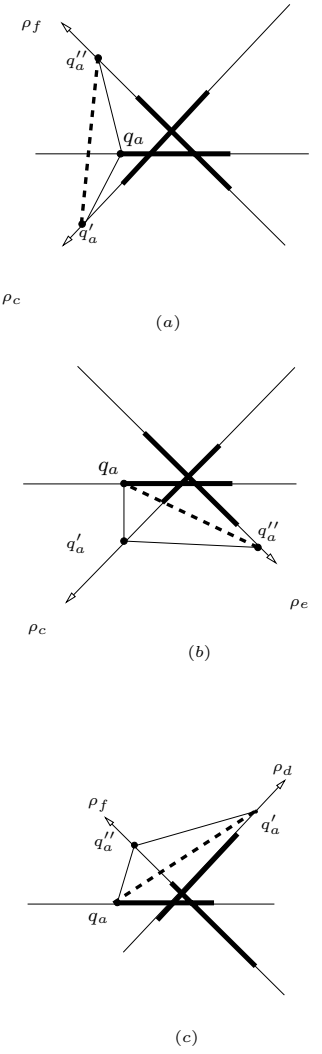


Fig. 3. The three types of objects and their straightening.

Note that the segment f, being clockwise, meets ρ_a and ρ_c. Hence its orientation is on one hand counterclockwise to that of $\overline{v_6 q_a} = \overline{v_6 v_1}$ (or else it would not have met ρ_c), and is on the other hand clockwise to that of $\overline{v_6 v_1}$ (since $v_6 \in \rho_f$ and f meets ρ_a). This contradiction implies that the graph G cannot have three pairwise crossing edges, and thus G is quasi-planar and has linear size. This completes the proof of Theorem 6 □

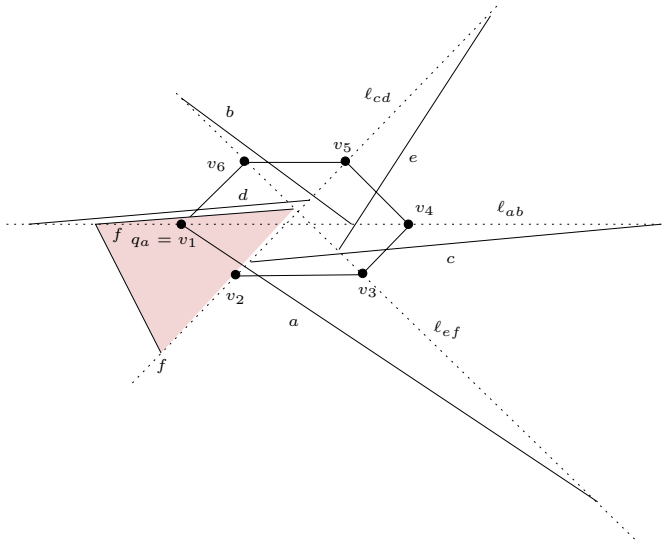


Fig. 4. The hexagon H and the final contradiction.

3 Conclusion

Theorem 6, combined with Lemma 5, yields an alternative proof that $g_2(n) = O(n)$ (albeit with a weaker bound than the tight $2n-2$ bound in [3]). The main open problem is to establish the conjecture in higher dimensions, in particular for $d = 3$.

Interesting by itself is the use of the linear bound on the size of quasi-planar graphs in the proof of Theorem 6. This application is among the very few known applications of quasi-planarity. It is conceivable, though, that the neighbor graph G is in fact planar. However, our specific (and quite natural) way of drawing G can have crossing edges, as can be easily shown.

References

[1] P. Agarwal, B. Aronov, J. Pach, R. Pollack and M. Sharir, Quasi-planar graphs have a linear number of edges, *Combinatorica* 17:1–9, 1997.

[2] N. Amenta. Finding a line transversal of axial objects in three dimensions. *Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms*, pages :66–71, 1992.

[3] H. Edelsbrunner and M. Sharir. The maximum number of ways to stab n convex non-intersecting sets in the plane is $2n-2$. *Discrete Comput. Geom.*, 5(1):35–42, 1990.

[4] J.E. Goodman and R. Pollack. Hadwiger's transversal theorem in higher dimensions. *J. Amer. Math. Soc.*, 1(2):301–309, 1988.

[5] J.E. Goodman, R. Pollack and R. Wenger. Geometric transversal theory. In *New Trends in Discrete and Computational Geometry*, (J. Pach, ed.), Springer Verlag, Heidelberg, 1993, pp. 163–198.

[6] M. Katchalski, T. Lewis, and A. Liu. Geometric permutations and common transversals. *Discrete Comput. Geom.*, 1:371–377, 1986.

[7] M. Katchalski, T. Lewis, and A. Liu. The different ways of stabbing disjoint convex sets. *Discrete Comput. Geom.*, 7:197–206, 1992.

[8] M.J. Katz and K.R. Varadarajan. A tight bound on the number of geometric permutations of convex fat objects in $\mathbb{R}^d$. *Proc. 17th Annu. ACM Sympos. Comput. Geom*, 2001 :249–251.

[9] V. Koltun and M. Sharir. The partition technique for overlays of envelopes. Manuscript, 2002.

[10] S. Smorodinsky. Geometric Permutations and Common Transversals. Master's thesis, Tel-Aviv University, School of Mathematical Sciences, Tel-Aviv, Israel, July 1998.

[11] S. Smorodinsky, J.S.B Mitchell, and M. Sharir. Sharp bounds on geometric permutations of pairwise disjoint balls in $\mathbb{R}^d$. *Discrete Comput. Geom.*, 23(2):247–259, 2000.

[12] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proc. SIGGRAPH '91)*, 25:61–69, 1991.

[13] R. Wenger. Upper bounds on geometric permutations for convex sets. *Discrete Comput. Geom.*, 5:27–33, 1990.

[14] Y. Zhou and S. Suri. Shape sensitive geometric permutations. *Proc. 12th ACM-SIAM Sympos. Discrete. Algorithms*, 2001 :234–243.

Powers of Geometric Intersection Graphs and Dispersion Algorithms

Geir Agnarsson[1], Peter Damaschke[2], and Magnús M. Halldórsson[3]

[1] Department of Computer Science, Armstrong Atlantic State University, Savannah, Georgia 31419-1997. geir@drake.armstrong.edu
[2] Department of Computer Science, Chalmers University, 41296 Göteborg, Sweden. ptr@cs.chalmers.se
[3] Department of Computer Science, University of Iceland, IS-107 Reykjavík, Iceland, and Iceland Genomics Corp. (UVS), Snorrabraut 60, IS-105 Reykjavik. mmh@hi.is

Abstract. We study powers of certain geometric intersection graphs: interval graphs, m-trapezoid graphs and circular-arc graphs. We define the *pseudo product*, $(G, G') \to G * G'$, of two graphs G and G' on the same set of vertices, and show that $G*G'$ is contained in one of the three classes of graphs mentioned here above, if both G and G' are also in that class and fulfill certain conditions. This gives a new proof of the fact that these classes are closed under taking power; more importantly, we get efficient methods for computing the representation for G^k if $k \geq 1$ is an integer and G belongs to one of these classes, with a given representation sorted by endpoints. We then use these results to give efficient algorithms for the *k-independent set*, *dispersion* and *weighted dispersion* problem on these classes of graphs, provided that their geometric representations are given.

1 Introduction

The dispersion problem is to select a given number of vertices in a graph so as to maximize the minimum distance between them. The problem is dual to the *Maximum k-Independent Set problem* (k-IS), which is that of finding a maximum collection of vertices whose inter-vertex distance is greater than a given bound k. That problem in turn is equivalent to the *Maximum Independent Set problem* (MIS) on the power graph G^k of the original graph. Thus, in order to give efficient dispersion algorithms, we are led to study efficient methods for constructing k-independent sets and power graphs, as well as to study structural properties of these powers.

In this article we do this for some classes of geometric intersection graphs listed below. The containment of graph classes under study is as follows: m-trapezoid graphs are interval graphs when $m = 0$, trapezoid graphs when $m \leq 1$, and cocomparability graphs for any m. Similarly, circular-arc graphs form a proper subclass of circular m-trapezoid graphs, and they also properly contain the respective non-circular class.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 140–149, 2002.

These and various other classes of graphs have been shown to be closed under taking power [5,6,9,10,16,17]. Generally, these proofs of containment do not immediately yield efficient algorithms. This led us to derive an efficient method for computing the power graph G^k of an interval graph G in time $O(n \log k)$ [2]. We both improve and generalize this result in the present paper.

To this end we define the *pseudo product* $(G, G') \to G * G'$ for two general graphs on the same set of vertices. This composition turns out to be commutative, but not associative in general. However, when we restrict to the class of various powers of a fixed graph, then the pseudo product is also associative, in fact if s and t are positive integers then $G^s * G^t = G^{s+t}$.

Our fast power computations finally lead to efficient algorithms for the *dispersion* problem in the mentioned classes. The problem is defined as follows:

(Weighted) Dispersion
Given: Graph G (with vertex weights $w : V(G) \to \mathbf{R}$), and a number q.
Find: A set of vertices with cardinality (total weight) at least q, with the minimum distance between the chosen vertices at maximum.

Dispersion is NP-hard for general graphs, since MIS is reducible to it, while it can be approximated in polynomial time within factor 2, see e.g. [18]. In fact, it is also hard to approximate within any factor less than 2, unless $P = NP$ [11].

There is an obvious relationship between *Dispersion* and k-independent sets in graph classes being closed under taking power. However the straightforward use of it yields $\Omega(n^2)$ time dispersion algorithms in such classes. For faster algorithms we have to avoid explicit insertion of edges when considering the k-th powers of G, such that fast power computations is exactly what we need here.

We want to make very clear that we always assume that our graphs are given by their geometric representations, rather than by their edge lists. Thus they are described in $O(n)$ space, by the extreme points of geometric sets representing the vertices. Without such a representation, it is impossible to achieve equally fast algorithms for the problems we study. (For m-trapezoid graphs, already the recognition problem is NP-hard [9].) We further assume everywhere in this paper that the endpoints of the intervals/arcs/trapezoids are given in a *sorted* list. This quite natural assumption which is common in the literature on algorithms in these classes allows us to derive $o(n \log n)$ algorithms for many of these problems.

Only a few subquadratic dispersion algorithms have been provided before: Dispersion is solvable in $O(n)$ time for trees [3], while weighted dispersion in $O(n \log n)$ time for paths [18], and in $O(n \log^4 n)$ time for trees [4]. The current paper extends a recent paper of Damaschke [7], who gave $O(n \log n/q^2)$ time algorithm for unweighted dispersion of circular-arc graphs, $O(n \log \log n)$ time for interval graphs, and $O(n \log^2 n)$ time for weighted dispersion on trapezoid graphs.

1.1 Outline of Results

The following table summarizes the asymptotic complexity of the problems that we consider, provided that a representation as stated above is given. In the table,

n denotes the number of vertices in G, and lg refers to the usual base-2 logarithm, an m is any constant.

Class	**Power**	**k-IS**	**Dispersion**	**W-Dispersion**
Interval	n	n	n	$n \lg k$
Circular-arc	n	n	n	
m-Trapezoid	$n \lg k$	$n(\lg k + (\lg\lg n)^m)$	$n(\lg k + (\lg\lg n)^m)$	$n \lg k (\lg\lg n)^m$

1.2 Notation

We will denote the positive integers $\{1, 2, 3, \ldots\}$ by $\mathbf{N}$, the nonnegative integers $\{0, 1, 2, \ldots\}$ by $\mathbf{N}_0$, the set of real numbers by $\mathbf{R}$, the Cartesian product $\mathbf{R} \times \mathbf{R}$ by $\mathbf{R}^2$, and the set of the closed interval $\{x : a \leq x \leq b\}$ by $[a; b]$. All graphs we consider are simple unless otherwise stated. For a graph G the set of its vertices will be denoted by $V(G)$, and the set of its edges by $E(G)$. The open neighborhood of a vertex v in G, that is, the set of neighbors of v not including v, will be denoted by $N_G(v)$. The closed neighborhood of a vertex v in G, that is, the set of neighbors of v, including v itself, will be denoted by $N_G[v]$. For two vertices u and v in G, the distance between them will be denoted by $d_G(u, v)$ or simply by $d(u, v)$ when unambiguous. We use notation compatible with [19].

Recall for a graph G and an integer k, the *k-th power* of G is the graph G^k on the same set of vertices as G, and where every pair of vertices of distance k or less in G are connected by an edge. Also, a graph G is called the *intersection graph* of a collection of sets $\{S_1, \ldots, S_n\}$ if $V(G) = \{v_1, \ldots, v_n\}$ and $\{v_i, v_j\} \in E(G) \Leftrightarrow S_i \cap S_j \neq \emptyset$, for all distinct $i, j \in \{1, \ldots, n\}$. Note that when G is represented by $\{S_1, \ldots, S_n\}$, then $d(S_i, S_j)$ is just the distance $d_G(v_i, v_j)$ between v_i and v_j in the intersection graph G.

2 Powers of m-Trapezoid Graphs

In this section we discuss a way to calculate the k-th power of an m-trapezoid graphs efficiently, and, as a special case, an interval or a circular-arc graph by means of the pseudo product which we define here below.

Definition 1. *Let G and G' be simple graphs on the same set of vertices $V(G) = V(G') = V$, where $|V| = n \geq 1$. Define the* pseudo product *of G and G' to be the simple graph $G * G'$ on the set vertex set V with the edge set $E(G * G') = E(G) \cup E(G') \cup E^*$ where*

$$E^* = \{\{u, v\} : \exists w \in V : \{u, w\} \in E(G), \{w, v\} \in E(G'), \\ \text{and } \exists w' \in V : \{u, w'\} \in E(G'), \{w', v\} \in E(G)\}.$$

If we view $*$ as an operation among all the simple graphs on V, then it is *not* an associative operation, in other words, the formula $(G*G')*G'' = G*(G'*G'')$ does not hold in general. We have however the following, which is a direct consequence of Definition 1.

Proposition 1. *For a simple graph G and nonnegative integers s and t, we have $G^s * G^t = G^{s+t}$. In particular, the pseudo product is an associative operation on the set $\{G^k : k \in \{0, 1, 2, \ldots\}\}$ for any fixed simple graph G.*

Assume that for each $l \in \{0, 1, \ldots, m\}$ we have two real numbers, a_l and b_l where $a_l < b_l$. As defined in [9], an *m-trapezoid* T is simply the closed interior of the polygon formed by the points $S = \{(a_l, l), (b_l, l) : l \in \{0, 1, \ldots, m\}\} \subseteq \mathbf{R}^2$. That means, the left side of the polygon is the chain of straight-line segments connecting (a_l, l) and $(a_{l+1}, l+1)$ $(l \in \{0, 1, \ldots, m-1\})$, and similarly for the right side and numbers b_l. The lower and upper boundary of T is the horizontal line with ordinate 0 and l, respectively. We denote that by $T = \text{inter}(S)$. The horizontal lines with ordinates $l \in \{0, 1, \ldots, m\}$ will be called *lanes*.

An *m-trapezoid graph* is a graph G on n vertices $\{v_1, \ldots, v_n\}$ which is an intersection graph of a set $\{T_1, \ldots, T_n\}$ of m-trapezoids, that is, $\{v_i, v_j\} \in E(G) \Leftrightarrow T_i \cap T_j \neq \emptyset$. Let G be an m-trapezoid graph represented by $\{T_1, \ldots, T_n\}$ where each

$$T_i = \text{inter}(\{(a_{li}, l), (b_{li}, l) : l \in \{0, 1, \ldots, m\}\}). \tag{1}$$

We will write $\tilde{a}_{li}$ (resp. $\tilde{b}_{li}$) for the point (a_{li}, l) (resp. (b_{li}, l)) in $\mathbf{R}^2$. We say that the left sides of T_i and T_j *cross* (or synonymously, *intersect*) if there are distinct indices $p, q \in \{0, 1, \ldots, m\}$ such that $a_{pi} < a_{pj}$ and $a_{qi} > a_{qj}$.

If G and G' are two m-trapezoid graphs, both on n vertices, represented by sets of m-trapezoids $\mathcal{T} = \{T_1, \ldots, T_n\}$ and $\mathcal{T}' = \{T'_1, \ldots, T'_n\}$ respectively, where the left side of T_i and the left side of T'_i coincide, that is $T_i = \text{inter}(\{\tilde{a}_{li}, \tilde{b}_{li} : l \in \{0, 1, \ldots, m\}\})$ and $T'_i = \text{inter}(\{\tilde{a}_{li}, \tilde{b}'_{li} : l \in \{0, 1, \ldots, m\}\})$, for all $i \in \{1, \ldots, n\}$, then we will say that $\mathcal{T}$ and $\mathcal{T}'$ are *left-coincidal*.

Recall that $d(T_i, T_\beta)$ and $d(T'_i, T'_\alpha)$ denote the distances between corresponding vertices in G and G' respectively. We now put $b^*_{li} = \max_{d(T'_i, T'_\alpha) \leq 1}\{b_{l\alpha}\}$ and $b^{*\prime}_{li} = \max_{d(T_i, T_\beta) \leq 1}\{b'_{l\beta}\}$ for each $i \in \{1, \ldots, n\}$ and $l \in \{0, 1, \ldots, m\}$.

Theorem 1. *For an integer $m \geq 0$ let G and G' be two m-trapezoid graphs on the same number of vertices, with left-coincidal representations $\{T_1, \ldots, T_n\}$ and $\{T'_1, \ldots, T'_n\}$ respectively. Assume further that for each i we have either $b^*_{li} \leq b^{*\prime}_{li}$ for all $l \in \{0, 1, \ldots, m\}$, or $b^*_{li} \geq b^{*\prime}_{li}$ for all $l \in \{0, 1, \ldots, m\}$. In this case, the pseudo product $G * G'$ is also an m-trapezoid graph with an m-trapezoid representation $\mathcal{T}^* = \{T^*_1, \ldots, T^*_n\}$, which is left-coincidal with both $\mathcal{T}$ and $\mathcal{T}'$, and where the right sides of each T^*_i are determined by b^{**}_{li} where $b^{**}_{li} = \max\{b_{li}, b'_{li}, \min\{b^*_{li}, b^{*\prime}_{li}\}\}$, for all $i \in \{1, \ldots, n\}$ and $l \in \{0, 1, \ldots, m\}$.*

Proof. To prove Theorem 1 we need to show

$$\{v_i, v_j\} \in E(G * G') \Leftrightarrow T^*_i \cap T^*_j \neq \emptyset. \tag{2}$$

We can assume that the left sides of T_i and T_j do not cross, say $a_{li} < a_{lj}$ for all $l \in \{0, 1, \ldots, m\}$, otherwise there is nothing to prove. Furthermore, if $\{v_i, v_j\}$ is either in $E(G)$ or in $E(G')$ then $T^*_i \cap T^*_j \neq \emptyset$ by definition. Hence, we can further assume

$$a_{li} < b_{li} < a_{lj} \text{ and } a_{li} < b'_{li} < a_{lj} \tag{3}$$

to hold for all $l \in \{0, 1, \ldots, m\}$ throughout the proof.

"⇒" in (2): By definition of $E(G * G')$, there are v_α and v_β such that $\{v_\alpha, v_j\}, \{v_i, v_\beta\} \in E(G)$ and $\{v_i, v_\alpha\}, \{v_\beta, v_j\} \in E(G')$. This, together with (3), means that there are indices $p, q \in \{0, 1, \ldots, m\}$ such that $T'_i \cap T'_\alpha \neq \emptyset$, $a_{pj} < b_{p\alpha}$, $T_i \cap T_\beta \neq \emptyset$ and $a_{qj} < b'_{q\beta}$. If $b^*_{pi} \leq b^{*\prime}_{pi}$ then we have $b^{**}_{pi} = b^*_{pi} \geq b_{p\alpha} > a_{pj}$, and hence $T^*_i \cap T^*_j \neq \emptyset$. If however $b^*_{pi} \geq b^{*\prime}_{pi}$ then by assumption in the theorem we have that $b^*_{qi} \geq b^{*\prime}_{qi}$ also holds and hence we have $b^{**}_{qi} = b^{*\prime}_{qi} \geq b'_{q\beta} > a_{qj}$, which implies that $T^*_i \cap T^*_j \neq \emptyset$.

"⇐" in (2): By (3) there is an $l \in \{0, 1, \ldots, m\}$ such that $b^{**}_{li} > a_{lj}$. By definition of b^*_{li} and $b^{*\prime}_{li}$ we can find α and β such that $T'_i \cap T'_\alpha \neq \emptyset$, $b_{l\alpha} = b^*_{li}$, $T_i \cap T_\beta \neq \emptyset$ and $b'_{l\beta} = b^{*\prime}_{li}$. Since now both $b_{l\alpha}$ and $b_{l\beta}$ are greater or equal to b^{**}_{li} we have $T'_i \cap T'_\alpha \neq \emptyset$, $b_{l\alpha} > a_{lj}$, $T_i \cap T_\beta \neq \emptyset$ and $b'_{l\beta} > a_{lj}$. By our assumption in (3) we have $T'_i \cap T'_\alpha \neq \emptyset$, $T_\alpha \cap T_j \neq \emptyset$, $T_i \cap T_\beta \neq \emptyset$ and $T'_\beta \cap T'_j \neq \emptyset$, which implies that $\{v_i, v_j\} \in E(G * G')$. □

Let us now consider the more special cases of a pseudo product of two powers of a fixed m-trapezoid graph G. By Proposition 1 we have that $G^s * G^t = G^{s+t}$, and hence Theorem 1 gives us a way to obtain the representation of G^{s+t} *directly* from the representations of G^s and G^t. In [1] it is shown that if G is an m-trapezoid graph represented by a set $\{T_1, \ldots, T_n\}$ of m-trapezoids (as in (1) and $k \geq 1$ is an integer, then G^k is represented by m-trapezoids $\{T_1(k), \ldots, T_n(k)\}$ which are given by

$$T_i(k) = \text{inter}(\{\tilde{a}_{li}, \tilde{b}_{li}(k) : l \in \{0, \ldots, m\}\}), \tag{4}$$

where $\tilde{b}_{li}(k) = \max_{d(T_\alpha, T_i) \leq k-1}\{b_{l\alpha}\}$. Although (4) provides a formula for the representation of G^k from the representation of G, this is not computationally feasible, since the definition of $\tilde{b}_{li}(k)$ is complex from a computational point of view. We are, however, able to compute precisely this representation much more efficiently, by applying the pseudo product.

Let $s, t \geq 1$ be integers, and G a fixed m-trapezoid graph. If G^s and G^t have $\{T_1(s), \ldots, T_n(s)\}$ and $\{T_1(t), \ldots, T_n(t)\}$, respectively, as their representations, then we can get the representation of the pseudo product $G^{s+t} = G^s * G^t$, given in Theorem 1, by calculating b^*_{li} explicitly and get

$$b^*_{li} = \max_{d(T_i(t), T_\alpha(t)) \leq 1}\{b_{l\alpha}(s)\} = \max_{d(T_i, T_\alpha) \leq t}\left\{\max_{d(T_\alpha, T_\beta) \leq s-1}\{b_{l\beta}\}\right\}$$

$$= \max_{d(T_i, T_\beta) \leq s+t-1}\{b_{l\beta}\} = b_{li}(s+t).$$

In the same way we get that $b^{*\prime}_{li} = b_{li}(s+t)$, and hence we have in the case for pseudo product of G^s and G^t that
$b^{**}_{li} = \max\{b_{li}, b'_{li}, \min\{b^*_{li}, b^{*\prime}_{li}\}\} = \max\{b_{li}(s), b_{li}(t), b_{li}(s+t)\} = b_{li}(s+t)$. Hence, the representation of G^{s+t} is $\{T_1(s+t), \ldots, T_n(s+t)\}$, as given in (4).

We see from the above that Theorem 1 applies when considering various powers of a fixed graph G, as the following observation shows.

Proposition 2. *If both G and G' are powers of the same m-trapezoid graph on n vertices, then $b^*_{li} = b^{*\prime}_{li}$ holds for all $l \in \{0, 1, \ldots, m\}$ and $i \in \{1, \ldots, n\}$.*

Recall that the pseudo product is associative on the set of powers of a fixed graph G, and therefore the notion $G^{r_1} * \cdots * G^{r_k}$ (k times) is perfectly sensible.

Corollary 1. *Let $k = \sum_{i=1}^{s} 2^{t_i}$ be the binary representation of k. For an m-trapezoid graph G represented by $\{T_1, \ldots, T_n\}$, the representation for $G^k = G^{2^{t_1}} * \cdots * G^{2^{t_s}}$ from Theorem 1 is $\{T_1(k), \ldots, T_n(k)\}$, the representation of G^k in (4).*

3 Computing Powers of m-Trapezoid Graphs

In this section we implement the theory of Section 2, to obtain a fast method of computing the representation of G^k, where $k \in \mathbf{N}$ and G is an m-trapezoid graph with a given representation. Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ and $\mathcal{T}' = \{T'_1, \ldots, T'_n\}$ be two such left-coincidal representations for G and G' respectively, as given by (1). Here we shall assume that for each lane $l \in \{0, 1, \ldots, m\}$ the endpoints, a_{li} and b_{li}, where $i \in \{1, \ldots, n\}$, have been translated to the set $\{1, 2, \ldots, 2n\}$.

We want to compute the pseudo product $G * G'$, whose right endpoints are denoted by b^{**}_{li} as in Theorem 1. We shall compute a series of $m+1$ by $2n$ matrices A_p, where for $p, l = 0, 1, \ldots, m$ and $q = 1, \ldots, 2n$, the entry $A_p[l, q]$ equals the rightmost coordinate along lane p among trapezoids T'_α in G' with $a_{l\alpha} \leq q$. Each trapezoid T_α that intersects T_i must satisfy $a_{l\alpha} < b_{li}$, for some l. Thus, given the values of A_p, we compute b^{**}_{pi} from Theorem 1 by setting $b^{**}_{pi} = \max\{b^{*}_{pi}{}', b_{pi}\}$ where $b^{*}_{pi}{}' \leftarrow \max_{l \in \{0,\ldots,m\}} A_p[l, b_{li}]$, which takes $m + 1$ operations. To compute A_p, we first initialize with zero and insert values for each trapezoid: For each $\alpha \in \{1, \ldots, n\}$ and $l \in \{0, \ldots, m\}$ let be $A_p[l, a_{l\alpha}] = b'_{l\alpha}$. This, together with the zero initialization, uses a total of $2(m + 1)n$ operations. We can then complete it in one pass from left to right, using the trivial observations that coordinates to the left of $q - 1$ are also to the left of q. That is, we form a prefix maxima of A_p by $A_p[l, q] \leftarrow \max(A_p[l, q], A_p[l, q - 1])$. This second loop also uses $2(m+1)n$ operations as l goes through $\{0, \ldots, m\}$ and q through $\{1, \ldots, 2n\}$, so we perform $4(m + 1)n$ operations to compute each matrix A_p. Therefore the computation of A_p where $p \in \{0, \ldots m\}$ takes a total of $4(m + 1)^2 n$ operations. Hence, by Proposition 1 and Theorem 1 we have the following.

Theorem 2. *Given powers G^s and G^t of an m-trapezoid graph G, the power graph G^{s+t} can be computed in $O(m^2 n)$ time.*

This generalizes the algorithm given in [2] for interval graphs. The same construction holds also for circular-arc and circular-trapezoid graphs, where the max operator is viewed in modular arithmetic.

If $k \in \mathbf{N}$ and $k = \sum_{i=1}^{s} 2^{t_i}$ is its binary representation, then $G^k = G^{2^{t_1}} * G^{2^{t_2}} * \cdots * G^{2^{t_s}}$. Using fast multiplication G^k can be computed in at most $t_s + s - 1 \leq 2 \log k - 1$ pseudo products. By Theorem 2 and Corollary 1 we have:

Corollary 2. *The representation of G^k where G is an m-trapezoid graph, can be computed in $O(m^2 n \log k)$ time.*

4 Computing Powers of Interval Graphs and Circular-Arc Graphs

Let G be an interval graph on n vertices, represented by a set $\mathcal{I}_G$ of n intervals. We may assume that all the intervals have their $2n$ endpoints distinct among the numbers $\{1, 2, \ldots, 2n\}$. For each interval $I \in \mathcal{I}_G$ there is a unique interval $I' \in \mathcal{I}_G$ with the rightmost endpoint of any interval which intersects I. This yields a mapping $f : \mathcal{I}_G \to \mathcal{I}_G$, defined by $f(I) = I'$. This mapping is acyclic and thus induces a directed forest $\boldsymbol{F}_G$ on $\mathcal{I}_G$ (which is a directed tree if G is connected), with an arc from each $I \in \mathcal{I}_G$ to $f(I)$. Note that the root of any tree of $\boldsymbol{F}_G$ will point to itself.

The representation of G^k can now be obtained quickly: For each interval $I = [a_I; b_I] \in \mathcal{I}_G$ we obtain an interval $I(k) = [a_{I(k)}; b_{I(k)}]$ where $a_{I(k)} = a_I$ and $b_{I(k)} = b_{I_k}$, where I_k is the k-th ancestor of I in the tree of the above forest $\boldsymbol{F}_G$ (where the parent of the root is the root itself).

This is computed in a single traversal of the tree. As we traverse the tree, we keep the nodes on the path from the root to the current node on an indexable stack. This is a data structure supporting all the stack operation, as well as constant-time indexing of elements in the stack. Namely, we use an array X, and as we traverse a node I at depth d_I, we store it in $X[d]$. Then, the root is stored in $X[0]$, and the k-th ancestor of v is stored at $X[d_I - k]$, for $k \leq d$. We obtain I_k simply as $X(\max\{d_I - k, 0\})$, and for each node I, we output new interval $I(k)$ obtained by $I(k) = [a_I; b_{X(\max\{d_I - k, 0\})}]$.

When G is a circular-arc graph, mapping f is a *pseudo forest* (or a *pseudo tree* if G is connected), i.e. each component contains exactly one cycle, as the number of edges equals the number of vertices. We must now treat nodes at depth less than k differently. Select any node R on the sole cycle to be a "root", and set its depth to be 0. Extend the array X to negative indices, and let $X[-1] = f(R)$ and generally $X[-i] = f^{(i)}(R)$. We now traverse the tree rooted at R, as before, and set I_k to be $X[d_I - k]$ for each node I of depth d_I from R. Otherwise, the process is identical. We have therefore the following.

Theorem 3. *Let G be a circular-arc graph with a given representation. For any k, we can compute the representation of the power graph G^k in $O(n)$ time.*

5 k-Independent Set and Dispersion Algorithms

By computing the k-th power of a graph, we reduce the problems k-IS and k-WIS to MIS and MWIS, respectively, on the corresponding class of graphs, within an additive factor of $O(n \log k)$. The following is known about those problems.

Proposition 3. *MWIS can be computed in $O(n)$ time for interval graphs, and in $O(n \log\log n)$ time for trapezoid graphs. MIS can be computed in $O(n)$ time for circular-arc graphs.*

For the MWIS result on interval graphs see [13]. The MIS result on circular-arc graphs has been rediscovered several times [12,14,15,21]. Felsner et al. [8]

showed that weighted IS of trapezoid graphs can be computed in $O(n \log n)$ time, when the representation is given. Their algorithm uses a data structure supporting Insert, Delete, Predecessor, and Successor operations of endpoints, and the complexity is equal to the complexity of n of each of these operations. With $O(n \log n)$ preprocessing, we may assume that all endpoints are integers from 1 to $2n$. Then, the data structure of van Emde Boas supports these operations in $\log \log n$ steps. Hence, we can compute the weighted IS of trapezoid graphs in $O(n \log \log n)$ time. Thus we obtain:

Theorem 4. *k-WIS can be found in $O(n)$ time for interval graphs, and in $O(n(\log \log n + \log k))$ time for trapezoid graphs. k-IS can be found in $O(n)$ time for circular-arc graphs.*

Proof. By Theorem 3 and Corollary 2 we can compute the k-th power of an interval graph and of a trapezoid graph in $O(n)$ time and $O(n \log k)$ time, respectively, and MWIS on the k-th power is equivalent to k-MWIS. This and Proposition 3 gives the results for these classes. The bound on k-IS for circular-arc graphs follows similarly. □

5.1 Dispersion via Binary Search for k

A simple algorithm looks for the largest power G^k of G that still admits an independent set of weight at least q. This is achieved by repeated doubling followed by binary search; the details are straightforward. This k is, of course, the solution to the dispersion problem. The time complexity is dominated by the number of computations of maximum (weighted) independent sets. Here, it is at most $2 \log k$. Hence:

Theorem 5. *For Weighted Dispersion we have the following time bounds: $O(n \log k)$ on interval graphs and $O(n \log k \log \log n)$ on trapezoid graphs.*

5.2 Unweighted Dispersion of Geometric Graphs

For convenience let k-IS(G) denote the size of a minimum k-independent set in graph G. Recall the notion of a lane from Section 2.

Lemma 1. *Let G be an m-trapezoid graph, and d be the distance between trapezoids that are furthest in each direction along some lane. Then $\lfloor d/(k+1) \rfloor + 1 \leq k\text{-}IS(G) \leq \lfloor d/(k-1) \rfloor + 1$.*

Proof. Let u (resp. u') be the trapezoid furthest to the left (resp. right) along a given lane, and let $P = \langle u = u_0, u_1, u_2, \ldots, u_d = u' \rangle$ be a shortest path between u and u'. The set $\{u_{i(k+1)} | i = 0, 1, 2, \ldots, \lfloor d/(k+1) \rfloor\}$ then forms a k-IS, thus showing the first part of the claim.

On the other hand, suppose $\{v_1, v_2, \ldots, v_t\}$ is a k-IS. For each v_i, the trapezoid representing v_i intersects some trapezoid representing a node u_{x_i} in the abovementioned path P. Since v_i and v_{i+1} are of distance at least $k+1$, we have that $x_{i+1} \geq x_i + k - 1$. It follows by induction that $d \geq x_t \geq x_1 + (t-1)(k-1) \geq (t-1)(k-1)$. Thus, $t \leq \lfloor d/(k-1) \rfloor + 1$, yielding the second part of the claim. □

Theorem 6. *Let G be an m-trapezoid graph, d be the distance between vertices respectively with the leftmost and rightmost endpoint along some lane, and K be $\lfloor d/(q-1) \rfloor$. Then, the optimum dispersion of G is one of the three values $\{K-1, K, K+1\}$.*

Proof. Let OPT be the optimum dispersion of G, i.e. the largest value t such that t-IS$(G) \geq q$. By the definition of K, $K(q-1) \leq d$, and thus by Lemma 1, $q \leq \lfloor d/K \rfloor + 1 \leq (K-1)$-IS$(G)$. That is, $OPT \geq K-1$. By the definition of K, $d/(K+1) < q$, so K is the largest number such that $\lfloor \frac{d}{K} \rfloor \geq q-1$. By Lemma 1, $q \leq OPT$-IS$(G) \leq \lfloor d/(OPT-1) \rfloor + 1$. Thus, $OPT \leq K+1$. □

This can be extended to circular-arc graphs. A greedy covering of the circle is defined as follows: Start with an arbitrary arc I, add $f(I)$ and let $I := f(I)$, until the whole circle is covered. (Do not put the initial I in the set.) Such a covering exists unless the graph is actually an interval graph. Note that a greedy covering is a chordless cycle in the graph and can be computed in $O(n)$ time. The following result holds by an argument similar to Lemma 1.

Lemma 2. *Let c be the size of a greedy covering of a circular-arc graph G. Then $\lfloor c/(k+1) \rfloor \leq k$-IS$(G) \leq \lfloor c/(k-1) \rfloor$.*

This can be further extended to circular m-trapezoid graphs. Due to lack of space, we only state the result:

Theorem 7. *Let G be a circular m-trapezoid graph, and let K be $\lfloor c/q \rfloor$. Then, the optimum dispersion of G is one of the three values $\{K-1, K, K+1\}$.*

The proof follows the lines of Theorem 6. On circular-arc graphs, we can compute each k-IS in linear time, as mentioned earlier. Thus we finally get the following

Corollary 3. *Dispersion has equivalent complexity as k-IS on interval, circular-arc, m-trapezoid, and circular m-trapezoid graphs. In particular, it can be computed in $O(n(\log k + (\log\log n)^m))$ time on m-trapezoid graphs, and in $O(n)$ time on interval and circular-arc graphs.*

Acknowledgments. Parts of this work were done while Geir was a Visiting Scholar at Los Alamos National Laboratory in Los Alamos, New Mexico, Summer of 2000. He is grateful to Madhav Marathe for his hospitality. We thank Rasmus Pagh for advice.

References

1. G. Agnarsson. On Powers of some Intersection Graphs, *Congressus Numerantium*, submitted in April 2001.
2. G. Agnarsson, R. Greenlaw, M. M. Halldórsson. On Powers of Chordal Graphs and Their Colorings, *Congressus Numerantium*, **144**:41–65, (2000).
3. B. K. Bhattacharya, M. E. Houle. Generalized Maximum Independent Sets for Trees. *Computing - The 3rd Australian Theory Symposium CATS'97.*
4. B. K. Bhattacharya, M. E. Houle. Generalized Maximum Independent Sets for Trees in Subquadratic Time. *10th Int. Symp. on Algorithms and Computation ISAAC'99*, *LNCS* 1741 (Springer), 435–445.

5. E. Dahlhaus and P. Duchet. On Strongly Chordal Graphs. *Ars Combinatoria*, **24 B**:23–30, (1987).
6. P. Damaschke. Distances in Cocomparability Graphs and Their Powers. *Discrete Applied Mathematics*, **35**:67–72, (1992).
7. P. Damaschke. Efficient Dispersion Algorithms for Geometric Intersection Graphs. *26th Int. Workshop on Graph-Theoretic Concepts in Computer Science WG'2000*, *LNCS* 1928 (Springer), 107–115.
8. S. Felsner, R. Müller and L. Wernisch. Trapezoid Graphs and Generalizations, Geometry and Algorithms. *Discrete Applied Mathematics*, **74**:13–32, (1997).
9. C. Flotow. On Powers of m-Trapezoid Graphs. *Discrete Applied Mathematics*, **63**:187–192, (1995).
10. C. Flotow. On Powers of Circular Arc graphs and Proper Circular Arc Graphs. *Discrete Applied Mathematics*, **74**:199–207, (1996).
11. J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, **182**:105–142, (1999).
12. W. L. Hsu, K. H. Tsai. Linear-time Algorithms on Circular Arc Graphs. *Information Proc. Letters*, **40**:123–129, (1991).
13. J. Y. Hsiao, C. Y. Tang, R. S. Chang. An Efficient Algorithm for Finding a Maximum Weight 2-Independent Setw on Interval Graphs. *Information Proc. Letters*, **43**:229–235, (1992).
14. D. T. Lee, M. Sarrafzadeh, Y. F. Wu. Minimum Cuts for Circular-Arc Graphs. *SIAM J. Computing*, **19**:1041–1050, (1990).
15. S. Masuda and K. Nakajima. An Optimal Algorithm for Finding a Maximum Independent Set of a Circular-Arc Graph. *SIAM J. Computing*, **17**:219–230, (1988).
16. A. Raychaudhuri. On Powers of Interval and Unit Interval Graphs. *Congressus Numerantium*, **59**:235–242, (1987).
17. A. Raychaudhuri. On Powers of Strongly Chordal and Circular Graphs. *Ars Combinatoria*, **34**:147–160, (1992).
18. D. J. Rosenkrantz, G. K. Tayi, S. S. Ravi. Facility Dispersion Problems Under Capacity and Cost Constraints. *J. of Combinatorial Optimization*, **4**:7–33 (2000).
19. D. B. West. Introduction to Graph Theory. *Prentice-Hall Inc.*, Upper Saddle River, New Jersey, (1996).
20. M. Yannakakis. The complexity of the partial order dimension problem. *SIAM J. Alg. Disc. Meth.* **3**:351–358, 1982.
21. S. Q. Zheng. Maximum Independent Sets of Circular Arc Graphs: Simplified Algorithms and Proofs. *Networks*, **28**:15–19, (1996).

Efficient Data Reduction for Dominating Set: A Linear Problem Kernel for the Planar Case

Jochen Alber*[1], Michael R. Fellows[2], and Rolf Niedermeier[1]

[1] Universität Tübingen, Wilhelm-Schickard-Institut für Informatik, Sand 13, D-72076 Tübingen, Fed. Rep. of Germany, alber,niedermr@informatik.uni-tuebingen.de

[2] University of Newcastle, School of Electrical Engineering and Computer Science, University Drive, Callaghan, NSW 2308, Australia, mfellows@cs.newcastle.edu.au

Abstract. Dealing with the NP-complete Dominating Set problem on undirected graphs, we demonstrate the power of data reduction by preprocessing from a theoretical as well as a practical side. In particular, we prove that Dominating Set on planar graphs has a so-called problem kernel of linear size, achieved by two simple and easy to implement reduction rules. This answers an open question from previous work on the parameterized complexity of Dominating Set on planar graphs.

1 Introduction

In this work, two lines of research meet. On the one hand, there is Dominating Set, one of the NP-complete core problems of combinatorial optimization and graph theory. According to a 1998 survey [12, Chapter 12], more than 200 research papers and more than 30 PhD theses investigate the algorithmic complexity of domination and related problems [15]. Moreover, domination problems occur in numerous practical settings, ranging from strategic decisions such as locating radar stations or emergency services through computational biology to voting systems (see [14] for a survey). On the other hand, the second line of research is that of algorithm engineering and, in particular, the power of data reduction by efficient preprocessing. Weihe [16,17] gave a striking example when dealing with the closely related NP-complete Hitting Set problem in context of the European railroad network. In a preprocessing phase, he applied two simple data reduction rules again and again until no further application was possible. The impressive result of his empirical study was that each of his real-world instances was broken into very small pieces such that for each of these a simple brute-force approach was sufficient to solve the hard problems efficiently and optimally. Here, we present two easy to implement reduction rules for Dominating Set and analytically (not only empirically) substantiate their strength in the case of planar graphs. More precisely, we can prove a linear size problem

* Work supported by the Deutsche Forschungsgemeinschaft (DFG), research project PEAL (Parameterized complexity and Exact ALgorithms), NI 369/1-1,1-2.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 150–159, 2002.

kernel for Dominating Set on planar graphs. This parallels results on a linear problem kernel for the Vertex Cover problem observed by Chen *et al.* [8] based on a well-known theorem of Nemhauser and Trotter [13,7].

A *k-dominating set* D of an undirected graph G is a set of k vertices of G such that each of the rest of the vertices has at least one neighbor in D. The minimum k such that G has a k-dominating set is called the *domination number* of G, denoted by $\gamma(G)$. The Dominating Set problem is to decide, given a graph $G = (V, E)$ and a positive integer k, whether $\gamma(G) \leq k$. Dominating Set belongs to the best-studied problems in parameterized complexity theory [5,10, 11]. Here, the fundamental question is whether a given parameterized problem is *fixed-parameter tractable*, i.e., whether it can be solved in time $f(k) \cdot n^{O(1)}$, that is, time polynomial with respect to the input except for the parameter k where exponential behavior is allowed. It is well-known that Dominating Set is probably *not* fixed-parameter tractable on general graphs, more precisely, Dominating Set is W[2]-complete [9,10]. By way of contrast, restricting it to planar graphs (i.e., those graphs that can be drawn in the plane without edge crossing), where it still remains NP-complete, Dominating Set becomes fixed-parameter tractable. Currently, the two best known results in this respect are a time $O(c^{\sqrt{k}} \cdot n)$ (with $c = 4^{6\sqrt{34}}$) algorithm based on tree decompositions [1][1] and a time $O(8^k \cdot n)$ search tree algorithm [2]. In both these works, it remained an open question to show whether Dominating Set on planar graphs possesses a linear problem kernel. More precisely, the question was whether, given a planar graph G and a parameter k as input, can we—in a polynomial time preprocessing phase—construct a planar graph G' and a parameter $k' \leq k$ such that

1. G' consists of only $c \cdot k$ vertices for some constant c, and
2. G has a dominating set of size k iff G' has a dominating set of size k'.[2]

We answer this question affirmatively, in this way also providing both easy and strong data reduction rules in the sense of Weihe [16,17].

Our main result is that Dominating Set on planar graphs has a problem kernel of size $335k$. Note, however, that our main concern in analyzing the multiplicative constant 335 was conceptual simplicity, for which we deliberately sacrificed the aim to further lower it by way of refined analysis (without changing the reduction rules). Our problem kernel can be constructed by applying two simple data reduction rules again and again until no further application is possible. In the worst case, for a planar input graph of n vertices this data reduction needs time $O(n^2)$. Besides answering an open question from previous work, the linear problem kernel also leads to further improvements of known results. First, on the structural side, combining our linear problem kernel with the graph separator

[1] In [1], an exponential base $c = 3^{6\sqrt{34}}$ is stated, involving a tiny flaw in the analysis. The correct worst case upper bound should read $c = 4^{6\sqrt{34}}$ (see also [6]).

[2] Usually, one also wants to efficiently get the dominating set for G once having the dominating set for G'. This is easily achieved in our and basically all known reductions to problem kernel.

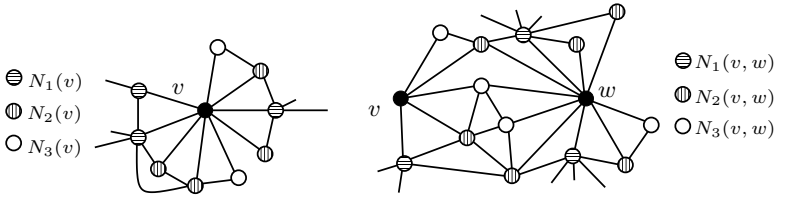


Fig. 1. The left-hand side shows the partitioning of the neighborhood of a single vertex v. The right-hand side shows the partitioning of a neighborhood $N(v,w)$ of two vertices v and w. Since, in the left-hand figure, $N_3(v) \neq \emptyset$, reduction Rule 1 applies. In the right-hand figure, since $N_3(v,w)$ cannot be dominated by a single vertex at all, Case 2 of Rule 2 applies

approach presented in [4] immediately results in an $O(c^{\sqrt{k}} \cdot k + n^{O(1)})$ DOMINATING SET algorithm on planar graphs (for some constant c). Also, the linear problem kernel directly proves the so-called "Layerwise Separation Property" [3] for DOMINATING SET on planar graphs, again implying an $O(c^{\sqrt{k}} \cdot k + n^{O(1)})$ algorithm. Second, the linear problem kernel improves the time $O(8^k \cdot n)$ search tree algorithm from [2] to an $O(8^k k + n^3)$ algorithm. Apart from these theoretical results, we also underpin the practical importance of our approach through (ongoing) experimental studies. These indicate that the two proposed reduction rules can be implemented efficiently and lead to massive reductions on given input data. Notably, our data reductions also significantly accelerate implemented tree decomposition based algorithms for DOMINATING SET on planar graphs [6] due to an empirically observed reduction of the treewidth of the tested graphs. Due to the lack of space, many details had to be omitted and will appear in the full version of the paper.

2 The Reduction Rules

We present two reduction rules for DOMINATING SET. Both reduction rules are based on the same principle: We explore the local structure of the graph and try to replace it by a simpler structure.

2.1 The Neighborhood of a Single Vertex

Consider a vertex $v \in V$ of the given graph $G = (V, E)$. We partition the vertices of the neighborhood $N(v)$ of v into three different sets $N_1(v)$, $N_2(v)$, and $N_3(v)$ depending on what neighborhood structure these vertices have. More precisely, setting $N[v] := N(v) \cup \{v\}$, we define

$$N_1(v) := \{u \in N(v) \,:\, N(u) \setminus N[v] \neq \emptyset\},$$
$$N_2(v) := \{u \in N(v) \setminus N_1(v) \,:\, N(u) \cap N_1(v) \neq \emptyset\},$$
$$N_3(v) := N(v) \setminus (N_1(v) \cup N_2(v)).$$

An example which illustrates the partitioning of $N(v)$ into the subsets $N_1(v)$, $N_2(v)$, and $N_3(v)$ can be seen in the left-hand diagram of Fig. 1. Based on the above definitions we give our first reduction rule.

Rule 1 *If $N_3(v) \neq \emptyset$ for some vertex v, then*

- *remove $N_2(v)$ and $N_3(v)$ from G and*
- *add a new vertex v' with the edge $\{v, v'\}$.*

Lemma 1. *Let $G = (V, E)$ be a graph and let $G' = (V', E')$ be the resulting graph after having applied Rule 1 to G. Then $\gamma(G) = \gamma(G')$.*

Proof. Consider a vertex $v \in V$ such that $N_3(v) \neq \emptyset$. The vertices in $N_3(v)$ can only be dominated by either v or by vertices in $N_2(v) \cup N_3(v)$. But, clearly, $N(w) \subseteq N(v)$ for every $w \in N_2(v) \cup N_3(v)$. This shows that an optimal way to dominate $N_3(v)$ is given by taking v into the dominating set. This is simulated by the "gadget" $\{v, v'\}$ in G'. It is safe to remove $N_2(v) \cup N_3(v)$, since these vertices need not to be used in an optimal dominating set. Hence, $\gamma(G') = \gamma(G)$. □

Lemma 2. *Rule 1 can be carried out in time $O(n)$ for planar graphs and in time $O(n^3)$ for general graphs.* □

2.2 The Neighborhood of a Pair of Vertices

Similar to Rule 1, we explore the set $N(v, w) := N(v) \cup N(w)$ of two vertices $v, w \in V$. Analogously, we now partition $N(v, w)$ into three disjoint subsets $N_1(v, w)$, $N_2(v, w)$, and $N_3(v, w)$. Setting $N[v, w] := N[v] \cup N[w]$, we define

$$N_1(v, w) := \{u \in N(v, w) \mid N(u) \setminus N[v, w] \neq \emptyset\},$$
$$N_2(v, w) := \{u \in N(v, w) \setminus N_1(v, w) \mid N(u) \cap N_1(v, w) \neq \emptyset\},$$
$$N_3(v, w) := N(v, w) \setminus (N_1(v, w) \cup N_2(v, w)).$$

The right-hand diagram of Fig. 1 shows an example which illustrates the partitioning of $N(v, w)$ into the subsets $N_1(v, w)$, $N_2(v, w)$, and $N_3(v, w)$.
Our second reduction rule—compared to Rule 1—is slightly more complicated.

Rule 2 *Consider $v, w \in V$ $(v \neq w)$ and suppose that $N_3(v, w) \neq \emptyset$. Suppose that $N_3(v, w)$ cannot be dominated by a single vertex from $N_2(v, w) \cup N_3(v, w)$.*

Case 1 *If $N_3(v, w)$ can be dominated by a single vertex from $\{v, w\}$:*

(1.1) *If $N_3(v, w) \subseteq N(v)$ as well as $N_3(v, w) \subseteq N(w)$:*
- *remove $N_3(v, w)$ and $N_2(v, w) \cap N(v) \cap N(w)$ from G and*
- *add two new vertices z, z' and edges $\{v, z\}, \{w, z\}, \{v, z'\}, \{w, z'\}$.*

(1.2) *If $N_3(v, w) \subseteq N(v)$, but not $N_3(v, w) \subseteq N(w)$:*
- *remove $N_3(v, w)$ and $N_2(v, w) \cap N(v)$ from G and*
- *add a new vertex v' and the edge $\{v, v'\}$ to G.*

(1.3) *If $N_3(v, w) \subseteq N(w)$, but not $N_3(v, w) \subseteq N(v)$:*
- *remove $N_3(v, w)$ and $N_2(v, w) \cap N(w)$ from G and*

- *add a new vertex w' and the edge $\{w, w'\}$ to G.*

Case 2 *If $N_3(v, w)$ cannot be dominated by a single vertex from $\{v, w\}$:*

- *remove $N_3(v, w)$ and $N_2(v, w)$ from G and*
- *add two new vertices v', w' and edges $\{v, v'\}$, $\{w, w'\}$.*

Lemma 3. *Let $G = (V, E)$ be a graph and let $G' = (V', E')$ be the resulting graph after having applied Rule 2 to G. Then $\gamma(G) = \gamma(G')$.*

Proof. Similar to the proof of Lemma 1, we observe that vertices from $N_3(v, w)$ can only be dominated by vertices from $M := \{v, w\} \cup N_2(v, w) \cup N_3(v, w)$. All cases in Rule 2 are based on the fact that $N_3(v, w)$ needs to be dominated. All rules only apply if there is not a *single* vertex in $N_2(v, w) \cup N_3(v, w)$ which dominates $N_3(v, w)$.

We first of all discuss the correctness of Case (1.2) (and similarly the symmetric Case (1.3)): If v dominates $N_3(v, w)$ (and w does not), then it is better to take v into the dominating set—and at the same time still leave the option of taking vertex w—than to take any combination of two vertices $\{x, y\}$ from the set $M \setminus \{v\}$. It may be that we still have to take w to a minimum dominating set, but in any case $\{v, w\}$ dominates at least as many vertices as $\{x, y\}$. The "gadget" $\{v, v'\}$ simulates the effect of taking v. It is safe to remove $N := (N_2(v, w) \cap N(v)) \cup N_3(v, w)$ since, by taking v into the dominating set, all vertices in N are already dominated and since, as discussed above, it is always better to take $\{v, w\}$ into a minimum dominating set than to take v and any other of the vertices from N.

In the situation of Case (1.1), we can dominate $N_3(v, w)$ by both v or w. Since we cannot decide at this point which of these vertices should be chosen to be in the dominating set, we use the "gadget" with vertices v' and w' which simulates a choice between v or w, as can be seen easily. In any case, however, it is better to take one of the vertices v and w (maybe both) than taking any two of the vertices from $M \setminus \{v, w\}$. The argument for this is similar to the one for Case (1.2). The removal of $N_3(v, w) \cup (N_2(v, w) \cap N(v) \cap N(w))$ is safe by a similar argument than the one that justified the removal of N in Case (1.2).

In Case 2, we need at least two vertices to dominate $N_3(v, w)$. Since $N(v, w) \supseteq N(x, y)$ for all pairs $x, y \in M$ it is best to take v and w into the dominating set, simulated by the gadgets $\{v, v'\}$ and $\{w, w'\}$. As in the previous cases removing $N_3(v, w) \cup N_2(v, w)$ is safe since these vertices are already dominated and since these vertices need not be used for an optimal dominating set. □

Lemma 4. *Rule 2 can be carried out in time $O(n^2)$ for planar graphs and in time $O(n^4)$ for general graphs.* □

2.3 Reduced Graphs

Definition 1. *Let $G = (V, E)$ be a graph such that both the application of Rule 1 and the application of Rule 2 leave the graph unchanged. Then we say that G is* reduced *with respect to these rules.*

Observing that the (successful) application of any reduction rule always "shrinks" the given graph implies that there can only be $O(n)$ successful applications of reduction rules. This leads to the following.[3]

Lemma 5. *A graph G can be transformed into a reduced graph G' with $\gamma(G) = \gamma(G')$ in time $O(n^3)$ in the planar case and in time $O(n^5)$ in the general case.* □

3 A Linear Problem Kernel for Planar Graphs

Here, we show that the reduction rules given in Section 2.1 yield a linear size problem kernel for DOMINATING SET on planar graphs.

Theorem 1. *For a planar graph $G = (V, E)$ which is reduced with respect to Rules 1 and 2, we get $|V| \leq 335\,\gamma(G)$, i.e., the* DOMINATING SET *problem on planar graphs admits a linear problem kernel.*

The rest of this section is devoted to sketch a proof of Theorem 1. The proof can be split into two parts. In a first step, we try to find a so-called "maximal region decomposition" of the vertices V of G. In a second step, we show, on the one hand, that such a maximal region decomposition must contain all but $O(\gamma(G))$ many vertices from V. On the other hand, we prove that such a region decomposition uses at most $O(\gamma(G))$ regions, each of which having size $O(1)$. Combining the results then yields $|V| = O(\gamma(G))$.

3.1 Finding a Maximal Region Decomposition

Suppose that we have a reduced planar graph G with a minimum dominating set D. We know that, in particular, neither Rule 1 applies to a vertex $v \in D$, nor Rule 2 applies to a pair of vertices $v, w \in D$. We want to get our hands on the number of vertices that lie in neighborhoods $N(v)$ for $v \in D$, or neighborhoods $N(v, w)$ for $v, w \in D$. A first idea to prove that $|V| = O(|D|)$ would be to find (at most $O(|D|)$ many) neighborhoods $N(v_1, w_1), \ldots, N(v_\ell, w_\ell)$ with $v_i, w_i \in D$, such that all vertices in V lie in at least one such neighborhood; and then use the fact that G is reduced in order to prove that each $N(v_i, w_i)$ has size at most $O(1)$. However, even if the graph G is reduced, the neighborhoods $N(v, w)$ of two vertices $v, w \in D$ may contain many vertices: the size of $N(v, w)$ in a reduced graph basically depends on how big $N_1(v, w)$ is.

In order to circumvent these difficulties, we define the concept of a region $R(v, w)$ for which we can guarantee that in a reduced graph it consists of only a constant number of vertices.

Definition 2. *Let $G = (V, E)$ be a plane*[4] *graph. A* region *$R(v, w)$ between two vertices v, w is a closed subset of the plane with the following properties:*

1. *the boundary of $R(v, w)$ is formed by two paths P_1 and P_2 in V which connect v and w, and the length of each path is at most three*[5]*, and*

[3] It must be emphasized here that our polynomial time bounds for the reduction rules give real worst-case bounds (which may not even be tight) and, in practice, the algorithms implementing these rules appear to be much faster.

[4] A plane graph is a particular planar embedding of a planar graph.

[5] The length of a path is the number of edges on it.

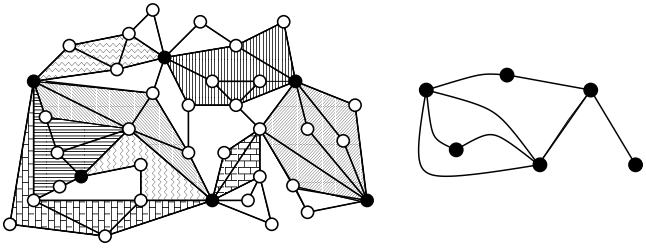


Fig. 2. The left-hand side diagram shows an example of a possible D-region decomposition $\mathcal{R}$ of some graph G, where D is the subset of vertices in G that are drawn in black. The various regions are highlightened by different patterns. The remaining white areas are not considered as regions. Note that the given D-region decomposition is maximal. The right-hand side shows the induced graph $G_{\mathcal{R}}$ (Definition 4)

2. all vertices which are strictly inside[6] *the region* $R(v,w)$ *are from* $N(v,w)$.

For a region R, *let* $V(R)$ *denote the vertices belonging to* R, *i.e.,*

$$V(R) := \{u \in V \mid u \textit{ sits inside or on the boundary of } R\}.$$

Definition 3. *Let* $G = (V,E)$ *be a plane graph and* $D \subseteq V$. *A* D-region decomposition *of* G *is a set* $\mathcal{R}$ *of regions between pairs of vertices in* D, *such that*

1. *for* $R(v,w) \in \mathcal{R}$ *no vertex from* D *(except for* v,w*) lies in* $V(R(v,w))$, *and*
2. *no two regions* $R_1, R_2 \in \mathcal{R}$ *do intersect (however, they may touch each other by having common boundaries).*

For a D*-region decomposition* $\mathcal{R}$, *we define* $V(\mathcal{R}) := \bigcup_{R\in\mathcal{R}} V(R)$. *A* D*-region decomposition* $\mathcal{R}$ *is called* maximal *if there is no region* $R \notin \mathcal{R}$ *such that* $\mathcal{R}' := \mathcal{R} \cup \{R\}$ *is a* D*-region decomposition with* $V(\mathcal{R}) \subsetneq V(\mathcal{R}')$.

For an example of a (maximal) D-region decomposition we refer to the left-hand side diagram of Fig. 2. It is not directly clear, whether, for a given graph G with dominating set D, a maximal D-region decomposition of G exists. We will see that this indeed is the case. Moreover, we will show that we can even find a special maximal D-region decomposition. For that purpose, we observe that a D-region decomposition induces a graph in a very natural way.

Definition 4. *The* induced graph $G_{\mathcal{R}} = (V_{\mathcal{R}}, E_{\mathcal{R}})$ *of a* D*-region decomposition* $\mathcal{R}$ *of* G *is the graph with possible multiple edges which is defined as follows:* $V_{\mathcal{R}} := D$, *and* $E_{\mathcal{R}} := \{\{v,w\} \mid$ *there is a region* $R(v,w) \in \mathcal{R}$ *between* $v,w \in D\}$.

Note that, by Definition 3, the induced graph $G_{\mathcal{R}}$ of a D-region decomposition is planar. For an example of an induced graph $G_{\mathcal{R}}$ see Fig. 2.

[6] i.e., not sitting on the boundary of $R(v,w)$

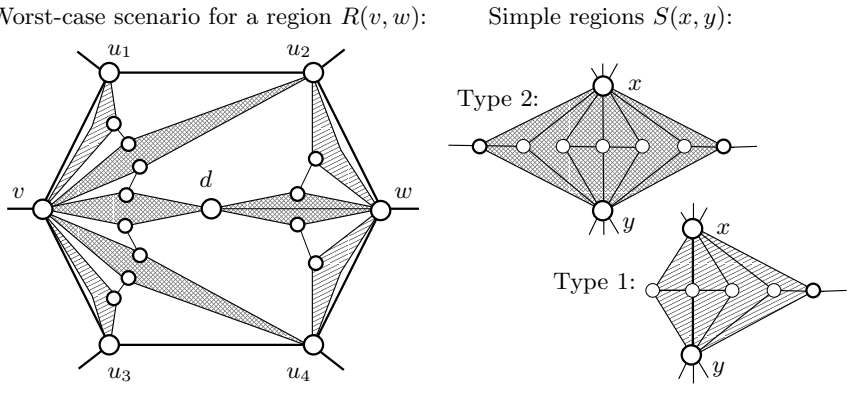


Fig. 3. The left-hand diagram shows a worst-case scenario for a region $R(v,w)$ between two vertices v and w in a reduced planar graph (cf. the proof of Proposition 3). Such a region may contain up to four vertices from $N_1(v,w)$, namely u_1, u_2, u_3, and u_4. The vertices from $R(v,w)$ which belong to the sets $N_2(v,w)$ and $N_3(v,w)$ can be grouped into so-called simple regions of Type 1 (marked with a line-pattern) or of Type 2 (marked with a crossing-pattern); the structure of such simple regions $S(x,y)$ is given in the right-hand part of the diagram. In $R(v,w)$ there might be two simple regions $S(d,v)$ and $S(d,w)$ (of Type 2), containing vertices from $N_3(v,w)$. And, we can have up to six simple regions of vertices from $N_2(v,w)$: $S(u_1,v), S(v,u_3), S(u_4,w), S(w,u_2), S(u_2,v)$, and $S(u_4,v)$ (among these, the latter two can be of Type 2 and the others are of Type 1)

Definition 5. *We say that a planar graph $G = (V,E)$ with multiple edges is* thin*, if there exists a planar embedding such that the following property holds: If there are two edges e_1, e_2 between a pair of distinct vertices $v, w \in V$, then there must be two further vertices $u_1, u_2 \in V$ which sit inside the two disjoint regions of the plane that are enclosed by e_1, e_2.*

Lemma 6. *For a thin planar graph $G = (V,E)$ we have $|E| \leq 3|V| - 6$.*

Proof. The claim is true for planar graphs without multiple edges. An easy induction on the number of multiple edges in G proves the claim. □

Using the notion of thin graphs, we can formulate the main result of this section.

Proposition 1. *For a reduced plane graph G with dominating set D, there exists a maximal D-region decomposition $\mathcal{R}$ such that $G_{\mathcal{R}}$ is thin.* □

3.2 Region Decompositions and the Size of Reduced Planar Graphs

Suppose that we are given a reduced planar graph $G = (V,E)$ with a minimum dominating set D. Then, by Proposition 1 and Lemma 6, we can find a maximal D-region decomposition $\mathcal{R}$ of G with at most $O(\gamma(G))$ regions. In order to see

that $|V| = O(\gamma(G))$, it remains to show that (1) there are at most $O(\gamma(G))$ vertices which do not belong to any of the regions in $\mathcal{R}$, and that (2) every region of $\mathcal{R}$ contains at most $O(1)$ vertices. These issues are treated by the following two propositions, the extensive proofs of which are omitted.

Proposition 2. *Let $G = (V, E)$ be a plane reduced graph and let D be a dominating set of G. If $\mathcal{R}$ is a maximal D-region decomposition, then $\mathcal{R}$ contains all but $O(|D| + |\mathcal{R}|)$ vertices of G. More precisely, $|V \setminus V(\mathcal{R})| \leq 2|D| + 56|\mathcal{R}|$.* □

We now investigate the maximal size of a region in a reduced graph. The worst-case scenario for a region in a reduced graph is depicted in Fig. 3.

Proposition 3. *A region R of a plane reduced graph contains at most 55 vertices, i.e., $|V(R)| \leq 55$.* □

To prove Theorem 1 we first of all observe that, for a graph G with minimum dominating set D, by Proposition 1 and Lemma 6, we can find a D-region decomposition $\mathcal{R}$ of G with at most $3\gamma(G)$ regions, i.e., $|\mathcal{R}| \leq 3\gamma(G)$. By Proposition 3, we know that $|V(\mathcal{R})| \leq \sum_{R \in \mathcal{R}} |V(R)| \leq 55|\mathcal{R}|$. By Proposition 2, we have $|V \setminus V(\mathcal{R})| \leq 2|D| + 56|\mathcal{R}|$. Hence, we get $|V| \leq 2|D| + 111|\mathcal{R}| \leq 335\,\gamma(G)$.

4 Conclusion

Presenting two simple and easy to implement reduction rules for Dominating Set, we proved that for planar graphs a linear size problem kernel can be efficiently constructed. Our result complements and partially improves previous results [1,2,3,4] on the parameterized complexity of Dominating Set on planar graphs. We emphasize that the proven bound on the problem kernel size is a pure worst-case upper bound. In first experimental studies to be reported elsewhere, we obtained much smaller problem kernels. An immediate open question is whether or not we can improve the constant factor to values around 10. This would bring the problem kernel for Dominating Set on planar graphs into "dimensions" as known for Vertex Cover, where it is of size $2k$ [8]. This could be done by either improving the analysis given or (more importantly) further improving the given reduction rules or both. Improving the rules should be doable by further extending the concept of neighborhood to more than two vertices. From a practical point of view, however, one also has to take into account to keep the reduction rules as simple as possible in order to avoid inefficiency due to increased overhead. It might well be the case that additional, more complicated reduction rules only improve the worst case bounds, but are of little or no practical use due to their computational overhead.

Acknowledgements. For two years, besides ourselves the linear size problem kernel question for Dominating Set on planar graphs has taken the attention of numerous people, all of whom we owe sincere thanks for their insightful and inspiring remarks and ideas. Among these people we particularly would like to mention Frederic Dorn, Henning Fernau, Jens Gramm, Michael Kaufmann, Ton Kloks, Klaus Reinhardt, Fran Rosamond, Peter Rossmanith, Ulrike Stege, and

Pascal Tesson. Special thanks go to Henning for the many hours he spent with us on "diamond discussions" and for pointing us to a small error concerning the application of the linear problem kernel, and to Frederic for again doing a perfect implementation job, which also uncovered a small error in a previous version of Rule 2.

References

1. J. Alber, H. L. Bodlaender, H. Fernau, and R. Niedermeier. Fixed parameter algorithms for planar dominating set and related problems. In *Proc. 7th SWAT 2000*, Springer-Verlag LNCS 1851, pp. 97–110, 2000.
2. J. Alber, H. Fan, M. R. Fellows, H. Fernau, R. Niedermeier, F. Rosamond, and U. Stege. Refined search tree technique for DOMINATING SET on planar graphs. In *Proc. 26th MFCS 2001*, Springer-Verlag LNCS 2136, pp. 111-122, 2001.
3. J. Alber, H. Fernau, and R. Niedermeier. Parameterized complexity: exponential speed-up for planar graph problems. In *Proc. 28th ICALP 2001*, Springer-Verlag LNCS 2076, pp. 261–272, 2001.
4. J. Alber, H. Fernau, and R. Niedermeier. Graph separators: a parameterized view. In *Proc. 7th COCOON 2001*, Springer-Verlag LNCS 2108, pp. 318–327, 2001.
5. J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, **229**: 3–27, 2001.
6. J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proc. 5th LATIN 2002*, Springer-Verlag LNCS 2286, pp. 613-627, 2002.
7. R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, **25**: 27–46, 1985.
8. J. Chen, I. A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, **41**:280–301, 2001.
9. R. G. Downey and M. R. Fellows. Parameterized computational feasibility. In *P. Clote, J. Remmel (eds.): Feasible Mathematics II*, pp. 219–244. Birkhäuser, 1995.
10. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer-Verlag, 1999.
11. M. R. Fellows. Parameterized complexity: the main ideas and some research frontiers. In *Proc. 12th ISAAC 2001*, Springer-Verlag LNCS 2223, pp. 291–307, 2001.
12. T. W. Haynes, S. T. Hedetniemi, and P. J. Slater. *Fundamentals of Domination in Graphs*. Monographs and textbooks in pure and applied Mathematics Vol. 208, Marcel Dekker, 1998.
13. G. L. Nemhauser and L. E. Trotter. Vertex packing: structural properties and algorithms. *Mathematical Programming*, **8**:232–248, 1975.
14. F. S. Roberts. *Graph Theory and Its Applications to Problems of Society*. SIAM Press 1978. Third printing 1993 by Odyssey Press.
15. J. A. Telle. Complexity of domination-type problems in graphs. *Nordic J. Comput.* **1**:157–171, 1994.
16. K. Weihe. Covering trains by stations or the power of data reduction. In *Proc. 1st ALEX'98*, pp. 1–8, 1998.
17. K. Weihe. On the differences between "practical" and "applied" (invited paper). In *Proc. WAE 2000*, Springer-Verlag LNCS 1982, pp. 1–10, 2001.

Planar Graph Coloring with Forbidden Subgraphs: Why Trees and Paths Are Dangerous*

Hajo Broersma[1], Fedor V. Fomin[2], Jan Kratochvíl[3], and Gerhard J. Woeginger[1]

[1] Faculty of Mathematical Sciences, University of Twente, 7500 AE Enschede, The Netherlands, {broersma, g.j.woeginger}@math.utwente.nl
[2] Heinz Nixdorf Institut, Fürstenallee 11, D-33102 Paderborn, Germany, fomin@uni-paderborn.de
[3] Faculty of Mathematics and Physics, Charles University, 118 00 Prague, Czech Republic, honza@kam.ms.mff.cuni.cz

Abstract. We consider the problem of coloring a planar graph with the minimum number of colors such that each color class avoids one or more forbidden graphs as subgraphs. We perform a detailed study of the computational complexity of this problem.
We present a complete picture for the case with a single forbidden connected (induced or non-induced) subgraph. The 2-coloring problem is NP-hard if the forbidden subgraph is a tree with at least two edges, and it is polynomially solvable in all other cases. The 3-coloring problem is NP-hard if the forbidden subgraph is a path, and it is polynomially solvable in all other cases. We also derive results for several forbidden sets of cycles.

Keywords: graph coloring; graph partitioning; forbidden subgraph; planar graph; computational complexity.

1 Introduction

We denote by $G = (V, E)$ a finite undirected and simple graph with $|V| = n$ vertices and $|E| = m$ edges. For any non-empty subset $W \subseteq V$, the subgraph of G induced by W is denoted by $G[W]$. A *clique* of G is a non-empty subset $C \subseteq V$ such that all the vertices of C are mutually adjacent. A non-empty subset $I \subseteq V$ is *independent* if no two of its elements are adjacent. An *r-coloring* of the vertices

* The work of HJB and FVF is sponsored by NWO-grant 047.008.006. Part of the work was done while FVF was visiting the University of Twente, and while he was a visiting postdoc at DIMATIA-ITI (supported by GAČR 201/99/0242 and by the Ministry of Education of the Czech Republic as project LN00A056). FVF acknowledges support by EC contract IST-1999-14186: Project ALCOM-FT (Algorithms and Complexity - Future Technologies). JK acknowledges support by the Czech Ministry of Education as project LN00A056. GJW acknowledges support by the START program Y43-MAT of the Austrian Ministry of Science.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 160–169, 2002.

of G is a partition $V_1, V_2, \ldots, V_r$ of V; the r sets V_j are called the *color classes* of the r-coloring. An r-coloring is *proper* if every color class is an independent set. The *chromatic number* $\chi(G)$ is the minimum integer r for which a proper r-coloring exists.

Evidently, an r-coloring is proper if and only if for every color class V_j, the induced subgraph $G[V_j]$ does not contain a subgraph isomorphic to P_2. This observation leads to a number of interesting generalizations of the classical graph coloring concept. One such generalization was suggested by Harary [15]: Given a graph property π, a positive integer r, and a graph G, a π r-coloring of G is a (not necessarily proper) r-coloring in which every color class has property π. This generalization has been studied for the cases where the graph property π is being acyclic, or planar, or perfect, or a path of length at most k, or a clique of size at most k. We refer the reader to the work of Brown & Corneil [5], Chartrand *et al* [7,8], and Sachs [20] for more information on these variants.

In this paper, we will investigate graph colorings where the property π can be defined via some (maybe infinite) list of forbidden induced subgraphs. This naturally leads to the notion of $\mathcal{F}$-free colorings. Let $\mathcal{F} = \{F_1, F_2, \ldots\}$ be the set of so-called forbidden graphs. Throughout the paper we will assume that the set $\mathcal{F}$ is non-empty, and that all graphs in $\mathcal{F}$ are connected and contain at least one edge. For a graph G, a (not necessarily proper) r-coloring with color classes $V_1, V_2, \ldots, V_r$ is called *weakly $\mathcal{F}$–free*, if for all $1 \leq j \leq r$, the graph $G[V_j]$ does not contain any graph from $\mathcal{F}$ as an *induced* subgraph. Similarly, we say that an r-coloring is *strongly $\mathcal{F}$–free* if $G[V_j]$ does not contain any graph from $\mathcal{F}$ as an (induced or non-induced) subgraph. The smallest possible number of colors in a weakly (respectively, strongly) $\mathcal{F}$-free coloring of a graph G is called the *weakly* (respectively, *strongly*) *$\mathcal{F}$-free chromatic number*; it is denoted by $\chi^W(\mathcal{F}, G)$ (respectively, by $\chi^S(\mathcal{F}, G)$).

In the cases where $\mathcal{F} = \{F\}$ consists of a single graph F, we will sometimes simplify the notation and not write the curly brackets: We will write F-free short for $\{F\}$-free, $\chi^W(F, G)$ short for $\chi^W(\{F\}, G)$, and $\chi^S(F, G)$ short for $\chi^S(\{F\}, G)$. With this notation $\chi(G) = \chi^S(P_2, G) = \chi^W(P_2, G)$ holds for every graph G. Note that

$$\chi^W(\mathcal{F}, G) \;\leq\; \chi^S(\mathcal{F}, G) \;\leq\; \chi(G).$$

It is easy to construct examples where both inequalities are strict. For instance, for $\mathcal{F} = \{P_3\}$ (the path on three vertices) and $G = C_3$ (the cycle on three vertices) we have $\chi(G) = 3$, $\chi^S(P_3, G) = 2$, and $\chi^W(P_3, G) = 1$.

1.1 Previous Results

The literature contains quite a number of papers on weakly and strongly $\mathcal{F}$-free colorings of graphs. The most general result is due to Achlioptas [1]: For any graph F with at least three vertices and for any $r \geq 2$, the problem of deciding whether a given input graph has a weakly F-free r-coloring is NP-hard.

The special case of weakly P_3-free colorings is known as the *subcoloring problem* in the literature. It has been studied by Broere & Mynhardt [4], by Albertson, Jamison, Hedetniemi & Locke [2], and by Fiala, Jansen, Le & Seidel [11].

Proposition 1. [Fiala, Jansen, Le & Seidel [11]]
Weakly P_3-free 2-coloring is NP-hard for triangle-free planar graphs.

A $(1,2)$-*subcoloring* of G is a partition of V_G into two sets S_1 and S_2 such that S_1 induces an independent set and S_2 induces a subgraph consisting of a matching and some (possibly no) isolated vertices. Le and Le [17] proved that recognizing $(1,2)$-subcolorable cubic graphs is NP-hard, even on triangle-free planar graphs.

The case of weakly P_4-free colorings has been investigated by Gimbel & Nešetřil [13] who study the problem of partitioning the vertex set of a graph into induced cographs. Since cographs are exactly the graphs without an induced P_4, the graph parameter studied in [13] equals the weakly P_4-free chromatic number of a graph. In [13] it is proved that the problems of deciding $\chi^W(P_4,G) \leq 2$, $\chi^W(P_4,G) = 3$, $\chi^W(P_4,G) \leq 3$ and $\chi^W(P_4,G) = 4$ all are NP-hard and/or coNP-hard for planar graphs. The work of Hoàng & Le [16] on weakly P_4-free 2-colorings was motivated by the Strong Perfect Graph Conjecture. Among other results, they show that weakly P_4-free 2-coloring is NP-hard for comparability graphs.

A notion that is closely related to strongly F-free r-coloring is the so-called *defective* graph coloring. A defective (k,d)-coloring of a graph is a k-coloring in which each color class induces a subgraph of maximum degree at most d. Defective colorings have been studied for instance by Archdeacon [3], by Cowen, Cowen & Woodall [10], and by Frick & Henning [12]. Cowen, Goddard & Jesurum [9] have shown that the defective $(3,1)$-coloring problem and the defective $(2,d)$-coloring problem for any $d \geq 1$ are NP-hard even for planar graphs. We observe that defective $(2,1)$-coloring is equivalent to strongly P_3-free 2-coloring, and that defective $(3,1)$-coloring is equivalent to strongly P_3-free 3-coloring.

Proposition 2. [Cowen, Goddard & Jesurum [9]]
(i) Strongly P_3-free 2-coloring is NP-hard for planar graphs.
(ii) Strongly P_3-free 3-coloring is NP-hard for planar graphs.

1.2 Our Results

We perform a complexity study of weakly and strongly $\mathcal{F}$-free coloring problems for *planar* graphs. By the Four Color Theorem (4CT), every planar graph G satisfies $\chi(G) \leq 4$. Consequently, every planar graph also satisfies $\chi^W(\mathcal{F},G) \leq 4$ and $\chi^S(\mathcal{F},G) \leq 4$, and we may concentrate on 2-colorings and on 3-colorings. For the case of a single forbidden subgraph, we obtain the following results for 2-colorings:

- If the forbidden (connected) subgraph F is not a tree, then *every* planar graph is strongly and hence also weakly F-free 2-colorable. Hence, the corresponding decision problems are trivially solvable.

- If the forbidden subgraph $F = P_2$, then F-free 2-coloring is equivalent to proper 2-coloring. It is well-known that this problem is polynomially solvable.
- If the forbidden subgraph is a tree T with at least two edges, then both weakly and strongly T-free 2-coloring are NP-hard for planar input graphs. Hence, these problems are intractable.

For 3-colorings with a single forbidden subgraph, we obtain the following results:

- If the forbidden (connected) subgraph F is not a path, then *every* planar graph is strongly and hence also weakly F-free 3-colorable. Hence, the corresponding decision problems are trivially solvable.
- For every path P with at least one edge, both weakly and strongly P-free 3-coloring are NP-hard for planar input graphs. Hence, these problems are intractable.

Moreover, we derive several results for 2-colorings with certain forbidden sets of cycles.

- For the forbidden set $\mathcal{F}_{345} = \{C_3, C_4, C_5\}$, weakly and strongly $\mathcal{F}_{345}$-free 2-coloring both are NP-hard for planar input graphs. Also for the forbidden set $\mathcal{F}_{cycle}$ of all cycles, weakly and strongly $\mathcal{F}_{cycle}$-free 2-coloring both are NP-hard for planar input graphs.
- For the forbidden set $\mathcal{F}_{odd}$ of all cycles of odd lengths, *every* planar graph is strongly and hence also weakly $\mathcal{F}_{odd}$-free 2-colorable.

2 The Machinery for Establishing NP-Hardness

Throughout this section, let $\mathcal{F}$ denote some fixed set of forbidden planar subgraphs. We assume that all graphs in $\mathcal{F}$ are connected and contain at least two edges. We will develop a generic NP-hardness proof for certain types of weakly and strongly $\mathcal{F}$-free 2-coloring problems. The crucial concept is the so-called *equalizer* gadget.

Definition 1. *(Equalizer)*
An (a, b)-equalizer for $\mathcal{F}$ is a planar graph $\mathcal{E}$ with two special vertices a and b that are called the contact points *of the equalizer. The contact points are non-adjacent, and they both lie on the outer face in some fixed planar embedding of $\mathcal{E}$. Moreover, the graph $\mathcal{E}$ has the following properties:*

(i) In every weakly $\mathcal{F}$-free 2-coloring of $\mathcal{E}$, the contact points a and b receive the same color.
(ii) There exists a strongly $\mathcal{F}$-free 2-coloring of $\mathcal{E}$ such that a and b receive the same color, whereas all of their neighbors receive the opposite color. Such a coloring is called a good 2*-coloring of $\mathcal{E}$.*

The following result is our (technical) main theorem. This theorem is going to generate a number of NP-hardness statements in the subsequent sections of the paper. We omit the proof of this theorem in this extended abstract.

Theorem 1. *(Technical main result of the paper)*
Let $\mathcal{F}$ be a set of planar graphs that all are connected and that all contain at least two edges. Assume that

- *$\mathcal{F}$ contains a graph on at least four vertices with a cut vertex, or a 2-connected graph with a planar embedding with at least five vertices on the outer face;*
- *there exists an (a, b)-equalizer for $\mathcal{F}$.*

Then deciding weakly $\mathcal{F}$-free 2-colorability and deciding strongly $\mathcal{F}$-free 2-colorability are NP-hard problems for planar input graphs.

3 Tree-Free 2-Colorings of Planar Graphs

The main result of this section will be an NP-hardness result for weakly and strongly T-free 2-coloring of planar graphs for the case where T is a tree with at least two edges (see Theorem 2). The proof of this result is based on an inductive argument over the number of edges in T. The following two auxiliary Lemmas 1 and 2 will be used to start the induction.

Lemma 1. *Let $K_{1,k}$ be the star with $k \geq 2$ leaves. Then it is NP-hard to decide whether a planar graph has a weakly (strongly) $K_{1,k}$-free 2-coloring.*

Proof. For $k = 2$, the statement for weakly $K_{1,k}$-free 2-colorings follows from Proposition 1, and the statement for strongly $K_{1,k}$-free 2-colorings follows from Proposition 2.(i). For $k \geq 3$, we apply Theorem 1. The first condition in this theorem is fulfilled, since for $k \geq 3$ the star $K_{1,k}$ is a graph on at least four vertices with a cut vertex. For the second condition, we construct an (a, b)-equalizer.

The equalizer is the complete bipartite graph $K_{2,2k-1}$ with bipartitions I, $|I| = 2k - 1$, and $\{a, b\}$. This graph satisfies Definition 1.(i): In any 2-coloring, at least k of the vertices in I receive the same color, say color 0. If a and b are colored differently, then one of them is colored 0. This yields an induced monochromatic $K_{1,k}$. A good coloring as required in Definition 1.(ii) results from coloring a and b by the same color, and all vertices in I by the opposite color.

For $1 \leq k \leq m$, a *double-star* $X_{k,m}$ is the tree of the following form: $X_{k,m}$ has $k + m + 2$ vertices. There are two adjacent central vertices y_1 and y_2. Vertex y_1 is adjacent to k leaves, and y_2 is adjacent to m leaves. In other words, the double-star $X_{k,m}$ results from adding an edge between the two central vertices of the stars $K_{1,k}$ and $K_{1,m}$. Note that $X_{1,1}$ is isomorphic to the path P_4.

Lemma 2. *Let $X_{k,m}$ be a double star with $1 \leq k \leq m$. Then it is NP-hard to decide whether a planar graph has a weakly (strongly) $X_{k,m}$-free 2-coloring.*

Proof. We apply Theorem 1. The first condition in this theorem is fulfilled, since $X_{k,m}$ is a graph on at least four vertices with a cut vertex. For the second condition, we will construct an (a, b)-equalizer.

The (a, b)-equalizer $\mathcal{E} = (V', E')$ consists of $2m + k - 1$ independent copies (V^i, E^i) of the double star $X_{k,m}$ where $1 \leq i \leq 2m + k - 1$. Moreover, there are five special vertices a, b, v_1, v_2, and v_3. We define

$$V' = \{v_1, v_2, v_3, a, b\} \cup \bigcup_{1 \leq i \leq 2m+k-1} V^i \quad \text{and}$$

$$\begin{aligned} E' = {} & \{v_i v_j : 1 \leq i, j \leq 3\} \cup av_3 \cup bv_3 \cup \\ & \bigcup_{1 \leq i \leq 2m+k-1} E^i \cup \\ & \bigcup_{1 \leq i \leq m} \{v_1 v \colon v \in V^i\} \cup \\ & \bigcup_{m+1 \leq i \leq 2m} \{v_2 v \colon v \in V^i\} \cup \\ & \bigcup_{2m+1 \leq i \leq 2m+k-1} \{v_3 v \colon v \in V^i\}. \end{aligned}$$

We claim that every 2-coloring of $\mathcal{E}$ with a and b colored in different colors contains a monochromatic induced copy of $X_{k,m}$: Consider some weakly $X_{k,m}$-free coloring of $\mathcal{E}$. Then each copy (V^i, E^i) of $X_{k,m}$ must have at least one vertex that is colored 0 and at least one vertex that is colored 1. If v_1 and v_2 had the same color, then together with appropriate vertices in V^i, $1 \leq i \leq 2m$, they would form a monochromatic copy of $X_{k,m}$. Hence, we may assume by symmetry that v_1 is colored 1, that v_2 is colored 0, and that v_3 is colored 0. Suppose for the sake of contradiction that a and b are colored differently. Then one of them would be colored 0, and there would be a monochromatic copy of $X_{k,m}$ with center vertices v_3 and v_2. Thus $\mathcal{E}$ satisfies property (i) in Definition 1.

To show that also property (ii) in Definition 1 is satisfied, we construct a good 2-coloring: The vertices a, b, v_1 are colored 0, and v_2 and v_3 are colored 1. In every set V^i with $1 \leq i \leq m$, one vertex is colored 0 and all other vertices are colored 1. In every set V^i with $m + 1 \leq i \leq 2m + k - 1$, one vertex is colored 1 and all other vertices are colored 0.

Now we are ready to prove the main result of this section.

Theorem 2. *Let T be a tree with at least two edges. Then it is NP-hard to decide whether a planar input graph G has a weakly (strongly) T-free 2-coloring.*

Proof. By induction on the number ℓ of edges in T. If T has $\ell = 2$ edges, then $T = K_{1,2}$, and NP-hardness follows by Lemma 1. If T has $\ell \geq 3$ edges, then we consider the so-called *shaved* tree T^* of T that results from T by removing all the leaves. If the shaved tree T^* is a single vertex, then T is a star, and

NP-hardness follows by Lemma 1. If the shaved tree T^* is a single edge, then T is a double star, and NP-hardness follows by Lemma 2.

Hence, it remains to settle the case where the shaved tree T^* contains at least two edges. In this case we know from the induction hypothesis that weakly (strongly) T^*-free 2-coloring is NP-hard. Consider an arbitrary planar input graph G^* for weakly (strongly) T^*-free 2-coloring. To complete the NP-hardness proof, we will construct in polynomial time a planar graph G that has a weakly (strongly) T-free 2-coloring if and only if G^* has a weakly (strongly) T^*-free 2-coloring: Let Δ be the maximum vertex degree of T. For every vertex v in G^*, we create Δ independent copies $T_1(v), \ldots, T_\Delta(v)$ of T, and we connect v to all vertices of all these copies.

Assume first that G^* is weakly (strongly) T^*-free 2-colorable. We extend this coloring to a weakly (strongly) T-free coloring of G by taking a proper 2-coloring of every subgraph $T_i(v)$ in G. It can be verified that this extended coloring for G does not contain any monochromatic copy of T.

Now assume that G is weakly (strongly) T-free 2-colorable, and let c be such a 2-coloring. Every subgraph $T_i(v)$ in G must meet both colors. This implies that every vertex v in the subgraph G^* of G has at least Δ neighbors of color 0 and at least Δ neighbors of color 1 in the subgraphs $T_i(v)$. This implies that the restriction of the coloring c to the subgraph G^* is a weakly (strongly) T^*-free 2-coloring. This concludes the proof of the theorem.

4 Cycle-Free 2-Colorings of Planar Graphs

In the previous sections we have shown that for every tree F with $|E(F)| \geq 2$, the problem of deciding whether a given planar graph has a weakly (strongly) F-free 2-coloring is NP-hard. If the forbidden tree F is a P_2, then F-free 2-coloring is equivalent to proper 2-coloring, and hence the corresponding problem is polynomially solvable.

We now turn to the case in which F is not a tree and hence contains a cycle (we assume F is connected).

If F contains an odd cycle, then the Four Color Theorem (4CT) shows that any planar graph G has a weakly (strongly) F-free 2-coloring: a proper 4-coloring of G partitions V_G into two sets S_1 and S_2 each inducing a bipartite graph. Coloring all the vertices of S_i by color i yields a weakly (strongly) F-free 2-coloring of G. If we extend the set of forbidden cycles to all cycles of odd length, denoted by $\mathcal{F}_{odd}$, then the converse is also true: In any $\mathcal{F}_{odd}$-free 2-coloring of G both monochromatic subgraphs of G are bipartite, yielding a 4-coloring of G. To summarize we obtain the following.

Lemma 3. *The statement "*$\chi^S(\mathcal{F}_{odd}, G) \leq 2$ for every planar graph G*" is equivalent to the 4CT.*

In case F is just the triangle C_3, one can avoid using the heavy 4CT machinery to prove that for every planar graph G $\chi^S(C_3, G) \leq 2$ by applying a result due to Burstein [6]. We omit the details.

If F contains no triangles, a result of Thomassen [21] can be applied. He proved that the vertex set of any planar graph can be partitioned into two sets each of which induces a subgraph with no cycles of length exceeding 3. Hence every planar graph is weakly (strongly) $\mathcal{F}_{\geq 4}$-free 2-colorable, where $\mathcal{F}_{\geq 4}$ denotes the set of all cycles of length exceeding 3. The following theorem summarizes the above observations.

Theorem 3. *If the forbidden connected subgraph F is not a tree, then every planar graph G is strongly and hence also weakly F-free 2-colorable.*

The picture changes if one forbids several cycles.

Theorem 4. *Let $\mathcal{F}_{345} = \{C_3, C_4, C_5\}$ be the set of cycles of lengths three, four, and five. Then the problem of deciding whether a given planar graph has a weakly (strongly) $\mathcal{F}_{345}$-free 2-coloring is NP-hard.*

We omit the proof of the theorem in the extended abstract.

Recently Kaiser & Škrekovski announce the proof of $\chi^W(\mathcal{F}, G) \leq 2$ for $\mathcal{F} = \{C_3, C_4\}$ and every planar graph G.

5 3-Colorings of Planar Graphs

A *linear forest* is a disjoint union of paths and isolated vertices. The following result was proved independently in [14] and [19]:

Proposition 3. [Goddard [14] and Poh [19]]
Every planar graph G has a partition of its vertex set into three subsets such that every subset induces a linear forest.

This result immediately implies that if a connected graph F is not a path, then $\chi^W(F, G) \leq 3$ and $\chi^S(F, G) \leq 3$ hold for *all* planar graphs G. Hence, these coloring problems are trivially solvable in polynomial time.

We now turn to the remaining cases of F-free 3-coloring for planar graphs where the forbidden graph F is a path. We start with a technical lemma that will yield a gadget for the NP-hardness argument.

Lemma 4. *For every $k \geq 2$, there exists an outer-planar graph Y_k that satisfies the following properties.*

(i) Y_k is not weakly P_k-free 2-colorable.
(ii) There exists a strongly P_k-free 3-coloring of Y_k, in which one of the colors is only used on an independent set of vertices.

We omit the proof of the lemma here.

Theorem 5. *For any path P_k with $k \geq 2$, it is NP-hard to decide whether a planar input graph G has a weakly (strongly) P_k-free 3-coloring.*

Proof. We will use induction on k. The basic cases are $k = 2$ and $k = 3$. For $k = 2$, weakly and strongly P_2-free 3-coloring is equivalent to proper 3-coloring which is well-known to be NP-hard for planar graphs.

Next, consider the case $k = 3$. Proposition 2.(ii) yields NP-hardness of strongly P_3-free 3-coloring for planar graphs. For weakly P_3-free 3-coloring, we sketch a reduction from proper 3-coloring of planar graphs. As a gadget, we use the outer-planar graph Z depicted in Figure 1. The crucial property of Z is that it does not allow a weakly P_3-free 2-coloring, as is easily checked. Now consider an arbitrary planar graph G. From G we construct the planar graph G': For every vertex v in G, create a copy $Z(v)$ of Z, and add all possible edges between v and $Z(v)$. It can be verified that $\chi(G) \leq 3$ if and only if $\chi^w(P_3, G') \leq 3$.

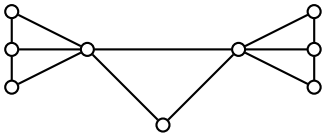


Fig. 1. The graph Z in the proof of Theorem 5.

For $k \geq 4$, we will give a reduction from weakly (strongly) P_{k-2}-free 3-coloring to weakly (strongly) P_k-free 3-coloring. Consider an arbitrary planar graph G, and construct the following planar graph G': For every vertex v in G, create a copy $Y_k(v)$ of the graph Y_k from Lemma 4, and add all possible edges between v and $Y_k(v)$. Since Y_k is outer-planar, the new graph G' is planar. If G has a weakly (strongly) P_{k-2}-free 3-coloring, then this can be extended to a weakly (strongly) P_k-free 3-coloring of G' by coloring the subgraphs $Y_k(v)$ according to Lemma 4.(ii). And if G' has a weakly (strongly) P_k-free 3-coloring, then by Lemma 4.(i) this induces a weakly (strongly) P_{k-2}-free 3-coloring for G.

Acknowledgments. We are grateful to Oleg Borodin, Alesha Glebov, Sasha Kostochka, and Carsten Thomassen for fruitful discussions on the topic of this paper.

References

1. D. Achlioptas, *The complexity of G-free colorability*, Discrete Math., 165/166 (1997), pp. 21–30. Graphs and combinatorics (Marseille, 1995).
2. M. O. Albertson, R. E. Jamison, S. T. Hedetniemi, and S. C. Locke, *The subchromatic number of a graph*, Discrete Math., 74 (1989), pp. 33–49.
3. D. Archdeacon, *A note on defective colorings of graphs in surfaces*, J. Graph Theory, 11 (1987), pp. 517–519.

4. I. Broere and C. M. Mynhardt, *Generalized colorings of outer-planar and planar graphs*, in Graph theory with applications to algorithms and computer science (Kalamazoo, Mich., 1984), Wiley, New York, 1985, pp. 151–161.
5. J. I. Brown and D. G. Corneil, *On uniquely $-G$ k-colorable graphs*, Quaestiones Math., 15 (1992), pp. 477–487.
6. M. I. Burstein, *The bi-colorability of planar hypergraphs*, Sakharth. SSR Mecn. Akad. Moambe, 78 (1975), pp. 293–296.
7. G. Chartrand, D. P. Geller, and S. Hedetniemi, *A generalization of the chromatic number*, Proc. Cambridge Philos. Soc., 64 (1968), pp. 265–271.
8. G. Chartrand, H. V. Kronk, and C. E. Wall, *The point-arboricity of a graph*, Israel J. Math., 6 (1968), pp. 169–175.
9. L. J. Cowen, W. Goddard, and C. E. Jesurum, *Defective coloring revisited*, J. Graph Theory, 24 (1997), pp. 205–219.
10. L. J. Cowen, R. H. Cowen, and D. R. Woodall, *Defective colorings of graphs in surfaces: partitions into subgraphs of bounded valency*, J. Graph Theory, 10 (1986), pp. 187–195.
11. J. Fiala, K. Jansen, V. B. Le, and E. Seidel, *Graph subcoloring: Complexity and algorithms*, in Graph-theoretic concepts in computer science, WG 2001, Springer, Berlin, 2001, pp. 154–165.
12. M. Frick and M. A. Henning, *Extremal results on defective colorings of graphs*, Discrete Math., 126 (1994), pp. 151–158.
13. J. Gimbel and J. Nešetřil, *Partitions of graphs into cographs*, Technical Report 2000-470, KAM-DIMATIA, Charles University, Czech Republic, 2000.
14. W. Goddard, *Acyclic colorings of planar graphs*, Discrete Math., 91 (1991), pp. 91–94.
15. F. Harary, *Conditional colorability in graphs*, in Graphs and applications (Boulder, Colorado, 1982), Wiley, New York, 1985, pp. 127–136.
16. C. T. Hoàng and V. B. Le, *P_4-free colorings and P_4-bipartite graphs*, Discrete Math. Theor. Comput. Sci., 4 (2001), pp. 109–122 (electronic).
17. H.-O. Le and V. B. Le, *The NP-completeness of $(1, r)$-subcoloring of cubic graphs*, Information Proc. Letters, 81 (2002), pp. 157–162.
18. D. Lichtenstein, *Planar formulae and their uses*, SIAM J. Comput., 11 (1982), pp. 329–343.
19. K. S. Poh, *On the linear vertex-arboricity of a planar graph*, J. Graph Theory, 14 (1990), pp. 73–75.
20. H. Sachs, *Finite graphs (Investigations and generalizations concerning the construction of finite graphs having given chromatic number and no triangles)*., in Recent Progress in Combinatorics (Proc. Third Waterloo Conf. on Combinatorics, 1968), Academic Press, New York, 1969, pp. 175–184.
21. C. Thomassen, *Decomposing a planar graph into degenerate graphs*, J. Combin. Theory Ser. B 65 (1995), pp. 305–314.

Approximation Hardness of the Steiner Tree Problem on Graphs

Miroslav Chlebík[1] and Janka Chlebíková[2⋆]

[1] Max Planck Institute for Mathematics in the Sciences
Inselstraße 22-26, D-04103 Leipzig, Germany
[2] Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Olshausenstraße 40, D-24098 Kiel, Germany
jch@informatik.uni-kiel.de

Abstract. Steiner tree problem in weighted graphs seeks a minimum weight subtree containing a given subset of the vertices (terminals). We show that it is NP-hard to approximate the Steiner tree problem within 96/95. Our inapproximability results are stated in parametric way and can be further improved just providing gadgets and/or expanders with better parameters. The reduction is from Håstad's inapproximability result for maximum satisfiability of linear equations modulo 2 with three unknowns per equation. This was first used for the Steiner tree problem by Thimm whose approach was the main starting point for our results.

1 Introduction

Given a graph $G = (V, E)$, a weight function $w\colon E \to \mathbb{R}^+$ on the edges, and a set of required vertices $T \subseteq V$, the *terminals*. A *Steiner tree* is a subtree of G that spans all vertices in T. (It might use vertices in $V \setminus T$ as well.)

The Steiner Tree Problem (STP) is to find a Steiner tree of minimum weight. Denote OPT the optimal value of the Steiner tree OPT := $\min\{w(\mathcal{T}) : \mathcal{T}$ is a Steiner tree$\}$.

An instance of the Steiner tree problem is called *quasi-bipartite* if there is no edge in the set $V \setminus T$, and *uniformly quasi-bipartite* if it is quasi-bipartite and edges incident to the same non-terminal vertex have the same weight.

Steiner trees are important in various applications, for example VLSI routing, wirelength estimation and network routing.

The Steiner Tree Problem is among the 21 basic problems for which Karp has shown NP-hardness in his paper [6].

As we cannot expect to find polynomial time algorithms for solving it exactly (unless P = NP), the search is for effective approximation algorithms. During the last years many approximation algorithms for the Steiner Tree Problem were designed, see [5] for survey. The currently best approximation algorithm

⋆ The second author has been supported by the EU-Project ARACNE, Approximation and Randomized Algorithms in Communication Networks, HPRN-CT-199-00112.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 170–179, 2002.

of Robins and Zelikovsky ([8]) has a performation ratio of 1.550, and 1.279 for quasi-bipartite instances. In the case of uniformly quasi-bipartite instances, the best known algorithm has a performation ratio 1.217 [5].

It is a natural question how small the performation ratio of a polynomial time algorithm can get. Unless P = NP, it cannot get arbitrarily close to 1. This follows from PCP-Theorem [1] and from the fact that the problem is APX-complete [2].

Until now, the best known lower bound is inapproximability within 1.00617, due to Thimm ([9]). (In fact, Thimm's paper claims the lower bound of 1.0074, but there is a small error in the paper and only a slightly worse lower bound can be shown along the lines of the proof. Moreover, Thimm's paper uses the more restrictive assumption co-RP $\neq$ NP.)

Main Result

The main result of this article improves the lower bounds on approximability of the STP and reduces the gap between known approximability and inapproximability results.

Main Theorem. *It is NP-hard to approximate the* Steiner Tree Problem *within ratio* 1.01063 ($> \frac{96}{95}$). *For the special case of (uniformly) quasi-bipartite instances approximation within ratio* 1.00791 ($> \frac{128}{127}$) *is NP-hard.*

Our reduction is from Håstad's hard-gap result for maximum satisfiability of linear equations modulo 2 with three unknowns per equation, Max-E3-Lin-2.

Definition 1. Max-E3-Lin-2 *is the following optimization problem: Given a system of linear equations over* $\mathbb{Z}_2$*, with exactly* 3 *variables in each equation. The goal is to find an assignment to the variables that satisfies as many equations as possible.*

To suit our purposes we state Håstad's important result in the following way (see also [7] for application of that result in a similar context).

Theorem 1. ([4]) *For every* $\varepsilon \in \left(0, \frac{1}{4}\right)$ *and every fixed sufficiently large integer* $k \geq k(\varepsilon)$*, the following partial decision subproblem of* Max-E3-Lin-2 *is NP-hard:*

$$P(\varepsilon, k) \quad \begin{cases} \textit{Given an instance Max-E3-Lin-2 consisting of } n \textit{ equations and} \\ \textit{with exactly } 2k \textit{ occurrences of each variable, to decide if at least} \\ (1-\varepsilon)n \textit{ or at most } \left(\frac{1}{2}+\varepsilon\right)n \textit{ equations are satisfied by the optimal} \\ \textit{assignment.} \end{cases}$$

The same NP-hardness result holds on instances where all equations are of the form $x+y+z=0$ (respectively, all equations are of the form $x+y+z=1$), where literals x, y, z are variables or their negations, and each variable appears exactly k times negated and k times unnegated. This subproblem of the problem $P(\varepsilon, k)$ will be referred to as $P_0(\varepsilon, k)$ (respectively $P_1(\varepsilon, k)$) in what follows.

2 NP-Hard-Gap Preserving Reduction

We start with a set L of n linear equations over $\mathbb{Z}_2$, all of the form $x+y+z=0$ (respectively, all of the form $x+y+z=1$), where literals x, y, z are variables from the set $\mathcal{V}$ or their negations, and each variable $v \in \mathcal{V}$ appears in L exactly k times negated as $\overline{v}$ and k times unnegated.

For an assignment $\psi \in \{0,1\}^{\mathcal{V}}$ to variables let $S(\psi)$ be the number of equations of L satisfied by ψ. We will reduce the problem of maximizing $S(\psi)$ over all assignments to the instance of the Steiner Tree Problem. To get an approximation preserving reduction we will use equation gadgets and couple them properly using a graph with good vertex-expansion property.

The Equation Gadget

Now we introduce the notion of (α,β,γ)-gadget for the reduction from the equation system of the form $x+y+z=0$ (respectively, from the system of the form $x+y+z=1$). This will be an instance $G=(V,E)$, $w\colon E \to \mathbb{R}^+$, $T \subseteq V$ of the Steiner Tree Problem with the following properties:

1. One of (possibly more) terminal vertices is distinguished and denoted by O.
2. Three of (possibly more) non-terminal vertices are distinguished and denoted by x, y and z.
3. For any $u \in \{x,y,z\}$ there is a path from u to O of weight at most 1.
4. For any subset R of $\{x,y,z\}$ consider the instance of the STP with altered terminal set $T_R := T \cup R$. The weight of the corresponding minimum Steiner tree is denoted by s_R and is required to depend on the cardinality of the set R only in the following way,

$$s_R = \alpha + |R|\beta + (|R| \bmod 2)\gamma.$$

(Respectively, if our system L is of the form $x+y+z=1$, we require $s_R = \alpha + |R|\beta + (1-|R| \bmod 2)\gamma$.)

An (α,β,γ)-gadget with no edges between non-terminal vertices is called quasi-bipartite (α,β,γ)-gadget. A quasi-bipartite (α,β,γ)-gadget such that edges incident to the same non-terminal have the same weight and for vertices x, y, z the incident edges have the weight 1 is called uniformly quasi-bipartite (α,β,γ)-gadget.

In our reduction we will use one copy of a fixed (α,β,γ)-gadget per each equation of L. For each variable v, k negated and k unnegated occurrences of v will be further coupled using a particular k by k regular bipartite multigraph, which is a good expander.

The condition 3 above is just a proper normalization.

The condition 4 on $s_k := s_R$, $k = |R| \in \{0,1,2,3\}$, has the following interpretation in our construction: α is a *basic cost* per equation, β is an *extra payment* for connecting some of $\{x,y,z\}$ to the Steiner tree, and γ is a *penalty* for the failure in the parity check of the number of vertices of $\{x,y,z\}$ adjacent to the Steiner tree.

Example 1. For any $\gamma \in \left(0, \frac{1}{4}\right\rangle$ there is a $(0, 1-\gamma, \gamma)$-gadget (for the system L of the form $x+y+z=0$), depicted on Fig. 1. The vertex O is the only terminal. Clearly $s_0 = 0$, $s_1 = 1$, $s_2 = 2-2\gamma$, and $s_3 = 3-2\gamma$.

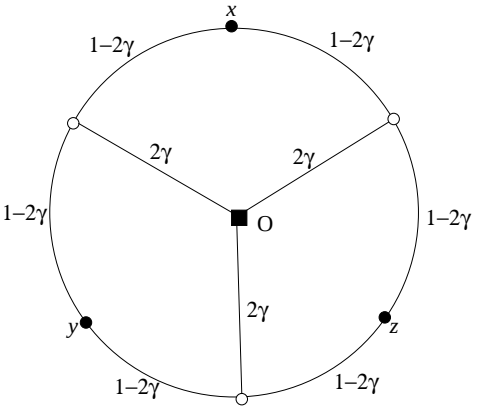


Fig. 1.

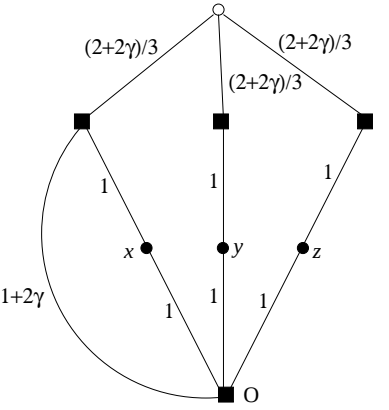


Fig. 2.

Example 2. For any $\gamma \in \langle 0, \frac{1}{2}\rangle$ there is a uniform quasi-bipartite $(3+3\gamma, 1-\gamma, \gamma)$-gadget (for the system L of the form $x+y+z=1$), depicted on Fig. 2.

There are 4 terminals in this gadget, all drawn as boxes. One can easily check that $s_0 = 3+4\gamma$, $s_1 = 4+2\gamma$, $s_2 = 5+2\gamma$, and $s_3 = 6$. This is essentially the gadget used by Thimm ([9]) in his reduction translated to our language.

Expanders

An expander with parameters (c, τ, d) (shortly, (c, τ, d)-expander) is a d-regular bipartite multigraph with balanced k by k bipartition (V_1, V_2), such that

$$\text{if } U \subseteq V_1 \text{ or } U \subseteq V_2, \text{ and } |U| \le \tau k, \text{ then } |\Gamma(U)| \ge c|U|.$$

Here $\Gamma(U)$ stands for the set of neighbors of U, $\Gamma(U) := \{y : y \text{ is a vertex adjacent to some } x \in U\}$.

It is known that for any sufficiently large k, (c, τ, d)-expander with k by k bipartition exists, provided that $0 < \tau < \frac{1}{c} < 1$ and $H_d(c, \tau) > 0$, where

$$H_d(c, \tau) := (d-1)F(\tau) - dc\tau F\Big(\frac{1}{c}\Big) - F(c\tau),$$

with $F(x) = -x \log x - (1-x)\log(1-x)$ being the entropy function. In fact, under the above condition, almost every random d-regular balanced bipartite multigraph is (c, τ, d)-expander, see Theorem 6.6 in [3].

Definition 2. *We say that d-regular bipartite graph with k by k bipartition (V_1, V_2) is a* c-good *expander provided the following implication holds:*

$$\text{if } U \subseteq V_1 \text{ or } U \subseteq V_2, \text{ then } |\Gamma(U)| \geq \min\{c|U|, k+1-|U|\}.$$

The condition of being c-good expander for a balanced d-regular bipartite graph is just a bit stronger than the one of being $\big(c, \frac{1}{c+1}, d\big)$-expander. In particular, it can be easily seen that (c, τ, d)-expander with k by k bipartition is c-good, provided that $\tau > \frac{1}{c+1}$ and $k \geq \frac{c}{(c+1)\tau - 1}$.

Consequently, for any sufficiently large k, a d-regular c-good expander with k by k bipartition exists, provided that $c > 1$ satisfies

$$H_d\Big(c, \frac{1}{c+1}\Big) > 0. \tag{1}$$

In fact, by continuity it follows that $H_d(c, \tau) > 0$ also for some $\tau \in \big(\frac{1}{c+1}, \frac{1}{c}\big)$, and we can use the existence result for (c, τ, d)-expanders cited above.

For any integer $d \geq 3$ we introduce the constant $c(d)$ defined in the following way:

$$c(d) = \sup\{c : \text{there are infinitely many } d\text{-regular } c\text{-good expanders}\}. \tag{2}$$

Denote by $x(d)$ the unique $x \in (1, \infty)$ for which $H_d\big(x, \frac{1}{x+1}\big) = 0$. It can be easily numerically approximated, as $(x+1)H_d\big(x, \frac{1}{x+1}\big)$ can be simplified to

$$(d-2)(x+1)\log(x+1) - (2d-2)x\log x + d(x-1)\log(x-1).$$

Hence (1) holds for any c in $(1, x(d))$ and, consequently, $c(d) \geq x(d)$ for any integer $d \geq 3$. In particular, $c(6) > 1.76222$ and $c(7) > 1.94606$.

Now we are ready to describe our reduction of instances like L to the instances of the Steiner Tree Problem. For this purpose we will use one fixed (α, β, γ)-gadget, and one fixed k by k bipartite d-regular multigraph H which is supposed to be $\frac{\beta+\gamma}{\beta-\gamma}$-good.

Construction

Take n pairwise disjoint copies of that (α, β, γ)-gadget, one for each equation of the system L, and identify their vertices labeled by O. The x, y, z vertices in each *equation gadget* correspond to occurrences of literals in that equation and we re-label them by those literals. By assumption, each variable from $\mathcal{V}$ appears exactly k times negated and k times unnegated as a label. We couple negated and unnegated occurrences of each variable using our fixed bipartite d-regular multigraph H with bipartition (V_1, V_2), $V_1 = \{a_1, a_2, \ldots, a_k\}$, $V_2 = \{b_1, b_2, \ldots, b_k\}$ in the following way:

Assume that equations (and their equation gadgets) are numbered by 1, 2, ..., n. Given literal x, i.e. $x = v$ or $x = \overline{v}$ for some $v \in \mathcal{V}$, let $m_1(x) < m_2(x) < \cdots < m_k(x)$ be the numbers of equations in which that literal occurs.

Consider one variable of $\mathcal{V}$, say v. For each a_ib_j edge e of H $(1 \le i, j \le k)$ we add a new *coupling* terminal vertex $t(v, e)$. Now connect it with the v-vertex in the $m_i(v)$-th equation gadget and with the $\overline{v}$-vertex in the $m_j(\overline{v})$-th equation gadget, by edges of weight 1.

Making the above coupling for all variables from $\mathcal{V}$ one after another, we get an instance of the Steiner tree problem, that corresponds to the system L. Consider any Steiner tree $\mathcal{T}$ for this instance, i.e. a tree spanning all terminals.

In the following claim we prove that in the Steiner trees with the optimal value OPT each coupling terminal vertex $t(v, e)$ is a leaf of $\mathcal{T}$. We call *simple* a Steiner tree $\mathcal{T}$ with mentioned property that each coupling terminal vertex $t(v, e)$ is a leaf of $\mathcal{T}$.

Claim. $\mathrm{OPT} = \min\{w(\mathcal{T}) : \mathcal{T}$ is a simple Steiner tree$\}$.

Proof. To show that, one can transform any given Steiner tree $\mathcal{T}$ with nonempty 'bad' set $\mathrm{BAD}(\mathcal{T}) := \{$coupling terminals that are not leaves of $\mathcal{T}\}$ to another Steiner tree $\mathcal{T}'$ with $|\mathrm{BAD}(\mathcal{T}')| < |\mathrm{BAD}(\mathcal{T})|$ and $w(\mathcal{T}') \le w(\mathcal{T})$. Fix $\mathcal{T}$ with nonempty bad set and choose $t = t(v, e) \in \mathrm{BAD}(\mathcal{T})$. Deleting one of edges incident to t decreases both $|\mathrm{BAD}(\mathcal{T})|$ and $w(\mathcal{T})$ by 1. But we have two components now, with one of vertices labeled by v or $\overline{v}$ in the distinct component than the vertex O belongs. Connect this vertex with O in its equation gadget in the cheapest possible way, to obtain the Steiner tree $\mathcal{T}'$.

By property 3 of (α, β, γ)-gadget it increases the weight by at most 1, hence $w(\mathcal{T}') \le w(\mathcal{T})$. □

Definition 3. *We say that a simple Steiner tree $\mathcal{T}$ is* well-behaved *if it is locally minimal in the following sense:*
Consider any equation of L, say i-th, $i \in \{1, 2, \ldots, n\}$. Let x, y, z be its literals, $T := T^i$ be the set of terminal vertices of its equation gadget, and $R := R^i$ be the set of vertices of this gadget labeled by x, y, or z, that belong to $\mathcal{T}$. The subgraph $\mathcal{T}^i$ of $\mathcal{T}$ induced by this equation gadget is supposed to be the local minimal Steiner tree (in this gadget) for the altered terminal set $T_R := T \cup R$.

Claim. $\mathrm{OPT} = \min\{w(\mathcal{T}) : \mathcal{T}$ is a well-behaved Steiner tree$\}$.

Proof. Clearly, any simple Steiner tree $\mathcal{T}$ with $w(\mathcal{T}) = \mathrm{OPT}$ has to be well-behaved, because otherwise one could create, by local change in some of its gadget, a Steiner tree with less weight. In particular, $\mathrm{OPT} = \min\{w(\mathcal{T}) : \mathcal{T}$ is a well-behaved Steiner tree$\}$. □

By property 4 of (α, β, γ)-gadget, the weight of subtree $\mathcal{T}^i$ is $\alpha + |R|\beta + (|R| \bmod 2)\gamma$ (respectively, $\alpha + |R|\beta + (1 - |R| \bmod 2)\gamma$). Hence, the weight of any well-behaved Steiner tree $\mathcal{T}$ can be expressed in the following way: denote by N the number of vertices corresponding to literals that belong to $\mathcal{T}$, and by M the number of equations for which $R := R^i$ above fails the parity check, i.e. $|R^i|$ is odd (respectively, $|R^i|$ is even). Then

$$w(\mathcal{T}) = \alpha n + \frac{3}{2}nd + N\beta + M\gamma. \tag{3}$$

Here $\frac{3}{2}nd$ edges of weight 1 connect all $\frac{3}{2}nd$ coupling terminals as leaves of the tree $\mathcal{T}$. Clearly, $N \geq \frac{3}{2}n$, as at least one from each coupled pair of vertices correspond to variables has to belong to $\mathcal{T}$, to connect the corresponding coupling terminal to the tree $\mathcal{T}$.

Suppose we are given an assignment $\psi \in \{0,1\}^{\mathcal{V}}$ to variables and let $S(\psi)$ be the number of equations satisfied by ψ. For i-th equation of L $(i = 1, 2, \ldots, n)$ let $R := R^i_\psi$ denote the set of vertices in its equation gadget labeled by literals with value 1 by the assignment ψ, and let $T := T^i$ denote the terminals of this equation gadget. Take one (of possibly more) local minimum Steiner tree in this gadget with altered terminal set $T_R := T \cup R$ and connect each vertex of R to all d coupling terminals adjacent to it. Such kind of well-behaved Steiner tree (denoted by $\mathcal{T}_\psi$), which follows from some assignment ψ will be called *standard* Steiner tree.

The weight of standard Steiner tree $\mathcal{T}_\psi$ can be expressed using (3), where we have now $N = \frac{3}{2}n$ (exactly half of vertices for variables correspond to literals assigned 1), and $M = n - S(\psi)$. Hence

$$w(\mathcal{T}_\psi) = \alpha n + \frac{3}{2}nd + \frac{3}{2}n\beta + (n - S(\psi))\gamma. \tag{4}$$

The challenge is to prove Lemma 1 below that OPT is achieved on a standard Steiner tree for some assignment ψ. If this is the case, from (4) it can be easily seen that hard-gap result of Håstad for the problem $\max S(\psi)$ implies the corresponding hard-gap and inapproximability results for the Steiner Tree Problem.

Lemma 1. *If (α, β, γ)-gadget has parameters $\beta > \gamma \geq 0$, and an expander graph used for the coupling is $\frac{\beta+\gamma}{\beta-\gamma}$-good, then*

$$OPT = \min\{w(\mathcal{T}) : \mathcal{T} \text{ is a standard Steiner tree}\}.$$

Proof. We already know that there exists a well-behaved Steiner tree $\mathcal{T}$ such that $w(\mathcal{T}) = \mathrm{OPT}$. Thus it is sufficient to show that $\mathcal{T}$ can be transformed into a standard Steiner tree $\mathcal{T}^*$ without increasing the weight. In the following we describe such construction of $\mathcal{T}^*$ from $\mathcal{T}$ in $|\mathcal{V}|$ steps. Consider one variable, $v \in \mathcal{V}$. Let A_1 be the set of vertices labeled by v, and A_2 be the set of vertices labeled by $\overline{v}$. Clearly $|A_1| = |A_2| = k$. Denote by C_i $(i = 1, 2)$ the set of vertices in A_i that are vertices of the tree $\mathcal{T}$, and put $U_i = A_i \setminus C_i$. We will assume that $|U_1| \leq |U_2|$, otherwise we change the role of A_1 and A_2 in what follows.

Let $\Gamma(U)$, for a set $U \subseteq A_1$, be the set of vertices in A_2 which are coupled with a vertex in U. Clearly $U_2 \cap \Gamma(U_1) = \emptyset$, because otherwise some coupling terminal is not connected to $\mathcal{T}$. Hence $\Gamma(U_1) \subseteq C_2$.

As our expander is $\frac{\beta+\gamma}{\beta-\gamma}$-good, it implies that either $|\Gamma(U_1)| \geq k + 1 - |U_1|$, or $|\Gamma(U_1)| \geq \frac{\beta+\gamma}{\beta-\gamma}|U_1|$.

We see that the first condition is not satisfied, as

$$k - |U_1| \geq k - |U_2| = |C_2| \geq |\Gamma(U_1)|.$$

Thus we can apply the second one to get

$$|C_2| \geq |\Gamma(U_1)| \geq \frac{\beta+\gamma}{\beta-\gamma}|U_1|. \tag{5}$$

Now we modify $\mathcal{T}$ to the new well-behaved ST $\mathcal{T}_{\text{new}}$ as follows: all vertices in A_1 and none in A_2 are in $\mathcal{T}_{\text{new}}$, and for any distinguished vertex u which is labeled by literal distinct from v and $\overline{v}$,

$$u \in \mathcal{T}_{\text{new}} \Leftrightarrow u \in \mathcal{T}.$$

We also connect the coupling terminals accordingly.

Applying formula (3) for well-behaved Steiner trees we obtain

$$w(\mathcal{T}) - w(\mathcal{T}_{\text{new}}) = (N - N_{\text{new}})\beta + (M - M_{\text{new}})\gamma.$$

Clearly, $N - N_{\text{new}} = |C_2| - |U_1|$ and $M_{\text{new}} \leq M + |C_2| + |U_1|$, hence

$$w(\mathcal{T}) - w(\mathcal{T}_{\text{new}}) \geq (|C_2| - |U_1|)\beta - (|C_2| + |U_1|)\gamma = |C_2|(\beta-\gamma) - |U_1|(\beta+\gamma),$$

which is nonnegative, by (5). Thus $w(\mathcal{T}_{\text{new}}) \leq w(\mathcal{T})$.

Now we apply the similar modification to $\mathcal{T}_{\text{new}}$ with another variable. It is easy to see that if we have done this for all variables, one after another, the result $\mathcal{T}^*$ is a standard tree for some assignment, with $w(\mathcal{T}^*) \leq w(\mathcal{T})$. Consequently, $w(\mathcal{T}^*) = \text{OPT}$. □

Theorem 2. *Given an integer $d \geq 3$ and let $c(d)$ be a constant defined in* (2). *Let further an (α, β, γ)-gadget with $\beta > \gamma > 0$ and $\frac{\beta+\gamma}{\beta-\gamma} < c(d)$ be given. Then for any constant r, $1 < r < 1 + \frac{\gamma}{3d+2\alpha+3\beta}$, it is NP-hard to approximate the* Steiner Tree Problem *within ratio r.*

Moreover, if the gadget above is (uniformly) quasi-bipartite, the same inapproximability results apply to the (uniformly) quasi-bipartite instances of the STP as well.

Proof. Let an integer $d \geq 3$, an (α, β, γ)-gadget and a number r with the above properties be fixed.

We can choose and keep fixed from now on an $\varepsilon \in \left(0, \frac{1}{4}\right)$ such that

$$r < 1 + \frac{(1-4\varepsilon)\gamma}{3d + 2\alpha + 3\beta + 2\varepsilon\gamma}.$$

Let $k(\varepsilon)$ be an integer such that for any integer $k \geq k(\varepsilon)$ the conclusion of Theorem 1 holds. Since $\frac{\beta+\gamma}{\beta-\gamma} < c(d)$, we can consider and keep fixed from now on one $\frac{\beta+\gamma}{\beta-\gamma}$-good d-regular expander graph H with k by k bipartition such that $k \geq k(\varepsilon)$. It will play a role of a constant in our (polynomial time preserving) reduction from NP-hard problem $P_0(\varepsilon, k)$ (respectively, $P_1(\varepsilon, k)$) to the problem of approximation STP within r. (Strictly speaking, we do not construct this reduction; we only show that there exists one. But this clearly suffices for proving

NP-hardness.) Hence (with everything above fixed, including k and H) we are ready to describe the reduction. Given an instance L of the problem $P_0(\varepsilon, k)$ (respectively, $P_1(\varepsilon, k)$) with n equations, whose optimum MAX of the maximal number of satisfiable equations is promised to be either at most $n(\frac{1}{2} + \varepsilon)$ or at least $n(1 - \varepsilon)$, the reduction described above produces the corresponding instance of the STP problem. Since the assumptions of Lemma 1 are satisfied, the optimum OPT is achieved on a standard Steiner tree. Hence, using (3), the optimum OPT of the corresponding instance of the Steiner tree problem is

$$\mathrm{OPT} = n\alpha + \frac{3}{2}nd + \frac{3}{2}n\beta + (n - \mathrm{MAX})\gamma,$$

which has to be now either at least $n\alpha + \frac{3}{2}nd + \frac{3}{2}n\beta + n\big(\frac{1}{2} - \varepsilon\big)\gamma$, or at most $n\alpha + \frac{3}{2}nd + \frac{3}{2}n\beta + n\varepsilon\gamma$.

Hence even the partial decision subproblem of the STP, namely the problem to distinguish between these two cases, is NP-hard. Consequently, since

$$\frac{n\alpha + \frac{3}{2}nd + \frac{3}{2}n\beta + n\big(\frac{1}{2} - \varepsilon\big)\gamma}{n\alpha + \frac{3}{2}nd + \frac{3}{2}n\beta + n\varepsilon\gamma} = 1 + \frac{(1 - 4\varepsilon)\gamma}{2\alpha + 3d + 3\beta + 2\varepsilon\gamma} > r,$$

it is NP-hard to approximate the STP within r.

Moreover, it can be easily seen that if the gadget above is (uniformly) quasi-bipartite, our reduction produces (uniformly) quasi-bipartite instances of the STP, and the inapproximability results apply to those instances as well. □

Theorem 3. *Given an integer $d \geq 3$, denote $q(d) = \min\big\{\frac{c(d)-1}{2c(d)}, \frac{1}{4}\big\}$, $r(d) = 1 + \frac{q(d)}{3(d+1-q(d))}$, where $c(d)$ be a constant defined in* (2)*. Then for any constant r, $1 < r < r(d)$, it is NP-hard to approximate the* Steiner Tree Problem *within ratio r.*

In particular, since $c(6) > 1.76222$ implies $r(6) > 1.01063$, inapproximability within 1.01063 ($> \frac{96}{65}$) follows for the STP.

Proof. Let an integer $d \geq 3$ and a number r, $1 < r < r(d)$, be fixed. We can find $\gamma \leq \frac{1}{4}$ with $\gamma < \frac{c(d)-1}{2c(d)}$ (i.e. $\frac{1}{1-2\gamma} < c(d)$) and such that $r < 1 + \frac{\gamma}{3(d+1-\gamma)}$, and apply the Theorem 2 with $(0, 1-\gamma, \gamma)$-gadget from Example 1 (with γ as above and $\alpha = 0$ and $\beta = 1 - \gamma$). □

Theorem 4. *Given an integer $d \geq 3$ and denote by $r(d) = 1 + \frac{c(d)-1}{6d \cdot c(d) + 7c(d) - 1}$, where $c(d)$ be a constant defined in* (2)*. Then it is NP-hard to approximate solution of (uniformly) quasi-bipartite* Steiner Tree Problem *within ratio r, for any r, $1 < r < r(d)$.*

In particular, since $c(7) > 1.94606$ implies $r(7) > 1.00791$, inapproximability within 1.00791 ($> \frac{128}{127}$) follows for (uniformly) quasi-bipartite STP.

Proof. Let an integer $d \geq 3$ and a number r, $1 < r < r(d)$, be fixed. We can find $\gamma < \frac{c(d)-1}{2c(d)}$ such that $r < 1 + \frac{\gamma}{3(d+3+\gamma)}$, and apply the Theorem 2 with the uniformly quasi-bipartite $(3+3\gamma, 1-\gamma, \gamma)$-gadget from Example 2 (with γ as above and hence $\alpha = 3+3\gamma$ and $\beta = 1-\gamma$). □

Remark. The methods of this paper provide a new motivation for the study of bounds for the parameters of expanders that provably exist. For our purposes we need not restrict ourselves to expanders that can be effectively constructed; the existence is enough. There is a substantial gap between the known upper and lower bounds for parameters of the best possible expanders. We believe that lower bounds on our expander constants $c(d)$ can be improved significantly. This would improve our inapproximability results. Another way to improve the results would be to provide the gadgets with better parameters.

References

1. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification an hardness of approximation problems. Proceedings of the 33rd Annual Symposium on Fundations of Computer Science, 1992, 14–23.
2. Bern, M., Plassmann, P.: The Steiner Problem with edge lengths 1 and 2. Information Processing Letters **32** (1989) 171–176
3. Chun, F. R. K.: Spectral Graph Theory. CBMS Regional Conference Series in Mathematics, American Mathematical Society, 1997, ISSN 0160-7642, ISBN 0-8218-0315-8.
4. Håstad, J.: Some optimal inapproximability results. Proceedings of the 28rd Annual Symposium on Theory of Computing, ACM, 1997.
5. Hougardy, S., Gröpl, C., Nierhoff, T., Prömel, H. J.: Approximation algorithms for the Steiner tree problem in graphs. In Steiner Trees in Industry, (X. Cheng and D.-Z. Du, eds.), Kluwer Academic Publishers, 2001, 235–279.
6. Karp, R. M.: Reducibility among combinatorial problems, In Complexity of Computer Computations, (Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972), New York: Plenum 1972, 85–103.
7. Papadimitriou, C. H., Vempala, S.: On the Approximability of the Traveling Salesman Problem. Proceedings of the 32nd ACM Symposium on the theory of computing, Portland, 2000.
8. Robins, G., Zelikovsky, A.: Improved Steiner tree approximation in graphs. Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms 2000, 770–779.
9. Thimm, M.: On the Approximability of the Steiner Tree Problem. Proceedings of the 26th International Symposium, MFCS 2001 Mariánske Lázne, Czech Republic, August 27-31, 2001, Springer, Lecture Notes in Computer Science **2136** (2001) 678–689.

The Dominating Set Problem Is Fixed Parameter Tractable for Graphs of Bounded Genus

J. Ellis[1], H. Fan[1], and M. Fellows[2]

[1] University of Victoria, Victoria, BC. Canada V8W 3P6
jellis, hfan@csr.uvic.ca
[2] University of Newcastle, Newcastle, New South Wales, Australia
mfellows@cs.newcastle.edu.au

Abstract. We describe an algorithm for the dominating set problem with time complexity $O((24g^2 + 24g + 1)^k n^2)$ for graphs of bounded genus g, where k is the size of the set. It has previously been shown that this problem is fixed parameter tractable for planar graphs. Our method is a refinement of the earlier techniques.

Keywords: graph, genus, dominating set, fixed parameter algorithm.

1 Introduction

The DOMINATING SET problem is defined as follows.

Input: A graph $G = (V, E)$ and an integer parameter k
Question: Does there exist a set of nodes $V' \subseteq V$ such that $|V'| \leq k$ and such that for all nodes $v \in V$ either $v \in V'$ or there is an edge $uv \in E$ and $u \in V'$.

This is a classic NP-complete problem which is also apparently not *fixed parameter tractable* because it is known to be $W[2]$-complete in the W-hierarchy of fixed parameter complexity theory [DF99]. In this theory, any problem for which there is an algorithm with time complexity $O(f(k)n^\alpha)$, for some problem parameter k, where n is the number of nodes in the graph and where α is a constant independent of k and n, is said to be *fixed parameter tractable*.

The dominating set problem remains NP-complete when restricted to planar graphs. However, Fellows and Downey [DF95,DF99] gave a "search tree" algorithm for this problem which has time complexity $O(11^k n)$, when the input is restricted to planar graphs. The best search tree algorithm so far, which uses *kernelization*, is described in [AFF$^+$01] and has a time complexity of $O(8^k n)$. A tree decomposition based algorithm with time complexity $O(c^{\sqrt{k}} n)$, where $c = 4^{6\sqrt{34}}$, is given in [ABFN00]. Related work includes [AFN01a,AFN01b]. Recently, in [AFN02], it has been shown that the problem can be reduced to a kernel of size $214k$.

In this paper, using the search tree approach, we show that the dominating set problem is fixed parameter tractable for graphs of bounded genus. For graphs of genus g we prove a time complexity of $O((24g^2 + 72g + 1)^k n^2)$.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 180–189, 2002.

2 The Algorithms

2.1 The Basic Algorithm

The result in [AFF+01] is based on the following simple algorithm, Figure 1. The node set B is a subset of the nodes incident with the edges in the edge set E and *exists* is a global Boolean variable. $N(v)$ denotes the neighbors of v (not including v itself) with respect to the edge set E and $I(v)$ denotes the set of edges incident with v. The claim that the global variable *exists* is set to true iff

```
procedure dominating_set (B : node_set, E : edge_set, k : integer);

{B is a subset of the nodes incident with edges in E.
The global variable exists is set to true iff there exists a dominating
set for the nodes in B (with respect to the edges in E) of cardinality not
greater than k}

    if |B| ≤ k then exists ← true
    else if k > 0
        then Choose some u ∈ B;
            for all v ∈ (N(u) ⋃ {u}) do
                dominating_set (B − (N(v) ⋃ {v}), E − I(v), k − 1);
```

Fig. 1. The Basic Algorithm

there exists a dominating set for the nodes in B of cardinality not greater than k follows from the observation that for every node u in B either u or one of its neighbors is in the dominating set. Hence we can decide the dominating set question for any graph $G = (V, E)$ and integer k by setting *exists* to false and invoking dominating_set(V, E, k).

Suppose the maximum degree of any node in B is d and let $t(n)$ denote the time complexity of the procedure, excluding the recursive invocation, where $n = |E|$. Then the time complexity $T(n)$ of the procedure is described by the recurrence relation: $T(n, k) \leq t(n) + (d + 1)T(n - 1, k - 1)$. The solution to this recurrence is $T(n, k) = O((d + 1)^k t(n))$, as can be seen by substituting $T(n, k) \leq (c(d + 1)^k - b)t(n)$, for some constants c and b, in the recurrence and assuming that $d \geq 2$ as we will see is the case.

It is straightforward to adjust node and edge sets and even to copy entire data structures for the recursive call, all in time linear in the number of edges. Noting that for graphs of bounded genus the number of edges is $O(n)$, we see that this simple algorithm has time complexity $O((d+1)^k n)$ and the dominating set problem for any family of graphs with bounded degree is fixed parameter tractable.

2.2 The Reduction Procedure

To make use of this algorithm for graphs of bounded genus, but not of bounded degree, a reduction procedure is applied to the graph at each level of recursion. This procedure does not change the minimum size of a dominating set but guarantees that there is at least one node in B of bounded degree. Figure 2 defines the reduction procedure. Rules R1 through R6 are among the reduction rules used in [AFF$^+$01]. R7 is a new rule which generalizes one of the old rules. Given this procedure, we replace the statement "Choose some $u \in B$" in the basic algorithm by two statements: "Invoke Reduce; Choose some $u \in B$ of minimum degree".

procedure Reduce;
{We assume the node sets B and E as in Figure 1.
For convenience, we refer to the nodes in B as the "black" nodes and those nodes incident with some edge in E, but not in B, as the "white" nodes.}

repeat

R1: Remove all edges between white nodes from E;

R2: Remove all edges incident with degree 1 white nodes from E;

R3: If there is a black node w of degree 1, incident with a node u,
then remove all neighbors of u from B; add u to B;
remove all edges incident with u from E;

R4: If there is a white node u of degree 2, with black neighbors u_1 and u_2,
and an edge $u_1u_2 \in E$, then remove the edges uu_1 and uu_2 from E;

R5: If there is a white node u of degree 2, with black neighbors u_1, u_3,
and there is a black node u_2 and edges u_1u_2 and $u_2u_3 \in E$,
then remove the edges uu_1 and uu_3 from E;

R6: If there is a white node u of degree 3, with black
neighbors u_1, u_2, u_3 and there are edges u_1u_2 and $u_2u_3 \in E$
then remove the edges uu_1, uu_2 and uu_3 from E;

R7: If there are white nodes u_1 and u_2 such that $d(u_1) \leq 7$ and $N(u_1) \subseteq N(u_2)$,
then remove all the edges $\in E$ incident with u_1 from E;

until no change in E;

Fig. 2. The Reduction Procedure

Let B' and E' be the node and edge sets resulting from the application of any one of the rules in the reduction procedure to the sets B and E. One can

examine each of the rules to see that there exists a dominating set for the nodes in B with respect to the edges in E of cardinality k iff there exists a dominating set for the nodes in B' with respect to the edges in E' of cardinality k.

A graph whose node set is partitioned into black and white nodes will be referred to as a *black and white* graph. The black nodes are those yet to be dominated. The white nodes are those already dominated. A graph that is derived by applying the reduction procedure to a black and white graph will be referred to as a *reduced* black and white graph.

2.3 Time Complexity of the Reduction Process

We justify an upper bound of $O(n^2)$ on the the time complexity, $t(n)$, of the reduction procedure. We assume the use of two data structures representing the graph and associated information:

- An elaborated adjacency list in which each list is a two way linked list, and for all edges $\{u, v\}$ the entry for u in the list for v is linked directly to the entry for v in the list for u, and *vice versa*. With these extra links, an edge, once identified can be removed in constant time. A black/white indicator and a degree count are associated with each node.
- A standard adjacency matrix permits the existence of an edge to be tested in constant time, and likewise the removal of an edge.

With these data structures in mind, consider the work done in executing an entire sequence of k invocations of the dominating set procedure. It can be seen that for all rules except R7, $O(n)$ is an upper bound on the work involved in each rule. For rule R7, it seems that, although no work is required unless the degree of a white node is reduced to seven or less, any such node must be compared to all other white nodes. Thus we see no better bound than $O(n^2)$.

3 The Bounded Genus Case

Let $\mathcal{S}_g$ be the orientable compact surface of genus g. An embedding of a graph G in $\mathcal{S}_g$ is a drawing in $\mathcal{S}_g$ such that each node corresponds to a point and each edge to a line, i.e., a simple curve with two ends, and such that two lines corresponding to a pair of edges are internally disjoint, i.e., do not intersect or intersect only at their ends. The genus of G, written $\gamma(G)$, is the minimum g such that G can be embedded in $\mathcal{S}_g$. By an embedded graph we mean a graph together with its embedding in a surface, which allows us to talk about both combinatorial and geometrical properties. For instance, an edge of an embedded graph can also be seen as a line connecting two end points.

Let G be an embedded graph. A face, F, of G is a connected component of $\mathcal{S}_g - G$. The boundary-graph of F, written $b(F)$, is the subgraph of G induced by those nodes of G incident with F. A face is said to be a 2-cell if it is homeomorphic to an open disk. G is a 2-cell embedding if every face is a 2-cell. The boundary-graph of F could contain a cut edge, i.e., an edge whose removal disconnects

the graph. A cut edge can appear twice in a boundary. The degree of a face F, written $d_G(F)$ or $d(F)$, is the number of edges appearing in the boundary, edges appearing twice being counted twice. A face of degree d is called a d-face. If $b(F)$ is a cycle in G, then F is called a proper face. We say that a closed line in $\mathcal{S}_g$ is contractible if it is the boundary of an open disk. Two lines joining two different points are said to be transformable if they form a contractible closed line. It can be seen that, in a 2-cell embedded graph, the loop edge forming the boundary of a 1-face is contractible, and two multiple edges forming the boundary of a 2-face are transformable.

Lemma 3.1. [Tho95, Chapter 5, Theorem 3.8] *Every embedding of G in $S_\gamma(G)$ is a 2-cell embedding.*

Lemma 3.2. [Tho95, Chapter 5, Theorem 3.3] *Let $G = (V, E)$ be a connected simple graph with $|V| \geq 4$, then*

$$|E| \leq 6(\gamma(G) - 1) + 3|V| \tag{1}$$

Lemma 3.3. *For any two distinct points in $\mathcal{S}_g$ ($g \geq 1$), there are at most $6g$ internally disjoint and mutually non-transformable lines joining them.*

Proof. Suppose, on the contrary, that there are $s > 6g$ internally disjoint and mutually non-transformable lines joining two points u, v in $\mathcal{S}_g$. If we view u and v as nodes and the lines joining them as edges, together they constitute an embedded graph G with two nodes and $s > 6$ edges. G has s multiple edges, no 1-face, no 2-face and no cut edge. The following relation, for the embedding of any graph in $\mathcal{S}_g$, is proved in [Tho92]: $n - e + f \geq 2 - 2g$, where n is the number of nodes, e is the number of edges and f is the number of faces in the embedding. Here $n = 2$ and $3f \leq 2s$, because there are no 1-faces or 2-faces. Hence $s \leq 6g$, which would contradict the assumption with which we began the proof. □

Let $G = (V, E)$ be graph and x, y be nodes of G, we denote by $G + xy$ the simple graph $(V, E \cup \{xy\})$. We denote by $\delta(G)$ the minimum degree of G.

Lemma 3.4. *Let $G = (V, E)$ be a connected graph for which $\delta(G) \geq 3$. If G is not 3-connected then G has a 2-node cut $\{x, y\}$ such that there is a component C of $G - \{x, y\}$ such that $G[V(C) \cup \{x, y\}] + xy$ is 3-connected.*

Proof. Because $\delta(G) \geq 3$, $|V(G)| = 4$ implies that G is isomorphic to K_4, which is 3-connected. No smaller graph is 3-connected.

Suppose $|V(G)| > 4$ and G is not 3-connected. Then either there exists a cut node $\{x\}$ which is in a pendant block or, if not, there exists a 2-cut $\{x, y\}$.

There Is No Cut Node

Then there exists a 2-cut $\{x, y\}$. Let C be a component of $G - \{x, y\}$ and let $G' = G[V(C) \cup \{x, y\}] + xy$. G' is 2-connected. Any such C has at least 2 nodes, else there is a node in G of degree 2. Hence all such G' contain at least 4 nodes.

If $|V'| = 4$ then $d_{G'}(x) = d_{G'}(y) = 3$, else there is a node in G of degree 2. Hence G' is isomorphic to K_4.

If $|V'| > 4$, consider the following procedure.

while G' is not 3-connected **do**
Let $\{a, b\}$ be a 2-cut in G';
Let C be a component in $G' - \{a, b\}$ not containing both x and y;
$x \leftarrow a$; $y \leftarrow b$;
$G' \leftarrow G[V(C) \cup \{x, y\}] + xy$
endwhile

G' does not contain a cut node, hence, if it is not 3-connected there is a 2-cut $\{a, b\}$ in G' such that $\{a, b\} \neq \{x, y\}$. We note that $\{a, b\}$ is also a cut in G. The procedure must terminate, with G' being 3-connected, because the number of nodes in G decreases at each iteration. If not before, it must terminate when $|V'| = 4$ because then $G' = K_4$, else one of the remaining two nodes had degree less than three in G.

There Exists a Cut Node

There must exist a cut node x which is in some pendant block (a maximal 2-connected component of G) say B. Note that since $\delta(G) \geq 3$, B must contain at least 4 nodes.

If B is 3-connected, then $\{x, a\}$, where a is a neighbour of x in B, is a 2-cut in G satisfying the lemma.

If B is not 3-connected, then $|V(B)| > 4$ and there exists a 2-cut, say $\{u, v\}$, in B, since there are no cut nodes. One of the components, say C, in $B - \{u, v\}$ does not contain x. If $B[C \bigcup \{u, v\}] + xy$ is not three connected then, by the argument used in the 2-connected case, there exists another 2-cut in C with the desired properties.

□

Lemma 3.5. *If $G = (B \bigcup W, E)$ is reduced, then the white nodes constitute an independent set.*

Proof. This follows from rule R1 of the reduction procedure, which is inside the loop. Hence, although rule R3 can add white nodes and hence maybe edges between white nodes, rule R1 will be again enacted and remove any edges between white nodes. □

Lemma 3.6. *For all d, if there exists a reduced black and white graph G in which the minimum degree of any black node is d then there there exists a reduced graph G' in which the minimum degree of any black node is also d and no white node has degree less than 3.*

Proof. By rule R2 of the reduction procedure, G has no white nodes of degree 1. Let u be a white node of degree 2 with neighbours x and y, which are necessarily

black, by rule R1. By rule R4, there is no edge xy. Let G' be obtained from G by replacing all white nodes of degree 2 by an edge between their neighbours. Note that this operation does not change the the minimum degree of any black node. Suppose G' is not reduced. Then the only rule that could possibly apply is R6, i.e., there is some white node u of degree 3 in G' with black neighbours and edges between two pairs of these neighbours. Because G is reduced, rule R6 did not apply to u in G. Hence one of the edges between the neighbours of u must have been created in the derivation of G' from G, i.e., one of these edges replaced a white node u' of degree 2. But this implies that R7 would apply in G to u and u', contradicting the assumption that G is reduced. □

We are now ready to present the main result of this paper.

Theorem 3.7. *If $G = (B \bigcup W, E)$ is a reduced black and white graph of genus at most g, then there exists a node $b \in B$ such that $d_G(b) \leq 24g(g+3)$.*

Proof. A better bound for planar graphs, where $g = 0$, has already been proved in [AFF+01]. So let us assume that $g \geq 1$. In contradiction to the lemma, suppose that $G = (B \bigcup W, E)$ is a reduced black and white graph with genus $\gamma(G) \leq g$ and that every node $u \in B$ is such that $d_G(u) \geq 24g(g+3)+1$. We may assume that G is a connected simple graph and that, by Lemma 3.6, for any $u \in W$, $d_G(u) \geq 3$. Consider G to be embedded in $\mathcal{S}_{\gamma(G)}$. Then the embedding is a 2-cell embedding by Lemma 3.1. $W \neq \emptyset$ because otherwise $|E| > (24g(g+3)+1)|V|/2 > 6(g-1)+3|V| \geq 6(\gamma(G)-1)+3|V|$, contradicting Equation (1). By Lemmas 3.5 and 3.6, we have that $|B| \geq 3$. We consider two cases: either G is 3-connected or it is not.

G Is 3-Connected

First we show that $|B| \geq 6$. Suppose $3 \leq |B| \leq 5$. The maximum number of mutually non-inclusive subsets of five elements is $\binom{5}{\lceil \frac{5}{2} \rceil} = 10$, [Bol86, Chapter 3, Theorem 1]. But $|W| \geq 24g(g+3) - 4 \geq 92$ and so there must exist two white nodes such that the neighbours of one of them constitute a subset of the neighbours the other. But that contradicts R7 of the reduction procedure.

Let u be a white node of G, and let $v_0, v_1, \ldots, v_{t-1}$ be the sequence of nodes adjacent to u in clockwise order where $t = d_G(u)$. For i from 0 to $t-1$, let F_i be the face containing boundary edges $e_i = uv_i$ and $uv_{i+1 \bmod t}$ in the current graph. If F_i is not a proper 3-face, then add an edge $e'_i = v_i v_{i+1 \bmod t}$ so that e_i, e_{i+1}, e'_i form a proper 3-face.

Let us repeat this operation on all white nodes and denote the resulting embedded graph by G_1. Note that G_1 may contain multiple edges, but that no two multiple edges form the boundary of a 2-cell because G is a connected simple graph and the operation does not create a 2-face. Moreover, no two multiple edges are transformable in $\mathcal{S}_\gamma(G)$ since otherwise the two end nodes of two transformable edges would form a 2-node cut of G which contradicts our assumption that G is 3-connected. G_1 is uniquely determined, up to isomorphism, by the embedding of G.

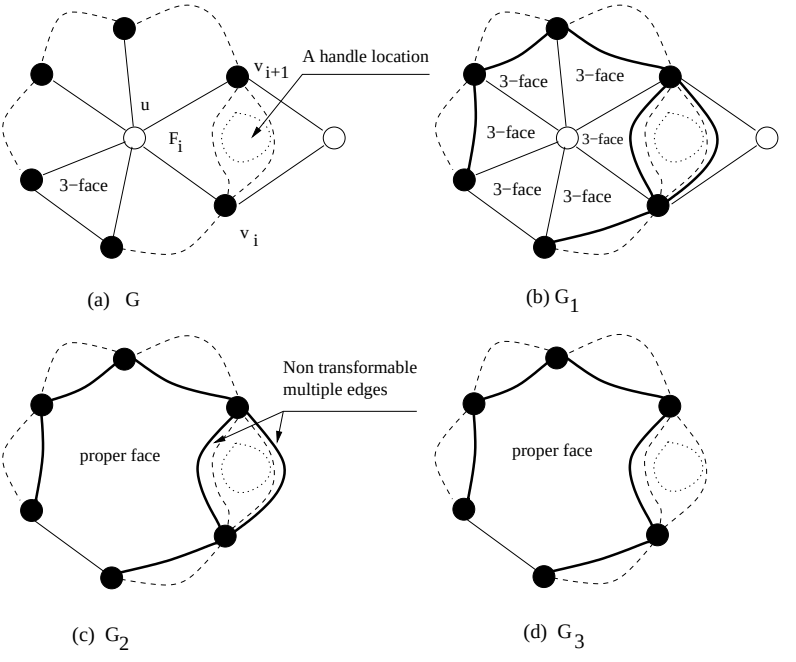


Fig. 3. Embedding Operations

Now let G_2 be the embedded graph obtained from G_1 by removing all white nodes and their incident edges. Then G_2 is a 2-connected 2-cell embedded graph in which no two multiple edges are transformable.

By construction, a white node of G_1 must be located in what becomes a proper face of G_2, and a proper face of G_2 contains at most one such white node in G_1. Since there are at most $d_{G_2}(x)$ faces incident with any node x in G_2, there are at most $d_{G_2}(x)$ white nodes adjacent to x in G_1 and hence

$$d_{G_1}(x) \leq 2d_{G_2}(x) \tag{2}$$

Let G_3 be the graph obtained from G_2 by reducing instances of multiple edges to one edge. See Figure 3 for the construction of G_1 through G_3. Then $\gamma(G_3) \leq \gamma(G) \leq g$ and, by Lemma 3.2, we have

$$|E(G_3)| \leq 6(\gamma(G_3) - 1) + 3|B| \leq 6(\gamma(G) - 1) + 3|B| \leq 6(g - 1) + 3|B|.$$

Since $\delta(G_3)|B| \leq \sum_{u \in V(G_3)} d_{G_3}(u) = 2|E(G_3)|$, $\delta(G_3)|B| \leq 12(g-1) + 6|B|$ and hence $\delta(G_3) \leq 12(g-1)/|B| + 6 \leq 12(g-1)/6 + 6 = 2(g+2)$ and there exists $y \in B$ such that

$$d_{G_3}(y) \leq 2(g + 2). \tag{3}$$

Since the G_2 is an embedded graph in a surface of genus $\gamma(G)$ and no two multiple edges are transformable, by Lemma 3.3 there are at most $6g$ multiple edges joining a pair of nodes in G_2. Then by inequality (3) we have

$$d_{G_2}(y) \leq 6g \times d_{G_3}(y) \leq 12g(g+2)$$

By inequality (2) we obtain

$$d_G(y) \leq d_{G_1}(y) \leq 2d_{G_2}(y) \leq 24g(g+2) < 24g(g+3)+1.$$

But this contradicts the assumption we used to begin the proof.

G Is Not 3-Connected

By Lemma 3.4, G has a 2-node cut $\{x, y\}$ such that $G - \{x, y\}$ has a component C such that $G_0 = (V(C) \bigcup \{x, y\}, E(G[V(C) \bigcup \{x, y\}]) \cup \{xy\})$ is 3-connected. Since G contains a path connecting x and y which is internally disjoint with $V(C)$, the embedding of G induces an embedding of G_0 in $\mathcal{S}_{\gamma(G)}$ whether or not xy is an edge of G. Therefore $\gamma(G_0) \leq \gamma(G) \leq g$.

Let $B' = B \cap (V(C) \bigcup \{x, y\})$ and $W' = W \cap (V(C) \bigcup \{x, y\})$. We show that $|B'| \geq 8$. Since W is an independent set and $d_G(v) \geq 3, v \in W$, we must have $B' \setminus \{x, y\} \neq \emptyset$. Let $u \in B' \setminus \{x, y\}$, then $d_{G_0}(u) = d_G(u) \geq 24g(g+3)+1 \geq 97$. Then we have $|B'|+|W'| \geq 98$. If $|B'| \leq 7$, then $|W'| \geq 91$ and $|W' \setminus \{x, y\}| \geq 89$. The neighbours of nodes in $W' \setminus \{x, y\}$ are in B', and so there must exist two white nodes such that the neighbours of one of them constitute a subset of the neighbours the other. This is because the maximum number of mutually non-inclusive subsets of seven elements is $\binom{7}{\lceil \frac{7}{2} \rceil} = 35 < 89$, [Bol86, Chapter 3, Theorem 1]. But this would contradict R7 of the reduction procedure.

Now let $B_0 = B \cup \{x, y\}$ be the black node set for G_0 and consider G_0 to be embedded in $\mathcal{S}_{\gamma(G_0)}$. Let G_1, G_2 and G_3 be derived from G_0 by exactly the same process as described in the previous case. The inequality (2) still holds.

By Lemma 3.2, we have

$$|E(G_3)| \leq 6(\gamma(G_3) - 1) + 3|B_0| \leq 6(g-1) + 3|B_0|$$

Since $2|E(G_3)| = \sum_{u \in V(G_3)} d_{G_3}(u) \geq \delta(G_3)|B_0|$ and $|B_0| \geq |B'| \geq 8$, there is at least one black node $z \in B_0 \setminus \{x, y\}$ such that

$$d_{G_3}(z) \leq \frac{12(g-1)+6|B_0|}{|B_0|-2} = \frac{12g}{|B_0|-2} + 6 \leq \frac{12g}{6} + 6 \leq 2(g+3) \qquad (4)$$

Since G_2 is an embedded graph in $\mathcal{S}_{\gamma(G_0)}$ and $\gamma(G_0) \leq g$ and no two multiple edges are transformable, there are at most $6g$ internally disjoint edges joining a pair of points, by Lemma 3.3.

Then, by inequalities (4) and (2), we have

$$d_{G_2}(z) \leq 6g \times d_{G_3}(z) \leq 12g(g+3)$$

Therefore

$$d_G(z) = d_{G_0}(z) \leq d_{G_1}(z) \leq 2d_{G_2}(z) \leq 24g(g+3) < 24g(g+3) + 1.$$

But this contradicts the assumption we used to begin the proof. □

4 Conclusions

In Section 2 we showed a time complexity of $O(n^2)$ for the reduction procedure. In Section 3 we showed that in any reduced black and white graphs of bounded genus g there exists at least one node of degree at most $24g(g+3)$. Combining these results yields a time complexity of $O((24g^2 + 72g + 1)^k n^2)$ for the basic algorithm, after the reduction procedure has been incorporated.

It would be interesting to know if the other results known for planar graphs, for example, the existence of a linear kernel and the $O(2^{O(\sqrt{k})}n^c)$ algorithm via tree decomposition, can be extended to graphs of bounded genus.

References

[ABFN00] J. Alber, H. L. Bodlaender, H. Fernau, and R. Niedermeier. Fixed parameter algorithms for planar dominating set and related problems. In M. M. Halldórsson, editor, *7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *LNCS*, pages 97–110, 2000.

[AFF+01] J. Alber, H. Fan, M. R. Fellows, H. Fernau, R. Niedermeier, F. Rosamond, and U. Stege. Refined search tree techniques for the PLANAR DOMINATING SET problem. In J. Sgall, A. Pultr, and P. Kolman, editors, *Mathematical Foundations of Computer Science*, volume 2136 of *LNCS*, pages 111–122. Springer, 2001.

[AFN01a] J. Alber, H. Fernau, and R. Niedermeier. Graph separators: a parameterized view. Technical Report WSI–2001–8, Universität Tübingen (Germany), Wilhelm-Schickard-Institut für Informatik, 2001.

[AFN01b] J. Alber, H. Fernau, and R. Niedermeier. Parameterized complexity: exponential speedup for planar graph problems. In *28th ICALP 2001*, volume 2076 of *LNCS*, pages 261–272. Springer, 2001.

[AFN02] J. Alber, M. Fellows, and R. Niedermeier. Efficient data reduction for dominating set: A linear problem kernel for the planar case. In *8th Scandinavian Workshop on Algorithm Theory*, LNCS. Springer, 2002.

[Bol86] B. Bollóbas. *Combinatorics*. Cambridge University Press, 1986.

[DF95] R. G. Downey and M. R. Fellows. Parameterized computational feasibility. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 219–244. Birkhäuser, 1995.

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[Tho92] C. Thomassen. The jordan-schonfliess theorem and the classification of surfaces. *Amer. Math. Monthly*, 99:116–130, 1992.

[Tho95] C. Thomassen. Embeddings and minors. In R. L. Graham, M. Groschel, and L. Lovász, editors, *Handbook of Combinatorics Volume I*. The MIT Press, 1995.

The Dynamic Vertex Minimum Problem and Its Application to Clustering-Type Approximation Algorithms

Harold N. Gabow[1] and Seth Pettie[2,⋆]

[1] Department of Computer Science, University of Colorado at Boulder, Boulder, CO 80309, USA hal@cs.colorado.edu
[2] Department of Computer Science, University of Texas at Austin, Austin TX 78712, USA seth@cs.utexas.edu

Abstract. The dynamic vertex minimum problem (DVMP) is to maintain the minimum cost edge in a graph that is subject to vertex additions and deletions. DVMP abstracts the clustering operation that is used in the primal-dual approximation scheme of Goemans and Williamson (GW). We present an algorithm for DVMP that immediately leads to the best-known time bounds for the GW approximation algorithm for problems that require a metric space. These bounds include time $O(n^2)$ for the prize-collecting TSP and other direct applications of the GW algorithm (for n the number of vertices) as well as the best-known time bounds for approximating the k-MST and minimum latency problems, where the GW algorithm is used repeatedly as a subroutine. Although the improvement over previous time bounds is by only a sublogarithmic factor, our bound is asymptotically optimal in the dense case, and the data structures used are relatively simple.

1 Introduction

Many approximation algorithms are applications of the primal-dual algorithm of Goemans and Williamson (GW) [12]. (This algorithm is rooted in the approach proposed by Agrawal, Klein and Ravi [2].) This paper determines the asymptotic time complexity of the GW clustering operation on metric spaces. Although our improvement is a sublogarithmic factor, the issue is important from a theoretic viewpoint. Also our algorithm uses simple data structures that will not incur much overhead in a real implementation.

Aside from operations involving a problem-specific oracle, the only difficulty in implementing the GW algorithm is the clustering operation which determines the next components to merge. Goemans and Williamson's original implementation [12] uses time $O(n^2 \log n)$. This was improved to $O(n(n + \sqrt{m \log \log n}))$ [9] and $O(n\sqrt{m} \log n)$ [15]. Here and throughout this paper n and m denote the number of vertices and edges in the given graph, respectively. Regarding time

⋆ This work was supported in part by Texas Advanced Research Program Grant 003658-0029-1999, NSF Grant CCR-9988160 and an MCD Graduate Fellowship.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 190–199, 2002.

bounds one should bear in mind that many applications of the GW algorithm are for the dense case $m = \Theta(n^2)$ (see below). We improve the above bounds to $O(n\sqrt{m})$. Cole et. al.[5] present a modified version of the GW clustering algorithm that runs in time $O(km \log^2 n)$. Here k is an arbitrary constant, and the approximation factor of the modified algorithm increases (i.e., worsens) by the small additive term $O(1/n^k)$. This time bound is a substantial improvement for sparse graphs. Still in the dense case this algorithm is slower than the original GW implementation (and slightly less accurate).

One reason that dense graphs arise is assumption of the triangle inequality. Because of this our algorithm gives the best known time bound $O(n^2)$ for these applications of the GW algorithm [12]: 4-approximation algorithm for the exact tree, exact path and exact cycle partitioning problems; 2-approximation algorithm for the prize-collecting TSP (see [14] for the TSP time bound); and finally the 2-approximation for minimum cost perfect matching (whose primary motivation is speed). The GW approximation algorithms for prize-collecting Steiner tree and TSP, on metric spaces, are used as subroutines in several constant factor approximation algorithms for the minimum latency problem and the k-MST problem. For minimum latency these include [3,10] and the best-known 10.77-approximation algorithm of [7]. For k-MST they include [4], and the best-known 3-approximation of [7] for the rooted case and 2.5-approximation of [1] for the unrooted case. In all of these algorithms multiple executions of the GW algorithm dominate the running time, so our implementation improves these time bounds too (although the precise time bounds are higher).

For the approximation algorithm for survivable network design [9], [18], [8] the given graph can be sparse but there are additional quadratic computations. Our algorithm achieves time $O((\gamma n)^2)$ when the maximum desired connectivity is γ. (The bound of [9] is $O((\gamma n)^2 + \gamma n\sqrt{m \log \log n})$. For $\gamma = 2$ Cole et. al. avoid the quadratic computations and achieve time bound $O(km \log^2 n)$ for k as above.) For other applications of the GW algorithm where the time is dominated by clustering (generalized Steiner tree problem, prize-collecting Steiner tree problem, nonfixed point-to-point connection, and T-joins [12]) our bound is $O(n\sqrt{m})$, which improves the previous strict implementations of GW and improves the algorithm of Cole et. al. [5] in very dense graphs (but of course not in sparse graphs).

We obtain our results by solving the "dynamic vertex minimum problem" (DVMP). This problem is to keep track of the minimum cost edge in a graph where vertices can be inserted and deleted. Ignoring the graph structure it is obvious that $O(m \log n)$ is the best time bound possible, for m the total number of edges. Taking advantage of the graph structure we achieve time $O(n^2)$, for n the total number of vertices. This result immediately implies a time bound of $O(n^2)$ for GW clustering. It gives all the dense graph time bounds listed above. Our solution to DVMP is based on an amortized analysis of a system of binomial queues.

We apply the DVMP algorithm to implement the GW clustering algorithm on sparse graphs in time $O(n\sqrt{m})$. We actually solve the more general "merging

minimum problem" (MMP). This problem is to keep track of the minimum cost edge in a graph whose vertices can be contracted. The costs of edges affected by the contraction are allowed to change, subject to a "monotonicity property." We solve the MMP using our DVMP algorithm and a grouping technique.

Section 2 gives our solution to the DVMP. This immediately implements GW clustering for dense graphs. Section 3 of the complete paper [11] solves the MMP; this section is omitted here because of space limitations.

2 Dynamic Vertex Minimum Problem

The *dynamic vertex minimum problem (DVMP)* concerns an undirected graph G where each edge e has a real-valued cost $c[e]$. The graph is initially empty. We wish to process (on-line) a sequence of operations of the following types:

`Add_vertex`(v)	:	add v as a new vertex with no edges.
`Add_edge`(e)	:	add edge e with cost $c[e]$.
`Delete_vertex`(v)	:	delete vertex v and all edges incident to v.
`Find_min`	:	return the edge currently in G that has smallest cost.

This section shows how to support `Add_vertex`, `Add_edge` and `Delete_vertex` operations in $O(1)$ amortized time, and `Find_min` in worst-case time linear in the number of vertices currently in G. For convenience we assume the edge costs are totally-ordered. If two edges actually have the same cost we can break the tie using lexicographic order of the vertex indices.

Our high-level approach is to store the edges incident on each vertex in a heap. Since the DVMP only involves the edge of globally minimum cost, these heaps ignore important information: An edge that is not the smallest in one of its heaps is not a candidate for the global minimum, even if it is the smallest in its other heap. We capture this principle using a notion of "active" and "inactive" edges (defined precisely below). This allows us to economize on the number of heap operations.

2.1 The Algorithm

This section presents the data structure and algorithm, and proves correctness of the implementation.

Our data structure is a collection of heaps. Our heaps are a variant of the standard binomial queue data structure [16]. As usual, each heap is a collection of heap-ordered binomial trees. However our structure differs from the standard definition in two ways. First and most importantly, one heap is allowed to contain an arbitrary number of binomial trees of a given rank. Second, a lazy strategy is used for deletion: Heap elements are marked when they get deleted (elements are initially unmarked). The root of a heap is never marked.

The data structure consists of $|V(G)|$ heaps, one for each vertex. We denote by $H(u)$ the heap associated with $u \in V(G)$. Elements of $H(u)$ correspond to edges incident on u. An edge $\{u, v\}$ appears in both $H(u)$ and $H(v)$; we

differentiate these two elements with the notation (u, v) and (v, u), respectively. Let $twin(u, v)$ be synonymous with (v, u). $H(u)$ may contain marked elements (u, x) (for edges previously incident to u that have been deleted). But as already mentioned, such marked elements are never tree roots.

Each heap $H(u)$ is a collection of binomial trees divided into two groups: the *active* trees and the *inactive* trees. We sometimes refer to a whole tree by its root. Hence an *active root* is the root of an active tree. As usual the *rank* of an element e, denoted $rank(e)$, is the number of children it has, which is also the logarithm (base 2) of the size of the subtree rooted at e. The following invariant characterizes the active and inactive trees.

DVMP Invariant

(i) For all elements e, $rank(e) \leq rank(twin(e)) + 1$.
(ii) Consider a tree root f. If $rank(f) > rank(twin(f))$ then f is inactive. If $rank(f) \leq rank(twin(f))$ and f is inactive then $twin(f)$ is either active or a nonroot.
(iii) Consider a vertex u. $H(u)$ contains at most one active root per rank k, denoted $r_u(k)$ if it exists. At all times $s_u(k)$ points to the element with minimum cost among $\{r_u(k), r_u(k+1), \ldots\}$.

To better understand (ii) consider elements f and $twin(f)$ that are both roots. If the twin roots have equal rank then at least one of them is active. If the twins have unequal rank then the smaller rank element is active while the larger one is not.

Here is some motivation for the DVMP Invariant. The purpose of (i) is that the ranks of an item and its twin should not differ too much. To maintain the first part of (iii) we will sometimes merge two active trees of the same rank. This increments the rank of the resulting tree's root by one. If this happened too often the ranks of an item and its twin could differ by an arbitrary amount, violating (i). To avoid this when a root's rank is one more than that of its twin (ii) makes the root inactive. We never merge inactive trees. Hence we do not violate (i).

One consequence of the DVMP Invariant is that the minimum cost edge is easy to find:

Lemma 1. *If $\{u, v\}$ is the edge with minimum cost in G then either $s_u(0)$ or $s_v(0)$ points to it.*

Proof. Because $\{u, v\}$ is of minimum cost, (u, v) and (v, u) must be roots in $H(u)$ and $H(v)$, respectively. DVMP Invariant (ii) implies that either (u, v) or (v, u) is active. DVMP Invariant (iii) implies that in general, $s_u(0)$ points to the minimum element in an active tree of vertex u. Hence either $s_u(0)$ or $s_v(0)$ points to $\{u, v\}$. (The fact that nonroot elements can be marked, i.e., deleted, has no affect on this argument.) ∎

The procedure for `Find_min` follows directly from Lemma 1. We simply take the minimum cost element pointed to by $s_u(0)$, over all $u \in V(G)$.

The `Add_vertex`, `Add_edge` and `Delete_vertex` operations are implemented in a lazy fashion: They perform the least amount of work necessary to restore the DVMP Invariant. Each of these operations makes use of the routines `Activate`(e) and `Deactivate`(e). The purpose of these routines is to change a tree root $e = (u, v)$ from the inactive to the active state, or the reverse. Both these changes of state may violate DVMP Invariant (iii): Making a root e active may create two active roots of rank $rank(e)$. Making root e active or inactive may make $r_u(rank(e))$ or $s_u(0), \ldots, s_u(rank(e))$ out-of-date. The routines `Activate`(e) and `Deactivate`(e) repair these violations, as follows.

`Deactivate`$(e = (u, v))$

It is assumed that e is an active root. Furthermore deactivating e will not violate DVMP Invariant (ii).

1. Move the tree rooted at e to the set of inactive trees in $H(u)$.
2. If $r_u(rank(e)) = e$, set $r_u(rank(e)) =$ **nil**.
3. Update $s_u(0), \ldots, s_u(rank(e))$.

`Activate`$(e = (u, v))$

It is assumed that e is an inactive tree root and $rank(e) \leq rank(twin(e))$.

1. Remove the tree rooted at e from the set of inactive trees in $H(u)$.
2. Let $cur := e$.

   *The following loop sets $r_u(rank(cur)) = cur$ unless $r_u(rank(cur)) \neq$ **nil**, in which case merging is necessary.*
3. LOOP {
4. Let $k := rank(cur)$.
5. If $r_u(k) =$ **nil**
6. Let $r_u(k) := cur$.
7. Update $s_u(0), \ldots, s_u(k)$.
8. EXIT THE LOOP.
9. Otherwise, merge the trees rooted at cur and $r_u(k)$, and let cur be the resulting root.
10. Set $r_u(k) :=$ **nil**.
11. If $rank(cur) > rank(twin(cur))$
12. `Deactivate`(cur).
13. `Activate`$(twin(cur))$ if $twin(cur)$ is an inactive root.
14. EXIT THE LOOP.
15. }

It is clear that `Deactivate` works correctly. `Activate` is more complicated because of the tail recursion in Step 13. Its correctness amounts to the following fact.

Lemma 2. `Activate` *eventually returns with the DVMP Invariant intact.*

Proof. We will prove by induction that each time Step 5 of `Activate` is reached,

(i) $rank(twin(cur)) \geq k = rank(cur)$;
(ii) the only possible violation of the DVMP Invariant is part (iii), specifically, $H(u)$ can contain two active nodes of rank k, *cur* and $r_u(k)$, and the values $s_u(0), \ldots, s_u(k)$ can be incorrect.

In the inductive argument we will also note that the lemma holds whenever `Activate` returns.

For the base case of the induction note that activating e (in Step 1) cannot introduce a violation of DVMP Invariant (i)–(ii). Hence the first time Step 5 is reached, only DVMP Invariant (iii) for u can fail and inductive assertion (ii) holds. Assertion (i) follows from the corresponding inequality on ranks in the entry condition of `Activate`.

Now consider Step 5. If $r_u(k) =$ **nil**, Steps 6–7 restore DVMP Invariant (iii). Then `Activate` returns with the DVMP Invariant holding, as desired. So suppose $r_u(k) \neq$ **nil**, i.e., there is a previously existing active tree of rank k.

The inequality $rank(f) \leq rank(twin(f))$ holds for both $f = cur$ (by inductive assumption) and $f = r_u(k)$ (by DVMP Invariant (ii)–(iii)). Step 9 merges trees *cur* and $r_u(k)$ and makes *cur* point to the new tree root, which has rank $k+1$. The previous inequality (along with DVMP Invariant (i)) shows that now $rank(cur)$ is equal to either $rank(twin(cur))$ or $rank(twin(cur)) + 1$. In the first case, since *cur* is active DVMP Invariant (ii) permits $twin(cur)$ to be active or inactive. Hence the algorithm proceeds to the next execution of Step 5 with inductive assertions (i)–(ii) intact. So the first case is correct.

In the second case DVMP Invariant (ii) requires that *cur* be inactive and $twin(cur)$ be active if it is a root. Step 12 makes *cur* inactive and `Deactivate` restores DVMP Invariant (iii). (Note the entry condition to `Deactivate` is satisfied.) The recursive call of Step 13 fixes up DVMP Invariant (ii) in its Step 1. (Again note the entry condition to `Activate` is satisfied.) Then Step 5 is reached with inductive assertions (i)–(ii) intact. This completes the induction.

It remains to show that `Activate` eventually returns. This is clear, since every time it reaches Step 5 the number of trees in the data structure has decreased (in the merge of Step 9). ∎

Now consider the remaining operations `Add_vertex`, `Add_edge` and `Delete_vertex`. `Add_vertex` is trivial. The routines for `Add_edge` and `Delete_vertex` are given below. `Delete_vertex` works in a lazy fashion: We mark heap elements when they get deleted. We keep a marked element in the heap as long as possible, i.e., until all its ancestors become marked.

`Add_edge`$(\{u, v\})$
1. Create rank 0 nodes (u, v) and (v, u).
2. Put (u, v) and (v, u) in the inactive sets of $H(u)$ and $H(v)$, respectively.
3. `Activate`(u, v).

`Delete_vertex`(u)
1. For each element $e \in H(u)$, mark $twin(e)$ as deleted.
2. For each tree root $f = twin(e)$ marked in Step 1,
3. `Deactivate`(f) if it is active.
4. For each unmarked element g that is in the tree of f and has all its ancestors marked,
5. Designate g an inactive tree root (remove all its marked ancestors).
6. If $twin(g)$ is an inactive root,
7. If $rank(g) \le rank(twin(g))$, `Activate`($g$)
8. Else `Activate`($twin(g)$).

It is obvious that `Add_edge` is correct, so we turn to `Delete_vertex`. Step 1 can discard all the nodes of $H(u)$ since they are no longer needed. Step 4 finds the nodes g by a top-down exploration of the tree rooted at f. Step 2 ensures that every new tree root is found. Step 5 causes DVMP Invariant (ii) to fail if $twin(g)$ is an inactive root. In that case if g and $twin(g)$ have equal ranks one of them should be active; if they have unequal ranks the smaller rank element should be active. However all the rest of the DVMP Invariant is preserved in Step 5. Thus Steps 6–8 restore DVMP Invariant (ii). (Note the entry conditions to `Activate` and `Deactivate` are always satisfied.) We conclude that `Delete_vertex` works correctly.

We close this section with the final details of the data structure. Each value $twin(e)$ is represented by a pointer, so nodes (u, v) and (v, u) point to each other. The set of inactive trees in each heap $H(u)$ is represented as a doubly-linked list. Now almost every step of the four routines takes constant time. The exceptions are first, the updates of s_u: Step 3 of `Deactivate` takes $O(rank(e)+1)$ time and Step 7 of `Activate` takes $O(k+1)$ time. (We compute $s_u(i)$ in $O(1)$ time using the value of $s_u(i+1)$). Second, the top-down search in Step 4 of `Delete_vertex` amounts to $O(1)$ time for the root f plus $O(1)$ time for each binomial tree edge that is explored (and removed).

2.2 Timing Analysis

This section proves the claimed time bounds. It is immediate that the worst-case bounds for `Find_min` and `Add_vertex` are $O(n)$ and $O(1)$ respectively, where n is the current number of vertices. We show below that `Add_edge` and `Delete_vertex` use $O(1)$ amortized time.

To start off we charge each edge of G $O(1)$ time to account for work in its creation and destruction, specifically Steps 1–2 of `Add_edge` plus the possible time for marking the edge in Steps 1–2 of `Delete_vertex`. It is easy to see that the remaining work performed by our algorithm is linear in the number of comparisons between edge costs. (Specifically, the time for `Deactivate` is dominated by the comparisons to update s_u in Step 3; `Activate` is dominated by the comparison to merge binomial trees (Step 9) and the comparisons to update s_u (Step 7); in `Delete_vertex` each remaining unit of work corresponds to the

destruction of a binomial tree edge, i.e., a previous comparison.) It therefore suffices to bound the number of comparisons.

We do this using the accounting method of amortized analysis [6,17]. Define 1 credit to be the work required for 1 comparison. We will maintain the following invariant after each operation:

Credit Invariant

(i) Every heap element has $C = O(1)$ credits.
(ii) Every root of rank k has an additional $k + 4$ credits.
(iii) Every nonroot of rank k with a marked parent has an additional $2k + 5$ credits.

The precise value of the constant C will be determined below.

Note that each call `Deactivate`(e) uses $rank(e) + 1$ credits. We will require that each call `Activate`(e) is given $rank(e) + 1$ credits. Using this credit system the amortized cost of `Add_edge` is $2(C + 4) + 1$: $C + 4$ credits per rank 0 element created plus 1 credit for the call to `Activate`. Thus `Add_edge` takes $O(1)$ amortized time as desired.

Amortized Cost of `Activate` When `Activate`(e) is called $rank(e) + 1$ credits are available to be spent. We maintain that at each iteration of the loop (Step 3) at least $k + 1$ credits are available, for $k = rank(cur)$. This is clearly true for the first iteration.

Suppose in Step 5, $r_u(k) =$ **nil**. The only remaining comparisons in this call to `Activate` are for updating $s_u(0), \ldots, s_u(k)$. We pay for these with the $k + 1$ available credits.

Suppose now $r_u(k) \neq$ **nil**. Step 9 merges *cur* and $r_u(k)$, two rank k trees, producing a rank $k + 1$ tree, also denoted *cur*. The merge changes one root into a nonroot, releasing $k+4$ credits. We use one credit to pay for the comparison of the merge; additionally the new rank $k + 1$ root requires one more credit. This leaves a total of $(k + 1) + (k + 2) = 2k + 3$ credits available.

Suppose Steps 12–14 are executed. We pay for `Deactivate`(cur) in Step 12 with $k + 2$ credits. (Actually we could save a comparison in `Deactivate`, since it need not update $s_u(rank(e))$.) Since $rank(twin(cur)) = k$, we can pay for the call to `Activate`$(twin(cur))$ in Step 13 (if necessary) with the remaining $k + 1$ credits.

Finally suppose the 'If' statement in Step 11 fails. The loop returns to Step 3 with $2k + 3 \geq k + 2$ available credits, as called for.

Amortized Cost of `Delete_vertex` Consider an operation `Delete_vertex`(u) and an element $(u, v) \in H(u)$ with rank k. DVMP Invariant (i) ensures that $rank(v, u) \leq k + 1$. When Step 1 marks (v, u) Credit Invariant (iii) requires credits to be placed on the children of (v, u). Let us temporarily assume this has been done and discuss the rest of `Delete_vertex` before returning to this issue.

In Step 3 a possible operation `Deactivate`(v, u) requires at most $k + 2$ credits, paid for by the $k + 4$ credits on (v, u).

Now consider an unmarked element g as in Step 4. The cost of discovering g and processing it in Steps 4–6 has already been associated with merge comparisons. In addition if $rank(g) = j$ we need $2j + 5$ credits: Credit Invariant (ii) requires $j + 4$ credits when g becomes a root (Step 5), plus we need at most $j + 1$ credits to pay for the call to `Activate` for g or its twin (Steps 7–8). Credit Invariant (iii) for g gives the $2j + 5$ needed credits.

It remains only to explain how Credit Invariant (iii) is maintained when (v, u) is marked in Step 1. (v, u) has at most $k + 1$ children, one child of each rank $i = 0, \ldots, k$. So Credit Invariant (iii) requires a total of at most $\sum_{i=0}^{k}(2i + 5) = (k+1)(k+5)$ credits. We consider two cases, depending on whether or not (u, v) is a root of $H(u)$.

$H(u)$ contains at most $|H(u)|/2^{k+1}$ rank k nonroot elements (u, v), since the parent of such a nonroot has 2^{k+1} descendants. So the total cost associated with deleting all nonroots (u, v) of all ranks k is bounded by

$$\sum_{k=0}^{\infty} \frac{|H(u)| \cdot (k + 1)(k + 5)}{2^{k+1}}.$$

Recall that $\sum_{k=0}^{\infty} \frac{1}{2^k} = \sum_{k=0}^{\infty} \frac{k}{2^k} = 2$ and $\sum_{k=0}^{\infty} \frac{k^2}{2^k} = 6$. Hence the above sum is at most $|H(u)|(6 + 12 + 10)/2 = 14|H(u)|$. We pay for this by taking 14 credits from each element of $H(u)$, assuming $C \geq 14$ in Credit Invariant (i).

Next consider a rank k root (u, v). The elements in the binomial tree of (u, v) now have a total of $k+4+(C-14)2^k$ credits by Credit Invariant (i)–(ii). Choosing $C = 18$ makes this quantity at least $(k + 1)(k + 5)$, because $k^2 + 5k + 1 \leq 2^{k+2}$ for every $k \geq 0$. (This inequality follows by induction using base case $k \leq 1$. For the inductive step we use the identity $2k + 6 \leq 2^{k+2}$ for every $k \geq 1$.) Hence we can pay the cost associated with (u, v).

Theorem 1. *The dynamic vertex minimum problem can be solved in amortized time $O(1)$ for each* `Add_vertex`*,* `Add_edge` *and* `Delete_vertex` *operation and worst-case time $O(n)$ for each* `Find_min` *when the graph contains exactly n vertices.*

We close with three remarks. First, the constant C in the analysis can be lowered because the algorithm performs unnecessary comparisons. For instance in `Delete_vertex`(u), various executions of `Activate` may update $s_v(\cdot)$ values for the same vertex v. But only one update is sufficient.

Second, an actual implementation of this data structure will probably be more efficient if we modify the DVMP Invariants slightly. Invariant (i) can be changed to $rank(e) \leq c_1 \cdot rank(twin(e)) + c_2$, for constants c_1, c_2. This allows DVMP Invariant (ii) to be relaxed so there are fewer activates and deactivates. This speeds up the algorithm in practice. Theorem 1 remains valid.

Finally, if necessary we can ensure that the space is always $O(m)$, for m the current number of edges. The idea is to reconstruct the data structure whenever there are too many marked elements.

References

1. Arya, S., Ramesh, H.: A 2.5-factor approximation algorithm for the k-MST problem. Inf. Proc. Letters **65** (1998) 117–118
2. Agrawal, A., Klein, P., Ravi, R.: When trees collide: An approximation algorithm for the generalized Steiner problem on networks. SIAM J. Comp. **24** (1995) 440–456
3. Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P., Sudan, M.: The minimum latency problem. Proc. 26th Annual ACM Symp. on Theory Comput. (1994) 163–171
4. Blum, A., Ravi, R., Vempala, S.: A constant-factor approximation algorithm for the k-MST problem. J. Comp. Sys. Sci. **58** (1999) 101–108
5. Cole, R., Hariharan, R., Lewenstein, M., Porat, E.: A faster implementation of the Goemans-Williamson clustering algorithm. Proc. 12th Annual ACM-SIAM Symp. on Disc. Alg. (2001) 17–25
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. McGraw-Hill, NY (2001)
7. Garg, N.: A 3-approximation for the minimum tree spanning k vertices. Proc. 37th Annual Symp. on Foundations Comp. Sci. (1996) 302–309
8. Goemans, M., Goldberg, A., Plotkin, S., Shmoys, D., Tardos, E., Williamson, D.: Improved approximation algorithms for network design problems. Proc. 5th Annual ACM-SIAM Symp. on Disc. Alg. (1994) 223–232
9. Gabow, H.N., Goemans, M.X., Williamson, D.P.: An efficient approximation algorithm for the survivable network design problem. Math. Programming B **82** (1998) 13–40
10. Goemans, M.X., Kleinberg, J.: An improved approximation ratio for the minimum latency problem. Proc. 7th Annual ACM-SIAM Symp. on Disc. Alg. (1996) 152–158
11. Gabow, H.N., Pettie, S.: The dynamic vertex minimum problem and its application to clustering-type approximation algorithms. Univ. of Colorado TR# CU-CS-928-02 (2002)
12. Goemans M.X., Williamson, D.P.: A general approximation technique for constrained forest problems. SIAM J. Comp. **24** (1995) 296–317
13. Goemans M.X., Williamson, D.P.: The primal-dual method for approximation algorithms and its application to network design problems. in Approximation Algorithms for NP-hard Problems, D.S. Hochbaum (ed.) (1997) 144–191
14. Johnson, D.S., Minkoff, M., Phillips, S.: The prize collecting Steiner tree problem: Theory and practice. Proc. 11th Annual ACM-SIAM Symp. on Disc. Alg. (2000) 760–769
15. Klein, P.: A data structure for bicategories, with application to speeding up an approximation algorithm. Inf. Proc. Letters **52** (1994) 303–307
16. Sedgewick, R.: Algorithms in C++. 3rd edn. Addison-Wesley, Reading, MA (1998)
17. Tarjan, R.E.: Amortized computational complexity. SIAM J. on Algebraic and Discrete Methods **6** (1985) 306–318
18. Williamson, D.P., Goemans M.X., Mihail, M., Vazirani, V.V.: An approximation algorithm for general graph connectivity problems. Combinatorica **15** (1995) 435–454

A Polynomial Time Algorithm to Find the Minimum Cycle Basis of a Regular Matroid

Alexander Golynski and Joseph D. Horton

Faculty of Computer Science, University of New Brunswick
P.O. Box 4400, Fredericton, New Brunswick, Canada E3B 5A3
golynski@unb.ca, jdh@unb.ca

Abstract. An algorithm is given to solve the minimum cycle basis problem for regular matroids. The result is based upon Seymour's decomposition theorem for regular matroids; the Gomory-Hu tree, which is essentially the solution for cographic matroids; and the corresponding result for graphs. The complexity of the algorithm is $O((n+m)^4)$, provided that a regular matroid is represented as a binary $n \times m$ matrix. The complexity decreases to $O((n+m)^{3.376})$ using fast matrix multiplication.

1 Introduction

Coleman and Pothen study the sparse null space basis problem [CP86]. Given an $n \times m$ matrix A, find a matrix N with fewest non-zero entries, whose columns span the null space of A. Their purpose is to develop a practical algorithm for the linear equality problem, which is a fundamental concern in numerical optimization. Minimize a nonlinear objective function f subject to linear constraints $Ax = b$, assuming that $f\colon \Re^n \to \Re$ is a twice continuously differentiable function.

When A is a totally unimodular matrix, the sparsest null basis problem reduces to finding the minimum cycle basis of the regular matroid represented by A. A cycle of the matroid corresponds to the support of a solution to the system $Ax = b$. A totally unimodular matrix is one in which every square submatrix has determinant 0 or ± 1. Such matrices are exactly those which represent regular matroids [Tut65]. The minimum cycle basis problem (MCB) is: given a binary matroid M with nonnegative weights assigned to its elements, what is the set of circuits with the smallest total weight which generate all of the cycles of the matroid? The null space problem is the unweighted special case of the MCB problem. However the MCB problem applies to binary matroids only.

In this paper we solve the MCB problem for regular matroids in polynomial time. The method involves several results from the literature. Seymour [Sey80] proves that any regular matroid can be decomposed in polynomial time into 1-sums, 2-sums, and 3-sums of graphic matroids, cographic matroids and copies of the special ten element matroid R_{10}. Truemper [Tru90] gives a cubic algorithm to find such a decomposition. The MCB problem for graphs is solved in [Hor87]. The Gomory-Hu tree [GH61] algorithm solves the MCB problem for cographic matroids. We show how the minimum cycle bases of the parts of a decomposition can be glued together to form a minimum cycle basis of a k-sum.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 200–209, 2002.

The problem of extracting the minimum base of a matroid represented by a binary matrix arises in the algorithm in [Hor87] and in the gluing algorithm here. This minimum base (MB) problem can be solved faster than straightforward Gaussian elimination by using divide and conquer, and fast matrix multiplication. Moreover, it is proved in [Gol02] that the matrix multiplication and minimum base problems have almost the same asymptotic complexity. The complexity of the MCB algorithm for graphs is reduced to $O(nm^{2.376})$ from $O(nm^3)$, where n is the number of vertices in the graph and m is the number of edges.

2 Background

Let M be a binary matroid, S be its ground set and $\mathcal{C}$ be its cycle space. The elements of M are weighted by a nonnegative function defined on its ground set S, that is $w\colon S \to \Re^+ \cup \{0\}$. The weighting is linearly propagated to subsets of S and to families of subsets of S. A *minimum cycle basis* is a basis of the cycle space that has the minimum weight among all the bases. The following lemma characterizes minimum cycle bases.

Lemma 1. *Suppose that $\mathcal{B}$ is a cycle basis of a binary matroid M. Then $\mathcal{B}$ is a minimum cycle basis if and only if every cycle $C \in \mathcal{C}(M)$ has the representation*

$$C = C_1 + C_2 + \ldots + C_k \tag{1}$$

such that for every i, $w(C_i) \leq w(C)$ and $C_i \in \mathcal{B}$

A simple consequence is that a cycle is not in the minimum cycle basis if it can be written as the sum of smaller cycles.

2.1 Graphic and Cographic Matroids

For a graphic matroid M, let $G(M)$ denote a graph representing M. For a cographic matroid M, the space of cuts of $G(M^*)$ and the cycle space of M coincide.

The algorithm to find the minimum cycle basis for a weighted graph in [Hor87] is based on the following:

Lemma 2. *Let C be a circuit in a minimum cycle basis of a graph G, and let x be a vertex of C. Then there is an edge $e = \{u, v\} \in C$ such that C consists of a shortest path from u to x, a shortest path from v to x and the edge e.*

The following algorithm finds the minimum cycle basis:

1. Find the shortest path P_{xy} in G between each pair of vertices x and y.
2. List all the *candidate* circuits of the form

$$\mathcal{B}_t = \{P_{uw} + P_{vw} + \{u, v\} | \{u, v\} \in E, w \in V, P_{uw} \cap P_{vw} = \{w\}\}$$

3. Use a greedy algorithm to extract a minimum cycle basis from $\mathcal{B}_t$.

The last step in [Hor87] is based upon Gaussian elimination, and is improved in Section 4.1.

To find the minimum cycle basis of a cographic matroid, one can construct the Gomory-Hu tree [GH61], for which the best algorithm we know is in [GR98]. Given a weighted graph, the Gomory-Hu tree is a weighted tree on the same set of vertices for which the weights of the minimum cuts between any pair of nodes are the same in both the tree and the graph. Particularly, each edge of this Gomory-Hu tree corresponds to the minimum cut in the graph separating its two endpoints. That this solves the minimal cocycle space basis problem is easy to prove and has been known to the second author for many years, but we do not know any reference in the literature to this fact.

Let $G = (V, E, w)$ be a weighted graph and $T = (V, E_T, w_T)$ be a weighted tree. We use the notation $C_G(A, V \setminus A)$ for the cut separating the vertex sets A and $V \setminus A$ in a graph G. Every edge $e = \{s, t\}$ of T forms a cut in T, say $C_T(A_e, V \setminus A_e)$ for some set $A \subset V$. Consider the corresponding cut $C_e = C_G(A_e, V \setminus A_e)$ in G. Then T is a Gomory-Hu tree for G if C_e is a minimum $s - t$ cut in G for every $e \in E_T$.

The family of cuts $\mathcal{B} = \{C_e : e \in E_T\}$ forms a basis in the cut space of G. There are $n - 1$ cuts corresponding to the edges of the Gomory-Hu tree, which is exactly the number of dimensions of the cut space. If for some $e = \{s, t\}$, C_e is written as the sum of other cuts in the family, the sum must contain a cut which separates s and t. Since no other cut in $\mathcal{B}$ separates s and t, except for C_e, we conclude that $\mathcal{B}$ is an independent family of cuts. Therefore it forms a cut basis. This gives a method to find the representation of any cut C over $\mathcal{B}$, namely C_e $(e = \{s, t\})$ is in the representation if and only if C separates s and t. On other hand, note that C can not be lighter than any such C_e, since both C and C_e separate s and t, but C_e is a minimum cut. By Lemma 2, $\mathcal{B}$ forms a minimum cycle basis for the cographic matroid of G.

2.2 Separations and Sums

A partition (X, Y) of the ground set S of a matroid M is called a *k-separation* if $r(X) + r(Y) = r(S) - k + 1$. If the condition that $|X|, |Y| \geq 2^{k-1}$ is added for binary matroids, then we can express M as the *k-sum* of some extensions of $M \setminus X$ and $M \setminus Y$ (the symbol $M \setminus X$ represents the deletion of the elements of Y from the matroid M), such that both of the extensions are smaller than M. In [Tru90] such a separation is called a $(k|2^{k-1})$-separation.

The sum of two binary matroids $M_1 \oplus M_2$ is defined to be a matroid M on the ground set $S = S_1 + S_2$, where $+$ stands for the symmetric difference operation. S_1 and S_2 are the ground sets of M_1 and M_2 respectively. The sum is given by its cycle space

$$\mathcal{C}(M) = \{C_1 + C_2 : C_1 \in \mathcal{C}(M_1), C_2 \in \mathcal{C}(M_2), C_1 + C_2 \subseteq S\} \tag{2}$$

The members of $Z = S_1 \cap S_2$ are called the *connecting* elements. We require:

1. Z contains no cocircuit of either M_1 or M_2;
2. Z contains no parallel elements; and
3. $|S_1| < |S|$, $|S_2| < |S|$.

We are concerned about three special cases of this operation, namely:

1-**sum** (or direct sum) when $Z = \emptyset$;
2-**sum** when $Z = \{z\}$;
3-**sum** when $|Z| = 3$, Z is a *triangle* (circuit of three elements) of both M_1 and M_2.

There is a natural way to construct a sum, if a k-separation $S = X \cup Y$ of a matroid M is given. Consider the subspaces of the cycle space $\mathcal{C}$ of M, $V_X = \{C \in \mathcal{C} | C \subset X\}$ and $V_Y = \{C \in \mathcal{C} | C \subset X\}$, and consider the factor space

$$U = \mathcal{C}/(V_X \oplus V_Y) \tag{3}$$

The symbol $\oplus$ denotes direct sum of linear spaces. Let $\sim$ denote the corresponding equivalence relation on $\mathcal{C}$. One can think of the connecting elements in a matroid sum as the non-zero elements of this factor space. Let $Z = U \setminus \{0\}$. The dimension of U is

$$\begin{aligned} \dim(U) &= \dim(\mathcal{C}) - \dim(V_X) - \dim(V_Y) \\ &= |S| - r(S) - |X| + r(X) - |Y| + r(Y) = k - 1 \end{aligned}$$

Let E denote an extended matrix representation of M, the columns of which are labeled with the elements of S. Let v_x denote the column of E labeled with x. For each $z \in Z$, define

$$v_z := \sum_{x \in C_z \cap X} v_x \tag{4}$$

where C_z is a representative of the equivalence class z in $\mathcal{C}$. The vector v_z is well defined, since for any other representative C'_z the cycle $C_z + C'_z$ intersects X in another cycle C''. Thus

$$\sum_{x \in C_z \cap X} v_x + \sum_{x \in C'_z \cap X} v_x = \sum_{x \in C''} v_z = 0 \tag{5}$$

Let M_1 be a matroid on the ground set $X \cup Z$. Construct the extended matrix representation of M_1 by taking the old vectors $[v_x | x \in X]$ and adjoining the newly constructed vectors $[v_z | z \in Z]$. Similarly construct the matroid M_2.

To prove that $M = M_1 \oplus M_2$, take any cycle $C \in \mathcal{C}$. Let z be its equivalence class in (3). If $z = 0$, then C is decomposable into two cycles, one in V_X and one in V_Y. These cycles have to be in $\mathcal{C}(M_1)$ and $\mathcal{C}(M_2)$ respectively, and hence they are in the sum $M_1 \oplus M_2$ as well. If $z \neq 0$, then $C_X = (C \cap X) \cup \{z\}$ is a cycle of M_1 and $C_Y = (C \cap Y) \cup \{z\}$ is a cycle of M_2. Thus $C = C_X + C_Y$ is a cycle of the sum $M_1 \oplus M_2$.

Note that none of the connecting elements can be parallel to each other, since all of the v_z are distinct. The set of connecting elements cannot contain a cocircuit, since the deletion of all the connecting elements does not change the rank. We call the resulting matroid sum $M = M_1 \oplus M_2$ a *k-delta sum*.

A set A of M_1 or M_2 is called *good* with respect to the decomposition $M = M_1 \oplus M_2$ if $|A \cap Z| \leq 1$, otherwise A is called *bad*. Note that in the 1-sum and 2-sum cases, all sets are good. In the k-delta sum case every cycle $C \in \mathcal{C}(M)$ is the sum of two good cycles. Recall that if $C \sim 0$, then $C_X = C \cap X$ and $C_Y = C \cap Y$. If $C \sim z \neq 0$, then $C_X = (C \cap X) \cup \{z\}$ and $C_Y = (C \cap Y) \cup \{z\}$.

2.3 The Weighted Case

We now show how to use the previous construction with weighted matroids. Let M be a weighted binary matroid, $S = X \cup Y$ be a k-separation of M, $M = M_1 \oplus M_2$ be the corresponding k-delta sum. Define $w_1(x) = w(x)$, if $x \in S_1 \setminus S_2$ and $w_2(x) = w(x)$, if $x \in S_2 \setminus S_1$. For each $z \in Z$, call a lightest cycle equivalent to z a *z-round* (there are could be many z-rounds), and denote one of them by C_z. The set $C \cap X$ is called a *z-way* in M_1 (denoted by P_z) if C is a z-round, and the set $C \cap Y$ is called a z-way in M_2 (denoted by Q_z). Define $w_1(z) = w(Q_z)$ and $w_2(z) = w(P_z)$.

Let C be any cycle of $\mathcal{C}$. If $C \sim 0$, then $w(C) = w_1(C \cap X) + w_2(C \cap Y)$. If not, then let $C \sim z \neq 0$, the set $C \cap X$ (M_1-part of C) cannot be lighter than P_z, since the cycle $(C \cap X) \cup Q_z \sim z$ cannot be lighter than $C_z = P_z \cup Q_z$. Similarly, $w(C \cap Y) \geq w(Q_z) = w_1(z)$ (Q_z is a shortest cycle through z minus z). Thus if C is represented by the sum of two good cycles C_1 and C_2, then $w_1(C_1) = w_1(C \cap X) + w_1(z) \leq w(C)$ and similarly for $w_2(C_2)$. Thus every cycle $C \in \mathcal{C}$ is the unique sum of two good cycles C_1 and C_2, which are not heavier than C.

Lemma 3. *The triangle inequality holds for every circuit of three connecting elements $T = \{z_1, z_2, z_3\} \subseteq Z$, i.e. $w_i(z_1) \leq w_i(z_2) + w_i(z_3)$ for $i \in \{1, 2\}$.*

Proof. Let Q_1, Q_2, Q_3 be the corresponding z-ways. Since $z_1 + Q_2 + Q_3$ is a cycle through z_1, then $w(Q_2) + w(Q_3) \geq w(Q_1)$, i.e. $w_1(z_1) \leq w_1(z_2) + w_1(z_3)$.

Once all the z-rounds of the sum $M_1 \oplus M_2$ are fixed, define the *correspondent* of a set A of M_1 to be

$$c_1(A) = A + \sum_{z \in A \cap Z} (z + Q_z) \tag{6}$$

Similarly define the correspondent function c_2 on subsets of M_2. The connecting elements of a set are replaced in the correspondent by the corresponding z-ways of the other side of the separation. The inequality $w(c_1(A)) \leq w_1(A)$ holds, since $w_1(z) = w(Q_z)$, and $w(A + B) \leq w(A) + w(B)$ for every A and B. Note that $w(c_1(A)) = w_1(A)$, if A is good.

The following theorem is used to find the minimum cycle basis of a sum. It is assumed that minimum cycle bases of parts of the sum and the corresponding z-rounds are known.

Theorem 1. *Let $M = M_1 \oplus M_2$ be the sum of binary weighted matroids. Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be minimum cycle bases of M_1 and M_2 respectively. Let C_z be a fixed set of z-rounds. Then the following set contains a minimum cycle basis of M.*

$$\mathcal{B}_t = c_1(\mathcal{B}_1) \cup c_2(\mathcal{B}_2) \cup (\cup_z C_z) \tag{7}$$

Proof. Proof omitted.

The cardinality of $\mathcal{B}_t$ can be bounded.

$$\begin{aligned} |\mathcal{B}_t| &\leq |\mathcal{B}_1| + |\mathcal{B}_2| + |Z| = |S_1| - r_1(S_1) + |S_2| - r_2(S_2) + |Z| \\ &= |X| + |Z| - r(X) + |Y| + |Z| - r(Y) + |Z| \\ &= |S| - r(S) + 3|Z| - k + 1 \leq |S| - r(S) + 3 \cdot 2^{k-1} - k - 2 \end{aligned} \tag{8}$$

Therefore there are at most $3 \cdot 2^{k-1} - k - 2$ redundant cycles in $\mathcal{B}_t$.

The third term $\cup_z C_z$ in equation (7) can be omitted. In the case of 1-sums and 2-sums there are no redundant cycles, while in the case of a 3-sum there is at most one redundant cycle [Gol02].

3 The Algorithm

The first phase of the algorithm is based on the decomposition theorem [Sey80] for regular matroids. A regular matroid M can be decomposed into 1-sums, 2-sums, and 3-sums of graphic matroids, cographic matroids and isomorphic copies of the special ten element matroid R_{10}. Call them *atom* matroids. The algorithm of [Tru90] finds such a decomposition in time cubic in the number of elements. Actually, Truemper finds a decomposition using a slightly different type of 3-sums, but these sums can be converted into 3-delta sums (or *delta sums* in the terminology of [Tru92]) in time that is linear in the rank of the matroid M_1.

Using results from the Section 2.3, proceed as follows:

1. Invoke Truemper's decomposition algorithm. Modify the corresponding decompositions, so that they became delta sum decompositions.
2. For each atom M that is not isomorphic to R_{10}, obtain the corresponding graphs $G(M)$ and cographs $G(M^*)$. These graphs are side results of Truemper's algorithm.
3. Find a minimum cycle basis for each $G(M)$ and $G(M^*)$ obtained in the previous step.
4. For each decomposition encountered in the first step (in bottom up order),
   a) find the corresponding z-ways,
   b) use a greedy algorithm to extract a minimum cycle basis from the set $\mathcal{B}_t$ in (7).

It remains to describe how to perform Steps 4a and 4b. Step 4b can be implemented using Gaussian elimination, but it can be improved asymptotically using fast matrix multiplication, see Section 4.1.

3.1 z-Ways

Finding the z-rounds of a decomposition can be reduced to the following more general *shrinking problem*. Given a weighted matroid M defined on a ground set S with a weight function w and a subset T of S, for each t in T, find a shortest circuit C_t in $\mathcal{C}(M)$ that intersects T only in t.

Let $M = M_1 \oplus M_2$ be a decomposition, and C_z, D_z be solutions of the shrinking problem for each $z \in Z$ in M_1 and M_2 respectively ($T = Z$). Then $C_z + D_z$ is a z-round for $M_1 \oplus M_2$. For our purposes it suffices to restrict M to being a regular matroid and the set T to being a single edge or a triangle (for 2-sums and 3-sums respectively). We use the modified decomposition found in the Step 1 for solving the shrinking problem. Note that the output of the shrinking problem does not depend on the weights of the elements of T, so it is enough for w to be defined on $S \setminus T$ only.

The shrinking problem can be solved in an atom matroid. For the graphic case, delete the elements of T from the graph and solve the shortest path problem between the endpoints of the edge t. The path together with the edge of T is the answer. For the cographic case, the edges of T must be a simple cut in the graph and cannot contain a (graphic) circuit. Contract the edges of T (except for t) and solve the min-cut problem between the endpoints of t. All possible circuits can be considered for R_{10}.

Otherwise there is a decomposition $M = M_1 \oplus M_2$ and we can apply recursion. Let Z be the set of connecting elements. Weight the elements in M_1 and M_2, other than the elements of T and Z, the same as in M. If the sum is a 1-sum ($Z = \emptyset$), then each circuit of M is either a circuit of M_1 or a circuit of M_2. The problem can be solved in either M_1 or M_2, depending upon which submatroid the elements of T are in. In this case, if $|T| = 3$, it can not be partly in both M_1 and M_2, since T is a circuit.

The cases of a 2-sum or a 3-sum are considered together. Suppose that T is contained in one of the S_1 or S_2, say in S_1. Then all the elements e of M_2 have defined weights $w_2(e) = w(e)$ other than those in Z. Solve the shrinking problem for M_2 and Z, for each z in Z finding a z-way Q_z in M_2. For each connecting z in Z, assign weights $w_1(z) := w_2(Q_z)$. Next invoke the algorithm for the shrinking problem for M_1 and T to find a t-way P_t for each t in T.

Each way P_t contains at most one element from Z. If P_t contains two elements from Z, say z_1 and z_2, then there is a third element z_3, such that $Z = \{z_1, z_2, z_3\}$. Due to Lemma 3, $w_1(z_3) \leq w_1(z_1) + w_1(z_2)$. Thus z_3 can replace $\{z_1, z_2\}$ in P_t (in fact, the case where $w_1(z_3) < w_1(z_1) + w_1(z_2)$ is impossible, since $P_t + \{z_1, z_2, z_3\}$ can not be lighter than P_t). Thus the correspondent of $C_t = P_t \cup \{t\}$ (replacing z by Q_z), $c_1(C_t)$ is the solution to the shrinking problem in the case when T is only on one side of the decomposition.

Now suppose that T intersects both ground sets. Without loss of generality, let $T \cap S_1 = \{r, s\}$ and $T \cap S_2 = \{t\}$. Let z denote the connecting element that is equivalent to T in (3). This z is parallel to t in M_2. Modify the decomposition $M = M_1 \oplus M_2$ by deleting element t from M_2 and extending M_1 by t, such that t is parallel to z in M_1. Denote the modified matroids by M_1' and M_2'. It can

be checked that $M = M_1' \oplus M_2'$. Instead of deleting t from M_2, a sufficiently big weight can be assigned to t. To avoid the undesirable extension of M_1, we can decompose M_1' further using the 2-separation $(S_1 \setminus \{z\}) \cup (\{z, t\})$. Then $M_1' = M_1'' \oplus_2 R$, where the matroid M_1'' is isomorphic to M_1 and the matroid R consists of three parallel elements. The same method as in the case $T \subset S_1$ can be applied to solve the shrinking problem for the decomposition $M = M_1' \oplus M_2'$.

Let $S_1 \setminus \{z\} \cup \{z'\}$ be the ground set of M_1'' and $\{t, z, z'\}$ be the ground set of R. Let $\bar{P}_{z'}$, $\bar{P}_r$ and $\bar{P}_s$ be a solution of the shrinking problem for $\{z', r, s\}$ in M_1''. The t-way P_t is either contained within R and then it is $Q_t = \{z\}$ or it hits both R and M_1'' and then it is the sum of $\{t, z'\}$ and $\bar{P}_{z'}$. If the r-way P_r lies within M_1'', then it coincides with $\bar{P}_r$. Otherwise it hits both R and M_1'', and then it is $\{z, z'\}$ plus $\bar{P}_s$ ($\{z', r, s\}$ form a triangle in M_1''). The s-way can be found similarly. At last, note that the shrinking problem for $\{z', r, s\}$ in M_1'' is the same as the shrinking problem for $\{z, r, s\}$ in M_1, since M_1'' and M_1 are isomorphic. Thus we do not need to do any special preprocessing in this case.

4 Complexity and Improvements

4.1 Minimum Base Problem

We use the following result from [Gol02] to improve the asymptotic complexities of Step 3 in the algorithm for the MCB problem in the graphic case (Section 2.1) and Step 4b in the minimum cycle basis algorithm.

Theorem 2. *Let M be a weighted linear matroid given by an $n \times m$ standard or extended matrix representation. A minimum basis of M can be found in $O(nm \min\{n, m\}^{\omega-2})$ time, where ω is the best exponent for matrix multiplication.*

It is presently known [CW87] that ω is less than 2.376. Suppose that a standard matrix representation A for a matroid M is given. Let X be the set of row labels (current base) and Y is the set of column labels (current cobase). Our goal is to find a minimum base B and the standard matrix representing M with respect to this base. Assume that $|X| < |Y|$. Split Y into even parts Y_1 and Y_2. The algorithm uses recursion on the first part Y_1 to obtain a minimum base B_1 of $M \setminus Y_2$. Let X_1 be the set of elements which left the base X during the recursion, i.e. $X_1 = X \setminus B_1$, and let Y_{11} be the set of their replacements $Y_{11} = Y \cap B_1$. Also define $X_2 = X \setminus X_1$ and $Y_{12} = Y_1 \setminus Y_{11}$. The sets (X_1, X_2) and $(Y_{11}, Y_{12} \cup Y_2)$ give a partition of the original matrix A into four submatrices.

$$A = \begin{array}{c|c:c} & Y_{11} & Y_{12} \cup Y_2 \\ \hline X_1 & F & G \\ \hdashline X_2 & H & J \end{array} \tag{9}$$

After exchanging Y_{11} and X_1 in the current base the matrix A transforms to

$$\bar{A} = \begin{array}{c|c:c} & X_1 & Y_{12} \cup Y_2 \\ \hline Y_{11} & \bar{F} & \bar{G} \\ \hdashline X_2 & \bar{H} & \bar{J} \end{array} = \begin{array}{c|c:c} & X_1 & Y_{12} \cup Y_2 \\ \hline Y_{11} & F^{-1} & F^{-1}G \\ \hdashline X_2 & HF^{-1} & J - HF^{-1}G \end{array} \tag{10}$$

This operation is called a *group pivot* with respect to the pair (X_1, Y_{11}). Fast matrix multiplication can be used to speed up the group pivot. Then the algorithm is applied recursively on the right side (columns labeled with Y_2) of the matrix $\bar{A}$ to solve the original problem.

We can solve the minimum base problem for a matroid represented by an extended matrix representation introducing a fake basis consisting of heavy elements. The analysis of the algorithm [Gol02] shows that if square matrix multiplication can be done in $O(n^\omega)$ time, then the MB algorithm has the complexity $O(nm \min\{n, m\}^{\omega-2})$. Conversely, it can be shown that the MB problem cannot be solved faster than matrix multiplication by considering the simple linear matroid

$$\begin{array}{c|c:c} & Y_1 & X_2 \\ \hline X_1 & I & A \\ \hdashline Y_2 & B & 0 \end{array} \quad \rightarrow \quad \begin{array}{c|c:c} & X_1 & X_2 \\ \hline Y_1 & I & A \\ \hdashline Y_2 & B & -AB \end{array}$$

with the weight 0 assigned to the elements $Y = Y_1 \cup Y_2$ and the weight 1 assigned to the elements $X = X_1 \cup X_2$. Note that the transformed matrix contains the product AB. Thus matrix multiplication can be simulated by the MB algorithm.

4.2 Analysis of the Minimum Cycle Basis Algorithm

This result improves the time complexity of the MCB algorithm for graphs (Section 2.1) to $O(nm^\omega)$, where n is the number of nodes and m is the number of edges in the input graph. Step 4b in the MCB algorithm works in $O(|\mathcal{B}_t|m^{\omega-1})$ time using this result. Since $\mathcal{B}_t$ contains at most a constant number of redundant cycles in (8), it follows that $|\mathcal{B}_t| = O(m)$. Thus Step 4b runs in $O(m^\omega)$ time.

Let m be the number of elements in a regular matroid M. Truemper finds [Tru90] a complete decomposition in down to only atom matroids (graphic, cographic and R_{10}) in $O(m^3)$ time. Imagine a decomposition tree, the root of which is the original matroid; the descendants of a node are the matroids the node is decomposed into by Truemper's algorithm; and the leaves are atom matroids. There are at most $O(m)$ nodes in the tree as the children of a node always have fewer elements, and at most six more elements in total. Modifying the decomposition (from Truemper's 3-sum to our delta sum) requires at most $O(r)$ time for a matroid of the rank r. The modification process takes at most $O(mr)$ time.

Solving one shrinking problem at a given node M amounts to $O(1)$ work at each internal node below M in the decomposition tree. The graphic and cographic cases require solving $O(1)$ shortest path or network flow problems respectively. For the graphic case, we can use Dijkstra's algorithm, which has $O(m \log(m))$ complexity; for the cographic case, we use the algorithm in [GR98], the time bound for which is $O(m^{3/2})$. We need to solve at most $O(m)$ shrinking problems. Thus the total time for shrinking is $O(m^{5/2})$.

The Gomory-Hu tree for a graph can be constructed by solving $n-1$ network flow problems, where $n = O(m)$ is the number of vertices in the graph. The time complexity is $O(m^{5/2})$. The MCB algorithm for graphs, after the improvement of its last step runs in $O(nm^{\omega})$. Thus we need to spend $O(m^{\omega+1})$ time solving minimum cycle bases problems of the leaves ($O(m)$ elements total).

The time needed to glue the solutions at the leaf M of the cardinality m is $O(m^{\omega})$. We perform the gluing step at most $O(m)$ times. Thus gluing takes $O(m^{\omega+1})$. The overall complexity of the algorithm is $O(m^{\omega+1})$.

The more general MCB problem for binary matroids is known to be NP-hard. Indeed just the problem of finding a shortest circuit in a binary matroid is NP-hard, and a shortest circuit must be in any minimum cycle basis.

References

[CP86] T. Coleman and A. Pothen. The null space problem I. Complexity. *SIAM Journal of Algebraic Discrete Methods*, 7, 1986.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, 1987.

[GH61] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[Gol02] A. Golynski. A polynomial time algorithm to find the minimum cycle basis of a regular matroid. Master's thesis, University of New Brunswick, 2002.

[GR98] A. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783 – 797, 1998.

[Hor87] J. D. Horton. A polynomial-time algorithm to find the shortest cycle basis of a graph. *SIAM Journal on Computing*, 16(2):358–366, 1 1987.

[Sey80] P. Seymour. Decomposition of regular matroids. *Journal of Combinatorial Theory (B)*, 1980.

[Tru90] K. Truemper. A decomposition theory for matroids.V. Testing of matrix total unimodularity. *Journal of Combinatorial Theory (B)*, 1990.

[Tru92] K. Truemper. *Matroid Decomposition*. Academic Press, Boston, 1992.

[Tut65] W. Tutte. Lectures on matroids. *J. Res. Nat. Bur. Standards Sect. B*, 69B:1–47, 1965.

Approximation Algorithms for Edge-Dilation k-Center Problems*

Jochen Könemann, Yanjun Li, Ojas Parekh, and Amitabh Sinha

Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA.
{jochen,yanjun,odp,asinha}@andrew.cmu.edu

Abstract. In an ideal point-to-point network, any node would simply choose a path of minimum latency to send packets to any other node; however, the distributed nature and the increasing size of modern communication networks may render such a solution infeasible, as it requires each node to store global information concerning the network. Thus it may be desirable to endow only a small subset of the nodes with global routing capabilites, which gives rise to the following graph-theoretic problem.

Given an undirected graph $G = (V, E)$, a metric l on the edges, and an integer k, a k-center is a set $\Pi \subseteq V$ of size k and an assignment π_v that maps each node to a unique element in Π. We let $\mathtt{d}_\pi(u, v)$ denote the length of the shortest path from u to v passing through π_u and π_v and let $\mathtt{d}_l(u, v)$ be the length of the shortest u, v-path in G. We then refer to $\mathtt{d}_\pi(u, v)/\mathtt{d}_l(u, v)$ as the *stretch* of the pair (u, v). We let the stretch of a k-center solution Π be the maximum stretch of any pair of nodes $u, v \in V$. The minimum edge-dilation k-center problem is that of finding a k-center of minimum stretch.

We obtain combinatorial approximation algorithms with constant factor performance guarantees for this problem and variants in which the centers are capacitated or nodes may be assigned to more than one center. We also show that there can be no $5/4 - \epsilon$ approximation for any $\epsilon > 0$ unless $\mathcal{P} = \mathcal{NP}$.

1 Introduction

In this paper we consider the following graph-theoretic problem: we are given an undirected edge-weighted graph $G = (V, E, l)$, (l is the (metric) edge-weight function), and a parameter $k > 0$. We want to find a set $\Pi \subseteq V$ of k center nodes and assign each node $v \in V$ to a unique center $\pi_v \in \Pi$.

Let the *center distance* between nodes $u, v \in V$ be defined as

$$\mathtt{d}_\pi(u, v) = \mathtt{d}_l(u, \pi_u) + \mathtt{d}_l(\pi_u, \pi_v) + \mathtt{d}_l(\pi_v, v)$$

where $\mathtt{d}_l(u, v)$ denotes the shortest path distance between nodes u and v. The *stretch* for a pair of nodes $u, v \in V$ is then defined as the ratio $\mathtt{d}_\pi(u, v)/\mathtt{d}_l(u, v)$

* This material is based upon work supported by the National Science Foundation under Grant No. 0105548.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 210–219, 2002.

of center distance and shortest path distance. We let the stretch of a solution $(\Pi, \{\pi_v\}_{v\in V})$ be the maximum stretch of any pair of nodes $u, v \in V$. The goal in the *minimum edge-dilation k-center problem* (MEDKC) is to find a set $\Pi \subseteq V$ of cardinality at most k and an assignment $\{\pi_v\}_{v\in V}$ of nodes to centers of minimum stretch.

A closely related problem is that of finding a *k-center* in a given graph $G = (V, E)$. Here, we want to find a set of nodes $C \subseteq V$ of cardinality at most k such that the maximum distance from any node to its closest center is as small as possible. This problem admits a 2-approximation in the undirected setting [2,5, 12] and it is well-known that there cannot exist a $2 - \epsilon$ approximation for any $\epsilon > 0$ unless $\mathcal{P}$=$\mathcal{NP}$ [6,11]. We adapt techniques used for the k-center problem to minimize the bottleneck stretch of any pair of nodes $u, v \in V$. Our main result is the following:

Theorem 1. *There is a polynomial-time algorithm that computes a feasible solution $\Pi \subseteq V$ to the* MEDKC *problem such that for every two vertices $u, v \in V$ we have $\mathtt{d}_\pi(u,v)/\mathtt{d}_l(u,v) \leq 4 \cdot \mathtt{opt} + 3$, where* opt *is the optimal stretch. On the negative side, no $5/4 - \epsilon$ approximation can exist for any $\epsilon > 0$ unless $\mathcal{P}$=$\mathcal{NP}$.*

The multi-MEDKC problem is a natural extension of the MEDKC problem. Here, each vertex is allowed to keep a set of centers $\pi_v \subseteq \Pi$. We redefine the center distance between nodes u and v as

$$\mathtt{d}^m_\pi(u,v) = \min_{\pi_1\in\pi_u, \pi_2\in\pi_v} \mathtt{d}_l(u,\pi_1) + \mathtt{d}_l(\pi_1,\pi_2) + \mathtt{d}_l(\pi_2,v).$$

Again, the task is to find a set of center nodes $\Pi \subseteq V$ of cardinality at most k that minimizes the maximum stretch, now defined as $\mathtt{d}^m_\pi(u,v)/\mathtt{d}_l(u,v)$.

Theorem 2. *Given an undirected graph $G = (V, E)$ and a non-negative length function l on E, there is a polynomial-time algorithm that computes a solution Π to the* multi-MEDKC *problem such that for every two vertices $u, v \in V$ we have $\mathtt{d}^m_\pi(u,v)/\mathtt{d}_l(u,v) \leq 2 \cdot \mathtt{opt} + 1$.*

Subsequently, we extend the result in Theorem 1 to a natural capacitated version of the MEDKC problem (denoted by C-MEDKC): each potential center location $v \in V$ has an associated capacity U_v. We now want to find a minimum-stretch center set $\Pi \subseteq V$ of size at most k and an assignment $\{\pi_v\}_{v\in V}$ of nodes to centers such that the set $\pi_i^{-1} = \{v \in V : \pi_v = i\}$ has size at most U_i for all $i \in \Pi$. We adapt facility location techniques from [13] in order to obtain the following bicriteria result:

Theorem 3. *Given an instance of the* C-MEDKC *problem, there is a polynomial-time algorithm that computes a center set $\Pi = \{\pi_1, \ldots, \pi_{2k}\}$ and an assignment of nodes to centers such that $|\pi_i^{-1}| \leq 2\,U_i$ for all $1 \leq i \leq 2k$. The stretch of the solution is at most $12 \cdot \mathtt{opt} + 1$ where* opt *is the stretch of an optimum solution which places no more than k centers and obeys all capacity constraints.*

The problem motivation comes from (distributed) *routing* in computer networks. Here, a host v keeps information about routing paths to each other host u locally in its *routing table*. The entry for node v in u's routing table consists of the next node on the routing path from node u to node v. Clearly, we can ensure shortest-path routing if we allow each node to store $O(n)$ entries in its routing table.

Considering the size of modern computer networks that often connect millions of nodes, we can hardly ask each node to store information for every other host in the network. For this reason, modern routing protocols like OSPF[9] allow a subdivision of a network into areas. Now, each node keeps an entry for every other node in the same area. Routing between nodes in different areas is done via a backbone network of area border routers that interconnects the areas.

We can formalize the above problem as follows: We allow each node to store up to $O(B)$ entries in its routing table, where B is a constant representing the memory available at each node. These are the nodes with which it can directly communicate. In addition, we install a supporting backbone network of k center nodes. Each node is allowed to keep an additional entry in its routing table for the center node π_v that it is assigned to. Whenever node v needs to compute a route to node u that is not in its routing table, it has to route via its center π_v. As before, we assume that routing among center nodes is along shortest paths.

The problem now is to place k center nodes and configure the routing tables of each of the nodes in V such that the maximum stretch of any path is minimum. We refer to this problem as `MEDKC` *with bounded routing table space* (`B-MEDKC`). We obtain the following theorem whose proof we defer to the full version of this paper [8] due to space limitations.

Theorem 4. *Given an instance of the* `B-MEDKC` *problem, we can find in polynomial time a center set* Π *and an assignment* $\{\pi_v\}_{v\in V}$ *of nodes to centers that achieves stretch* $O(\rho \cdot \texttt{opt})$ *where* `opt` *denotes the optimum stretch of any* `B-MEDKC` *solution and* ρ *is the performance guarantee of any algorithm for the* `MEDKC` *problem.*

We note that the last result is closely related to work on *compact routing schemes* (see [1] and the references therein). Cowen [1] shows that if we allow $O(n^{2/3}\log^{4/3} n)$ table space at each node, we can achieve a solution where the routing path between any pair of nodes $u, v \in V$ is at most three times as long as the shortest u, v-path in G. Notice that this contrasts our results since we are comparing the stretch that we achieve with the minimum possible stretch.

Finally, our problem is related to that of designing *graph spanners*. In the unweighted version, first considered in [10], we are given an undirected edge-weighted graph $G = (V, E)$. A subgraph $H = (V, E_H)$ of G is called an α-spanner if we have $\mathtt{d}_H(u, v) \leq \alpha\, \mathtt{d}_G(u, v)$ for every pair of nodes $u, v \in V$. The literature on spanners is vast and includes variants such as degree-bounded spanners, sparse spanners, additive graph spanners as well as hardness results (see [3] and the references therein).

2 Hardness

We first show that the basic MEDKC problem is $\mathcal{NP}$-hard. Hardness of the extensions follow because each of the extensions is a strict generalization of the basic problem.

Theorem 5. *The minimum edge-dilation k-center problem is $\mathcal{NP}$-hard. Furthermore, unless $\mathcal{NP} = \mathcal{P}$, there can be no $5/4 - \epsilon$ approximation for any $\epsilon > 0$.*

Proof. The proof is by reduction from *minimum vertex-dominating set* (MVDS). In MVDS we are given an undirected graph $G = (V, E)$ and we want to find a set $S \subseteq V$ of minimum cardinality such that for all $v \in V$, either $v \in S$ or there is a $u \in S$ such that $vu \in E$. This problem is known to be $\mathcal{NP}$-hard [4].

Suppose we are given an instance of the MVDS problem: $G_1 = (V_1, E_1)$. We construct an edge-weighted auxiliary graph $G = (V, E, l)$ from G_1. For each node $v \in V_1$, let V contain two copies v_1 and v_2, along with an edge v_1v_2 of length 1. For each edge $uv \in E_1$, we let u_1v_1 of length 1 be in E. Finally, we include edge u_1v_1 of length 2 in E if the shortest path between u and v in G_1 has at least 3 edges.

It is not difficult to see that if there exists a vertex dominating set in G_1 of cardinality at most k then the optimum stretch of the MEDKC instance given by G is at most 4 (locate the centers exactly at the positions of the vertex dominating set). Also, if there exists no vertex dominating set in G_1 with size less than or equal to k, then for any center set $\Pi \subseteq V$ of cardinality k we can always find a vertex $v^* \in V_1$ such that its copies v_1^* and v_2^* satisfy $\mathtt{d}_\pi(v_1^*, v_2^*)/\mathtt{d}_l(v_1^*, v_2^*) \geq 5$. □

3 The Basic MEDKC Problem

In this section, we prove Theorem 1. We first develop a combinatorial lower-bound and use it to compute an approximate solution to the MEDKC problem. We then give an algorithm that computes an approximate solution to the proposed lower bound.

3.1 A Lower-Bound: Covering Edges with Vertices

For each pair $u, v \in V$, consider the set

$$S_{uv}^{\alpha} = \{w \in V \,:\, \mathtt{d}_l(u, w) + \mathtt{d}_l(v, w) \leq \alpha \cdot \mathtt{d}_l(u, v)\}. \tag{1}$$

It is clear that any optimum solution Π to MEDKC needs to have at least one node from $S_{uv}^{\mathtt{opt}}$ for all pairs $u, v \in V$.

The *minimum-stretch vertex cover problem* (MSVC-α) for a given graph $G = (V, E, l)$ and a parameter $\alpha > 0$ is to find a set $C \subseteq V$ of minimum cardinality such that $S_{uv}^{\alpha} \cap C \neq \emptyset$ for all pairs $u, v \in V$. Let k_α denote the cardinality of an optimal solution to MSVC-α. The following lemma is immediate.

Lemma 1. *Suppose there is a solution with stretch α for a given instance of the* MEDKC *problem. Then $k_\alpha \leq k$.*

3.2 Computing an Approximate MEDKC Solution

Given an instance of MEDKC, we first compute the smallest α such that the associated MSVC-α instance has a solution of cardinality at most k.

Lemma 2. *Given an instance of* MEDKC*, let* opt *be the minimum possible stretch of any solution. We can then efficiently compute* $\alpha \leq 2\,\mathtt{opt} + 1$ *such that* $k_\alpha \leq k$.

Our algorithm to locate a set of center nodes $\Pi \subseteq V$ is now straightforward: Let α be as in Lemma 2 and let Π be a solution to the respective instance of MSVC-α. For each vertex $v \in V$, we assign v to the closest center in Π, i.e. $\pi_v = \text{argmin}_{u \in \Pi}\, \mathtt{d}_l(v, u)$.

Proof of Theorem 1. Let u and v be an arbitrary pair of vertices in V. We want to bound $\mathtt{d}_\pi(u, v)$. Let c_{uv} be the node that covers the pair u, v in the MSVC-α solution.

It follows from our choice of π_v and π_u that $\mathtt{d}_l(u, \pi_u) \leq \mathtt{d}_l(u, c_{uv})$ and $\mathtt{d}_l(v, \pi_v) \leq \mathtt{d}_l(v, c_{uv})$. Hence,

$$\mathtt{d}_\pi(u, v) \leq 2(\mathtt{d}_l(u, \pi_u) + \mathtt{d}_l(v, \pi_v)) + \mathtt{d}_l(u, v) \leq (2\alpha + 1)\mathtt{d}_l(u, v)$$

Using Lemma 2 we can bound $(2\alpha + 1)\mathtt{d}_l(u, v)$ by $4\,\mathtt{opt} + 3$. □

3.3 Solving MSVC-α

We now proceed by giving a proof of Lemma 2. We first show how to compute a solution APX to MSVC-$(2\alpha + 1)$ of cardinality at most k_α.

For a vertex v and a subset of the edges $\overline{E} \subseteq E$ define $l_v(\overline{E}) = \min_{e \in \overline{E}} l_e$ to be the minimum length of any edge $e \in \overline{E}$ that is incident to v. Also, let $S_\alpha^{-1}(\overline{E}, v) = \{e \in \overline{E} : v \in S_e^\alpha\}$ be the subset of edges in $\overline{E}$ that are covered by vertex $v \in V$.

In the following we let $\alpha' = 2\alpha + 1$ and we say that a set $C \subseteq V$ *covers* edge $e \in E$ if $S_e^{\alpha'} \cap C \neq \emptyset$. Our algorithm starts with $C = \emptyset$ and repeatedly adds vertices to C until all edges in the graph are covered. More formally, in iteration i, let the remaining uncovered set of edges be $\overline{E}$ and let $\overline{V} \subseteq V$ be the set of vertices that have positive degree in $\overline{E}$. Let $e_i \in \overline{E}$ be the shortest edge in $\overline{E}$. We then choose v_i as one of the endpoints of e_i. Subsequently we remove $S_{\alpha'}^{-1}(\overline{E}, v_i)$ from $\overline{E}$ and continue.

Lemma 3. *If the above algorithm terminates with a feasible solution* $C \subseteq V$ *for a given instance of* MSVC-$(2\alpha + 1)$ *then we must have* $k_\alpha \geq |C|$.

Proof. Assume for the sake of contradiction that there exists a set $C^* \subseteq V$ such that $|C^*| < |C|$ and for all $e \in E$, there exists $v_e \in S_e^\alpha \cap C^*$.

Recall the definition of v_i and e_i. There must exist $1 \leq i < j \leq |C|$ and a node $v \in C^*$ such that $v \in S_{e_i}^\alpha \cap S_{e_j}^\alpha$. In the following, refer to Figure 1. By definition, we must have $a + b \leq \alpha \cdot l_{e_i}$ and $e + f \leq \alpha \cdot l_{e_j}$. Using this along with triangle inequality yields $c + d \leq \alpha \cdot l_{e_i} + \alpha \cdot l_{e_j} + l_{e_i}$.

The right hand side of the last inequality is bounded by $(2\alpha + 1)l_{e_j}$ by our choice of v_i. This contradicts the fact that e_j remains uncovered in iteration i. □

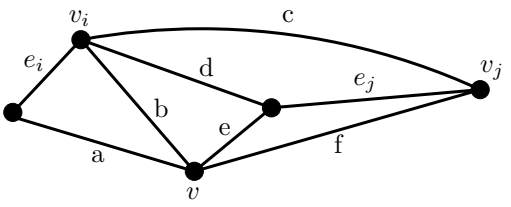


Fig. 1. Center $v \in C^*$ covers both e_i and e_j.

Proof of Lemma 2. The optimum stretch α^* of any instance has to be in the interval $[1, \texttt{diam}(G)]$. We can use binary search to find the largest α in this interval such that the above algorithm returns a solution of cardinality at most k. Our algorithm produces a solution with stretch $2\alpha + 1$ and it follows from Lemma 3 that $\alpha^* \leq \alpha$. □

4 Choosing among Many Centers – Multi-MEDKC

In the multi-MEDKC setting, we allow each node v to keep a set of center nodes $\pi_v \subseteq \Pi$. For each pair of nodes $u, v \in V$, we allow u and v to use the center nodes $\pi_u^v \in \pi_u$ and $\pi_v^u \in \pi_v$ that minimize

$$\mathtt{d}_l(u, \pi_u^v) + \mathtt{d}_l(\pi_u^v, \pi_v^u) + \mathtt{d}(\pi_v^u, v).$$

Notice that in an optimum solution, triangle inequality will always enforce $\pi_u^v = \pi_v^u$. Hence this problem has a solution with stretch α iff MSVC-α has a solution of cardinality at most k. This, together with Lemma 2, immediately yields Theorem 2.

A more interesting version of multi-MEDKC occurs when we restrict π_v for each node $v \in V$. For example, we might require that there is a global constant ρ such that every client node can only communicate with centers within distance ρ. We call this the *ρ-restricted* multi-MEDKC *problem.* We assume we are always given a "large enough" ρ, otherwise the problem is not meaningful.

We omit the proof of the following lemma since it is similar to that of Lemma 2.

Lemma 4. *Given an instance of ρ-restricted* multi-MEDKC*, let* opt *be its optimal stretch. We can then efficiently compute $\alpha \leq 2\,\texttt{opt} + 1$ such that $k_\alpha \leq k$.*

This shows that we can still use MSVC-α as a basis to construct a low-stretch center set. The following is again an immediate corollary of Theorem 1.

Corollary 1. *There is a polynomial time algorithm to solve the ρ-restricted* multi-MEDKC *problem that achieves a stretch of at most $4 \cdot$ opt $+ 3$.*

5 Capacitated Center Location

We now come to the capacitated version of the basic MEDKC problem. Here, we want to find a minimum-stretch center set (and assignment of nodes to centers) of cardinality at most k such that the number of nodes that are assigned to center i is no more than U_i, specified in the input.

5.1 A Modified Lower Bound

For each node v, we define $l_v = \min_{uv \in E} l_{uv}$. For a given stretch $\alpha \geq 1$ let S_v^α be the set of nodes whose distance from v is at most $\alpha \cdot l_v$, i.e. $S_v^\alpha = \{u \in V : \mathtt{d}_l(v,u) \leq \alpha \cdot l_v\}$. The optimum solution must have a center node in S_v^α in order to cover the shortest edge incident to v.

We now need to find a set of vertices $C \subseteq V$ of minimum cardinality such that $C \cap S_v^\alpha \neq \emptyset$ for all $v \in V$. Additionally, we require that each node v is assigned to exactly one center node π_v and that the sets $\pi_i^{-1} = \{u \in V : \pi_u = i\}$ have size at most U_i for all $i \in V$. Let this problem be denoted by $\mathtt{MSVC2} - \alpha$.

Lemma 5. *Suppose there is a solution with stretch α for a given instance of the* C-MEDKC *problem. Then there must be a solution C to the associated* $\mathtt{MSVC2} - \alpha$ *instance with $|C| \leq k$.*

We model the lower-bound by using an integer programming formulation. We then solve the LP relaxation of the model and round it to an integer solution using ideas from [13]. Finally, we prove that this solution yields a solution for the original instance of C-MEDKC with low stretch.

5.2 A Facility Location Type LP

In the IP, we have a binary indicator variable y_i for each $i \in V$ that has value 1 iff we place a center node at i. Additionally, we have variables x_{iv} that have value 1 iff $\pi_v = i$. The following IP formulation models $\mathtt{MSVC2} - \alpha$.

$$\min \sum_{i \in V} y_i \tag{IP}$$

$$\text{s.t} \sum_{i \in S_v^\alpha} x_{iv} \geq 1 \qquad \forall v \in V \tag{2}$$

$$\sum_{v} x_{iv} \leq U_i \, y_i \quad \forall i \in V \tag{3}$$

$$x_{iv} \leq y_i \qquad \forall i, v \in V \tag{4}$$

$$x_{iv}, y_i \in \{0,1\} \quad \forall i, v \in V \tag{5}$$

We refer to the LP relaxation of the above IP as (LP).

It follows from Lemma 5 that if there is a feasible solution for C-MEDKC with stretch opt, then (LP) with $\alpha =$ opt has a solution with value at most k.

We next show how to round a solution (x^0, y^0) of (LP) to a solution $(\overline{x}, \overline{y})$ of cost at most twice the cost of the original solution and such that $\overline{y}$ is binary. All capacity constraints are violated by at most a factor of two. Moreover, if node v is assigned to facility i, i.e. $\overline{x}_{iv} > 0$, then the distance between i and v is not too large, i.e. $i \in S_v^{3\alpha}$. Finally, we show how to assign each node v to a unique center π_v, and prove that this solution to C-MEDKC has low stretch.

5.3 Algorithm Details

Starting with a fractional solution (x^0, y^0) of (LP), we iteratively modify it in order to finish with $(\overline{x}, \overline{y})$ which satisfies the conditions above. We refer to the solution at the beginning of iteration j as (x^j, y^j).

We call a center i *fractionally opened* if $y_i^0 > 0$. In the course of the algorithm we *open* a subset of the set of fractionally opened centers. The indicator variables for open center nodes are rounded to one. We let O^j be the set of open centers at the beginning of iteration j. Initially, let O^0 be the empty set.

The procedure maintains the following invariants for all iterations $1 \leq j \leq t$:

(I1) $\sum_i y_i^j \leq 2 \sum_i y_i^0$
(I2) $\sum_i x_{iv}^j \geq 1/2$ for all nodes $v \in V$
(I3) $\sum_v x_{iv}^j \leq U_i y_i^j$ for all $i \in V$

We say that a node is *satisfied* if in iteration j we have $\sum_{i \in O^j} x_{iv}^j \geq 1/2$. Let S^j denote the set of satisfied nodes in iteration j. Our algorithm stops in iteration t when no unsatisfied nodes remain. We then increase the assignment of nodes j to open centers $i \in O^t$ such that the final solution satisfies the demand constraints (2).

A detailed description of an iteration follows. An iteration j starts by selecting an unsatisfied node v_j of minimum l_v value. We let $I(v_j)$ be the set of centers that fractionally serve v_j and have not yet been opened, i.e.

$$I(v_j) = \{i \in V \setminus O^j \,:\, x_{iv_j}^j > 0\} = \{i_1, \ldots, i_p\}.$$

W.l.o.g., assume that $U_{i_1} \geq U_{i_2} \geq \cdots \geq U_{i_p}$. We now open the first $\gamma = \left\lceil \sum_{i \in I(v_j)} y_i \right\rceil$ centers from $I(v_j)$ and close all fractional centers in $\{i_{\gamma+1}, \ldots, i_p\}$, i.e. $O^{j+1} = O^j \cup \{i_1, \ldots, i_\gamma\}$. Hence $y_i^{j+1} = 1$ for all $i \in \{i_1, \ldots, i_\gamma\}$ and $y_i^{j+1} = 0$ otherwise. We let $y_i^{j+1} = y_i^j$ for all $i \notin I(v_j)$.

Notice that a variable y_i^0 for a fractionally opened center node i is modified exactly once by the procedure outlined above. This modification happens whenever i is either opened or closed. It follows from this observation that

$$\sum_{i \in I(v_j)} y_i^0 = \sum_{i \in I(v_j)} y_i^j \geq 1/2$$

where the last inequality is a consequence of the fact that v_j is unsatisfied with respect to y^j. Therefore,

Lemma 6. *Let v_j be the unsatisfied node chosen in iteration j of our algorithm and let y^0 and y^{j+1} be defined as before. We then must have $\sum_{i\in I(v_j)} y_i^{j+1} \leq 2\sum_{i\in I(v_j)} y_i^0$.*

This shows that invariant (I1) is preserved throughout the algorithm.

It remains to modify x^j and obtain x^{j+1} so that invariants (I2) and (I3) are maintained. Specifically, we have to modify x^j such that no node is assigned to closed centers and all capacity constraints are satisfied with respect to y^{j+1}.

For an arbitrary node v, let $\omega_v = \sum_{i\in I(v_j)} x_{iv}^j$ be the assignment of v to centers from $I(v_j)$. Let Ω^j be the set of unsatisfied nodes that are attached to $I(v_j)$, i.e.

$$\Omega^j = \{v \in V \setminus S^j \,:\, \omega_v > 0\}.$$

We now (fractionally) assign the nodes from Ω^j to centers $i_1, \ldots, i_\gamma$ such that for all $1 \leq l \leq \gamma$ at most U_{i_l} nodes are assigned to y_{i_l} and no node is assigned to any node in $\{i_{\gamma+1}, \ldots, i_p\}$. The existence of such an assignment (and hence the validity of (I3)) follows from the following lemma (implicit in [13]):

Lemma 7. *Let v_j, γ and $I(v_j)$ be defined as above. Then, $\sum_{i\in I(v_j)} U_i y_i^j \leq \sum_{l=1}^{\gamma} U_{i_l}$.*

We do not reassign nodes v that were satisfied with respect to (x^j, y^j). Hence, a satisfied node v might lose at most $1/2$ of its demand that was assigned to now closed centers. This entails invariant (I2).

At termination time t, all nodes are satisfied. We now obtain a solution $\overline{x}$ that satisfies the demand constraints (2) of (LP) by scaling x^t appropriately. For all $i \in O^t$, $v \in V$, let $\overline{x}_{iv} = x_{iv}^t / \sum_{i\in O^t} x_{iv}^t$. Invariant (I2) implies the following lemma.

Lemma 8. *Let i be a center opened by the above algorithm. Then, $\sum_v \overline{x}_{iv} \leq 2U_i$.*

It remains to show that whenever we have $\overline{x}_{iv} > 0$ it must be that $v \in S_v^{3\alpha}$. The proof of this lemma is similar to that of Lemma 3. It crucially uses the ordering in which the algorithm considers vertices and triangle inequality. We omit the details from this extended abstract.

Lemma 9. *Let $\overline{x}, \overline{y}$ be the solution computed by the preceding algorithm. We must have $i \in S_v^{3\alpha}$ whenever $\overline{x}_{iv} > 0$.*

An observation from [14] enables us to assign each node v to a unique center π_v without increasing the violation of any of the capacity constraints.

Lemma 10. *Let $(\overline{x}, \overline{y})$ be feasible for (2), (4) and (5) such that for all $i \in V$ we have $\sum_{v\in V} \overline{x}_{iv} \leq 2U_i\overline{y}_i$ and $\overline{y}$ is binary. Then, there exists an integral feasible solution (x, y) such that $\sum_{v\in V} x_{iv} \leq 2U_i y_i$ for all $i \in V$ and $\sum_i y_i \leq \sum_i \overline{y}_i$.*

We let $\pi_v = i$ iff $x_{iv} = 1$ and prove Theorem 3.

Proof of Theorem 3. We only need to show that for any $u, v \in V$, we have $\mathtt{d}_\pi(u, v) \leq (12\alpha + 1) \cdot \mathtt{d}_l(u, v)$. Theorem 3 then follows from Lemma 5 and the fact that we can perform a binary search to find the right estimate for α.

Let us estimate $\mathtt{d}_\pi(u,v)$: From triangle inequality, we obtain that

$$\mathtt{d}_\pi(u,v) \leq 2(\mathtt{d}_l(u,\pi_u) + \mathtt{d}_l(v,\pi_v)) + \mathtt{d}_l(u,v). \tag{6}$$

It follows from Lemma 9 that we must have $\mathtt{d}_l(u,\pi_u) \leq 3\alpha \cdot l_u \leq 3\alpha \cdot \mathtt{d}(u,v)$ and $\mathtt{d}_l(v,\pi_v) \leq 3\alpha \cdot l_v \leq 3\alpha \cdot \mathtt{d}(u,v)$. Hence we obtain together with (6) that $\mathtt{d}_l(u,v) \leq (12\alpha+1)\mathtt{d}_l(u,v)$. □

6 Open Problems

An apparent open question is to develop a unicriteria approximation algorithm for the capacitated case (maybe based on the ideas in [7]). Furthermore, an interesting remaining problem is to extend Theorem 4 to the case where we do not have a backbone network. In other words, how close to the best possible stretch can we get given limited routing table space B? A possible direction would be to explore stronger combinatorial lower-bounds and explore the merit of LP techniques.

References

1. L. Cowen. Compact routing with minimum stretch. *J. Algorithms*, 38(1):170–183, 2001.
2. M. E. Dyer and A. M. Frieze. A simple heuristic for the p-centre problem. *Operation Research Letters*, 3(6):285–288, 1985.
3. M. Elkin and D. Peleg. $(1+\epsilon,\beta)$-spanner constructions for general graphs. In *Proceedings, ACM Symposium on Theory of Computing*, pages 173–182, 2001.
4. M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, San Francisco, 1979.
5. D. Hochbaum and D. Shmoys. A best possible approximation algorithm for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.
6. W. L. Hsu and G. L. Nemhauser. Easy and hard bottleneck location problems. *Discrete Appl. Math.*, 1:209–216, 1979.
7. S. Khuller and Y. Sussmann. The capacitated K-center problem. *SIAM Journal on Discrete Mathematics*, 13(2):403–418, 2000.
8. J. Könemann, Y. Li, O. Parekh, and A. Sinha. Approximation alorithms for edge-dilation k-center problems. Technical Report #2001-E29, Graduate School of Industrial Administration, Carnegie Mellon University, 2001.
9. J. Moy. OSPF version 2. IETF RFC 2328, 1998. (http://www.ietf.org/rfc.html).
10. David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18(4):740–747, 1989.
11. J. Plesnik. On the computational complexity of centers locating in a graph. *Aplikace Matematiky*, 25:445–452, 1980.
12. J. Plesnik. A heuristic for the p-center problem in graphs. *Discrete Applied Mathematics and Combinatorial Operations Research and Comp. Sci.*, 17:263–268, 1987.
13. D. Shmoys, E. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *ACM Symp. on Theory of Computing*, pages 265–274, 1997.
14. D. B. Shmoys and E. Tardos. An improved approximation algorithm for the generalized assignment problem. *Math. Programming*, A62:461–474, 1993.

Forewarned Is Fore-Armed: Dynamic Digraph Connectivity with Lookahead Speeds Up a Static Clustering Algorithm

Sarnath Ramnath

St Cloud State University, 720, 4th ave S., St Cloud MN 56303.
sarnath@eeyore.stcloudstate.edu

Abstract. Dynamic data structures are presented for directed graphs that maintain (a) Transitive Closure and (b) Decomposition into Strongly Connected Components in a "semi-online" situation which improve the static algorithms for minimum sum-of-diameters clustering are improved by a $O(\log n)$ factor.

1 Introduction

A basic problem of cluster analysis is to partition a given set of entities into homogeneous and/or well-separated classes, called *clusters*. Separation is commonly characterized by the dissimilarity between objects, which can be expressed as the *distance* between objects. A measure often used in characterizing the homogeneity of a set is the *diameter*, which is defined as the largest distance between any pair of items in the set. [5]

The minimum sum of diameters clustering problem is described as follows:

Input: *A set of n items, $a_1, a_2, ...a_n$ and an integer k; associated with each pair (a_i, a_j) is a length l_{ij}, which represents the distance between a_i and a_j.*

Output: *A partitioning of the set into k subsets, such that the sum of the diameters of the subsets is minimized.*

The input can be represented by a weighted graph, which we shall call the *Cluster Graph*, as follows: represent each item a_i by a vertex numbered i; add an edge e_{ij} between vertex i and vertex j with length l_{ij}. The output is a partitioning of the vertex set into two clusters C_0 and C_1, with diameters D_0 and D_1 respectively.

The problem is known to be NP-hard for $k \geq 3$, and the first approximation algorithms for general k were recently given by Doddi et al. [3]. For the case $k = 2$, Hansen and Jaumard gave an $O(n^3 \log n)$ algorithm [5] which was improved to $O(mn \log n)$ by Monma and Suri [10]. We shall assume that $k = 2$ for the rest of this paper.

The algorithm used in [5] to find the best partitioning, solves $O(n \log n)$ 2-SAT instances, each of which could take $O(m)$ time in the worst case. Here we present algorithms that dynamically solve $O(m)$ 2-SAT instances, performing an average of $O(n^3/m)$ and $O(n)$ operations, respectively, for each instance. As

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 220–229, 2002.

a result of these, we obtain algorithms for minimum sum-of-diameters clustering that run in $O(n^3)$ and $O(mn)$ respectively. The first algorithm dynamically maintains the transitive closure, and the second one dynamically maintains the partitioning of a graph into Strongly Connected Components (*SCC*s). Both these approaches use the notion of *perfect deletion lookahead*: at any instant we know all the deletable edges in the graph, and the order in which these edges are to be deleted. The update times obtained for digraph connectivity by the earlier researchers [8,4,7,2,9] do not help reduce the complexity of the clustering problem. In particular, the dynamic approach to the clustering operation requires in the worst case, a sequence of $O(m + n)$ updates (inserts and deletes, interleaved), and $O(n)$ queries after each update. To improve the algorithm in [5] would therefore mean an amortized/average update time of $O(n)$. In this paper we use the perfect lookahead available to us to speed up the update operations. Khanna, Motwani and Wilson [7] used partial lookahead to maintain the transitive closure, but the resulting update times ($O(n^{2.18})$ with $n^{0.18}$ lookahead) are not good enough to improve the upper-bounds for clustering in [5,10]. The scheme used in this paper to maintain transitive closure performs insertions in $O(n^2)$ time and deletions in $O(1)$ time; the scheme to maintain a decomposition into *SCC*s has an amortized cost of $O(m+n)$ for each deletable edge, and a cost of $O(n)$ for each non-deletable edge.

The next section presents an overview of the approach in [5] and some background on the relationship between 2-SAT and directed graphs. The two following sections present the $O(n^3)$ and $O(mn)$ algorithms; the last section concludes the paper. To reduce the length of the manuscript, some of the proofs have been moved to the appendix.

2 Preliminaries

We assume without loss of generality that $D_0 \geq D_1$. We say that an edge belongs to a cluster if both end vertices of the edge belong to the cluster. Since the diameter of a cluster is the length of the longest edge in the cluster, the only candidates for D_0 and D_1 that we need to consider are the are the edge lengths. Let S_l denote the set of edge lengths.

The algorithm in [5] works as follows:

Algorithm Cluster

Step 1: Identify all edge lengths, d_0, in S_l that are possible candidates for D_0.

Step 2: For each candidate edge d_0, found in Step 1, identify the smallest value d_1 in S_l, such that there exists a partitioning of the cluster graph into two sets with diameters not exceeding d_0 and d_1 respectively.

Step 3: Choose D_0 and D_1 to be the pair (d_0, d_1) such that the sum of d_0 and d_1 is minimized.

end Cluster

The following three results are from [5].

Lemma 1: *Consider a Maximum Spanning Tree (MST) for the cluster graph, constructed using Kruskal's algorithm. The only edges whose lengths are candidates for D_0 are the edge that completed the first odd cycle and the edges included in the spanning forest before the first odd cycle was encountered. It follows that there are at most n candidates for D_0, and that all these candidates can be found in time $O(m + n \log n)$.*

The above lemma tells us how to compute step 1 of Cluster in $O(m+n \log n)$ time. Since there are at most n candidates for D_0, step 3 is trivially done in $O(n)$ time. Step 2 is the most expensive part of the computation, for which we describe improved algorithms in the following sections.

Lemma 2: *Let D_{min} denote the length of the edge that completed the first odd cycle. All candidates for D_1 are lesser than or equal to D_{min}.*

Lemma 3: *Consider the assertion:* "There is a partitioning of the vertices into 2 clusters with diameters not exceeding d_0 and d_1"*. This assertion can be represented as a 2CNF expression with n variables and $p + q$ conjuncts, where p is the number of edges with length greater than d_0 and q is the number of edges with length greater than d_1.*

The construction of the 2CNF expression uses two kinds of constraints. If for some edge e_{ij}, $l_{ij} > d_1$, then we need to add a condition that vertex i and vertex j cannot both be in C_1; if x_i is a boolean variable set to 0 if vertex i falls in C_0 (set to 1 if vertex i falls in C_1), then this condition is expressed by the conjunct (not(x_i) OR not(x_j)). Likewise, if $l_{ij} > d_0$, we add the conjunct (x_i OR x_j). We shall refer to the constraint (x_i OR x_j) as the *Type0* constraint and the constraint (not(x_i) OR not(x_j)) as the *Type1* constraint of the edge e_{ij}. It has been shown in [1] that any 2CNF expression with n variables and m conjuncts can be represented as a digraph with $2n$ vertices and $2m$ directed arcs, as follows: For each variable x_i, $1 \leq i \leq n$, add 2 vertices - u_i, labelled x_i and v_i, labelled not(x_i). For each constraint a OR b, where a and b are literals, add two directed edges - one from the vertex labelled by the simplest expression equivalent to $not(a)$ (since a could itself be negated, we may have to remove the double negation to obtain the vertex label) to the vertex labelled b and another directed edge from the vertex labelled by the simplest expression equivalent to $not(b)$ to the vertex labelled a. The 2CNF expression is unsatisfiable if and only if there is a directed cycle containing both u_i and v_i, for some i, $1 \leq i \leq n$. It was shown in [1] that the satisfiability of a 2CNF expression can be decided by looking for such directed cycles in $O(m + n)$ time.

To decide a 2-SAT instance, we construct a directed graph. In this digraph we have to look at $O(n)$ pairs of vertices to check if any of these pairs falls in the same Strongly Connected Component(SCC). If we have the transitive closure or the decomposition into SCCs, this check can be performed with $O(n)$ queries, each requiring $O(1)$ time.

Let $l_1, l_2, ...l_q$ be the list of edge lengths in the Cluster graph, sorted in descending order, such that $l_p = D_{min}$, $1 \leq p \leq q$. There are at most p instances of d_0, viz., $l_1, l_2, ..., l_p$. In the process of finding d_1 for each d_0, we can either start with $d_0 = l_1$ and then decrease d_0 all the way down to l_p, or we can start with

$d_0 = l_p$ and increase to l_1. If we choose the former, we perform $O(n)$ inserts and $O(m)$ deletes; this is the approach we use in section 3, where we dynamically maintain the transitive closure for a digraph, with each insertion taking $O(n^2)$ time and each deletion taking $O(1)$ time, giving us a cost that is $O(n^3)$. If we choose the latter, we have $O(n)$ deletes and $O(m)$ inserts, i.e., the graph has $O(n)$ deletable edges. The scheme presented in section 4 maintains the decomposition into SCCs in such a way that each deletable edge requires $O(m)$ operations and each non-deletable edge requires $O(n)$ operations. Both these approaches must decide $O(m)$ 2-SAT instances, which would require $O(mn)$ steps. Thus we have an $O(n^3)$ algorithm if we maintain the transitive closure, and an $O(mn)$ algorithm if we maintain the decomposition into SCCs.

We designate the directed graph used to represent a 2CNF expression as the *Constraint Graph.* Looking at the types of constraints imposed by the clustering problem, it is obvious that no two negated literals are connected by a directed arc, and likewise, no two non-negated literals are connected by a directed arc. Each *Type0* constraint induces two edges that are directed from negated literals to non-negated ones, and each *Type1* constraint induces two edges that are directed from non-negated literals to negated ones. We designate these edges *Type0* and *Type1* edges respectively. In the following sections, we shall dynamically insert these edges into the constraint graph to obtain faster algorithms. Whenever we insert a constraint, it means that both the induced edges are added to the constraint graph. The following lemma tells us something about the edges in the cluster graph that complete even cycles in Kruskal's algorithm.

Lemma 4: *Let $e_1, e_2, ...e_m$ be a valid order in which Kruskal's algorithm considers the edges of the cluster graph, in order to construct the MST; let T_i denote the subset of edges from $e_1, e_2, ...e_i$ that are included in the MST by Kruskal's algorithm, and let e_i, $1 \leq i \leq m$, be an edge connecting vertices i_s and i_t, that completes an even cycle. Then, in the constraint graph induced by the* Type0 *and* Type1 *constraints of the edges in T_{i-1}, (i) u_{i_s} and v_{i_t} belong to the same strongly connected component; (ii) u_{i_t} and v_{i_s} belong to the same strongly connected component.*

3 An $O(n^3)$ Algorithm

This section first describes a fully dynamic graph connectivity algorithm with the following characteristics: *(i) needs complete lookahead on deletions, (ii) $O(n^2)$ time for each insertion, (iii) $O(1)$ time for each deletion, (iv) $O(1)$ query time and (v) $O(n^3)$ pre-computation time.*

We define the following concepts:

Deletion Time Stamp (DTS): Associated with each edge is a *DTS* that gives the order in which the edge will be deleted, the edge with largest DTS being the next edge to be deleted. For deletable edges, this is a unique integer in the range $[1..n^2]$. Edges that will never be deleted have a $DTS = 0$. If an edge does not belong to the graph, it has a *DTS* of ∞.

Current Time Stamp (CTS): An integer between 0 and n^2. When we delete an edge, the *CTS* is decremented. A *CTS* of i indicates that the graph has i

deletable edges. When we add a deletable edge, *CTS* is incremented. At any point, *CTS* is equal to the largest DTS for all the edges in the graph.

Persistence Number (PN): Associated with each directed path in the graph is a *PN* that is computed as the maximum of all the *DTS* values of the edges on that path. Intuitively, the *PN* of a path is a measure of how many deletes it will take to disconnect the path . For a path with a *PN* of p, this measure is computed as *CTS* - $p + 1$. Therefore, given 2 paths, the one with a lower *PN* is more persistent.

Connectivity Number (CN): For each pair of vertices (u, v) we have a *CN* that is computed as the minimum of the *PN*s over all paths that connect from u to v. Intuitively, the *CN* gives us a measure of the number of deletes needed to eliminate all paths from u to v. If $CN(u, v) = c$, then this measure is computed as *CTS* - $c + 1$. Thus, if $CN(u, v) \leq 0$, there is no directed path from u to v.

Our data structure, for a graph with n vertices is an $n \times n$ matrix, which contains, for each pair of vertices (u, v), the *DTS* of the edge (u, v) and $CN(u, v)$. Let $D^{(k)}(i, j)$ denote the *PN* for the most persistent path from i to j, such that none of the intermediate vertices has an index greater than k. We have the following dynamic programming recurrence: $D^{(0)}(i, j) = DTS(i, j)$ and $D^{(k+1)}(i, j) = min[D^{(k)}(i, j), max(D^{(k)}(i, k + 1), D^{(k)}(k + 1, j))]$ Since $CN(i, j) = D^{(n)}(i, j)$, our data structure can be pre-computed in $O(n^3)$ time.

The only work done to update the data structure is at the time of insertion, when the *CN* values are updated in accordance with the above recurrence. If (u, v) is the new edge to be inserted with *DTS* t, then, for each pair (x, y) we have a new potential path from x to y, viz., x to u to v to y, and the *PN* of this path must be taken into account to determine $CN(x, y)$. There are two other issues we must deal with: (1) Some deletes may have been performed since the last insert and (2) if $t > 0$, there maybe an existing edge in the graph with $DTS = t$, i.e., the new edge is inserted somewhere in the middle of the deletion sequence. These cases are taken care of in $Insert(u, v, t)$.

The data structure supports the following operations:

- $Insert(u, v, 0)$: (Insert a non-deletable edge from u to v) For all pairs of vertices (p, q) such that $CTS < CN(p, q) < \infty$, set $CN(p, q) = \infty$. For each pair of vertices (x, y), $CN(x, y) = \min(CN(x, y), \max(CN(x, u), CN(v, y)))$.
- $Insert(u, v, t)$: (Insert an edge with *DTS* t, from u to v) For all pairs of vertices (p, q) such that $CTS < CN(p, q) < \infty$, set $CN(p, q) = \infty$. If $CTS \geq t$, increment the *DTS* for each edge that has a current *DTS* value $\geq t$. Increment *CTS*. For each pair of vertices (p, q) with $CN \geq t$, increment $CN(p, q)$. For each pair of vertices (x, y), $CN(x, y) = \min(CN(x, y), \max(CN(x, u), CN(v, y), t))$.
- Delete(u,v): (Deletes the next edge in sequence). Decrement *CTS*; $DTS(u, v) = \infty$.
- Path(u, v): (Returns True if there is a path connecting from u to v; False otherwise). If $CTS < CN(u, v)$, return False; else return True.

Therorem 5: *The data structure described above maintains the transitive closure of a digraph, satisfying the following conditions: (i)* $O(n^2)$ *time for each*

insertion; (ii)$O(1)$ *time for each deletion; (iii)* $O(1)$ *query time; (iv) given an input graph, the required data structures can be pre-computed in* $O(n^3)$ *time.*

We use the above data structure to compute step 2 of Cluster as follows: *Add all the* Type1 *constraints to the constraint graph. Next, add the* Type0 *constraints for the edges with length greater than* D_{min}*, starting with the longest edge. As soon as we get a cycle, containing a variable and its negation, we remove* Type1 *constraints in decreasing order of lengths, until the cycle is removed. By keeping track of the length of the edges whose constraints created the cycle and the length of the edges whose constraints were removed to eliminate the cycle, we can obtain all the* (d_0, d_1) *pairs needed in Step 2 of Cluster.* This idea is elaborated below.

Let $l_1, l_2, ..., l_q$ be the q distinct edge lengths in the cluster graph, and let S_i denote the set of all edges of length l_i in the cluster graph. To simplify the presentation, we define $l_0 = \infty$, $l_{q+1} = 0$ and S_0 and S_{q+1} as the associated (empty) sets. Let $D_{min} = l_p$, $1 \leq p \leq q$.

Algorithm Cluster1

1. Insert all the *Type1* constraints for edges with length greater than $D_{min,}$, into the constraint graph as undeletable edges.
2. Insert all the *Type1* constraints for edges with length less than or equal to D_{min}, into the constraint graph as follows:
   $DTS = 1$;
   For $i = p$ to q
   For each edge e_{jk} in S_i,
   $Insert(u_j, v_k, DTS++)$; $Insert(u_k, v_j, DTS++)$
   end for;
   end for;
3. $j = q + 1$;
   For $i = 0$ to p-1
   For each edge e_{lk} in S_i that does not complete an even cycle
   $Insert(v_l, u_k, 0)$; $Insert(v_k, u_l, 0)$;
   end for;
   While (constraint graph is unsatisfiable)
   j= j-1; Delete all $Type1$ constraints for edges in S_j;
   end while;
   Record (l_{i+1}, l_j) as a (d_0, d_1) pair.
   end for;

Theorem 6: *Algorithm Cluster1 correctly computes Step 2 of the algorithm Cluster in* $O(n^3)$ *time.*

4 A $O(mn)$ Algorithm

Definition (Strongly Connected Subgraph): *A Strongly Connected Subgraph (SCS) of a digraph* $G(V, E)$ *is a set of vertices* $C \subseteq V$ *such that for every pair of vertices* (u, v), u, $v \in C$, *there exist directed paths* p_f *from* u *to* v *and* p_b *from* v *to* u *in* G, *such that neither* p_f *nor* p_b *contains an intermediate vertex*

that does not belong to C. A Strongly Connected Component (SCC) is a maximal SCS.

Cicerone et al [2] use the following technique for maintaining the transitive closure under insertions: *Maintain n incomplete BFS traversals, one starting at each vertex. Whenever an edge* (u, v) *is added, consider all vertices* x*, whose BFS has reached* u*, and restart these traversals, by adding* v *to the queue of each such* x*. Since there are n BFS traversals and each edge that is inserted can be traversed by a traversal at most once, the amortized cost of insertion is* $O(n)$. To adapt this technique for our situation (i.e., maintain the decomposition into *SCC*s), we make the following observations:

- Let it be that each of the vertices in the graph is arbitrarily assigned a unique integer value, called the *priority* of the vertex. (The priority of vertex x is denoted $pr(x)$.) We can then associate an integer with each *SCS*, which is the highest priority of all the vertices in the *SCS*. We also associate a *SCS* with each vertex x, denoted x_{SCS}, consisting of all vertices y such that the digraph contains a directed path from x to y and a directed path from y to x, neither of which pass through a vertex with priority greater than $pr(x)$.
- If there are no deletions, it is easy to maintain the decomposition into *SCC*s using the technique in [2]. For each vertex v, we maintain the largest integer v_{max} such that there is a directed path from v to the vertex with priority v_{max} and vice-versa. v is then designated as belonging to the *SCC* v_{max}.
- To deal with deletable edges, we introduce the concept of *enodes* (or "edge-nodes"). Associated with a deletable edge directed from node u to node v is an enode (β, say). Instead of adding an edge from node u to node v, we add two directed edges: one from u to β, and the other from β to v. Each deletable edge has an associated *Deletion Time Stamp* (DTS); the enode β is assigned a priority of (n + DTS of the edge from u to v), where n is the number of nodes in the original graph.
- When an edge is deleted, we need to ensure that the connectivity provided by the deleted edge is not being used anymore. To ensure this, the following constraint is imposed on the BFS traversals from each vertex: *Consider a BFS traversal rooted at* v*; a vertex* u *is visited by this traversal only if* $pr(v) > pr(u)$. Since an edge with a higher DTS corresponds to an enode with a higher priority, any path from a vertex v to a vertex u, discovered by a BFS rooted at v, cannot pass through any enode β such that $pr(\beta) > pr(v)$. Consequently, this path cannot be disconnected due to the deletion of the edge associated with any enode β that has a priority greater than $pr(v)$. This constraint does create a new problem: *given vertices* u*,* v *and* w *with* $pr(u) > pr(v) > pr(w)$ *such that* v *lies on every path from* w *to* u*, the BFS traversal from* w *will not find a path to* u*. Nonetheless,* u *and* w *must be recognized as belonging to the same SCC.* To overcome this problem, *we carry out two BFS traversals from each vertex - one following the forward arcs, and one following the backward arcs.* Thus with each vertex x (a vertex could be a node or an enode) we have an associated *SCS* consisting of all vertices y such that there is a path p_f from x to y following the directed arcs in the forward direction, and a path p_b from x to y following the directed arcs in the reverse direction, such that neither path has a a vertex with priority

greater than $pr(x)$. Such an arrangement correctly finds all the *SCC*s, since the node x which has the highest priority of all nodes in the SCC, will reach all nodes in the *SCC* both forwards and backwards, without passing through any node with priority greater than $pr(x)$. This arrangement also yields the *SCS*, x_{SCS}, for each vertex x; x_{SCS} contains all vertices y which can be reached from x both forwards and backwards without passing through any vertex with priority greater than $pr(x)$.

Our data structure keeps track of the following information with each vertex x:

1. two boolean arrays: F_x, that stores all the nodes visited by the forward BFS from x, and B_x, that stores all the nodes visited by the backward BFS from x. If x is an enode, these arrays are of size n^2; if x is a node, these are of size n.
2. the parent of x_{SCS}, if any, in F_{SCS} .
3. two lists: a list of nodes in the corresponding *SCS* x_{SCS}, and the list of children of x_{SCS} in F_{SCS}.
4. if x is a node, then we keep two lists - L_x^f that stores all vertices whose forward BFS has visited x and L_x^b that stores all vertices whose backward BFS has visited x.
5. two BFS queues: Q_x^f and Q_x^b
6. an integer x_{max}, which is the highest priority of all vertices y such that both forward and backward traversals from y have visited x.

The data structure supports the following operations:

- *Insert*(u, v, t): Insert a deletable edge from node u to node v, with *DTS* t.
  1. Create an enode β, and add directed edges from u to β and β to v.
  2. For each enode α such that $pr(\alpha) \geq n + t$, increment the priority of α; for each vertex x such that $x_{max} \geq n + t$, increment x_{max}.
  3. For each enode α in L_u^f, such that $pr(\alpha) > n + t$, insert β in Q_α^f and re-start that traversal.
  4. For each enode α in L_v^b, such that $pr(\alpha) > n + t$, insert β in Q_α^b and re-start that traversal.
  5. Do a forward BFS and a backward BFS from β , and enumerate the items in β_{SCS} by taking the intersection of the sets of nodes visited by the two traversals.
  6. Find the enode α with the smallest priority such that $\beta \in \alpha_{SCS}$, and make β_{SCS} the child of α_{SCS} in F_{SCS} .
- Insert(u, v, 0): Insert a non-deletable edge from node u to node v
  1. For each vertex x in L_u^f such that $pr(x) > pr(v)$, insert v into Q_x^f.
  2. For each vertex x in L_v^b such that $pr(x) > pr(u)$, insert u into Q_x^b.
- *SCC* (u, v): Check if node u and node v belong to the same strongly connected component.
  Return $u_{max} = v_{max}$.

– delete(): Delete the edge with the highest DTS
  Let w be the enode with the highest priority
  For each child v_{SCS} of w_{SCS}
    For each vertex x in v_{SCS}
      $x_{max} = pr(v)$

Theorem 7: *The data structure described above has the following characteristics:(i) Incurs an expense of $O(m^\star + n)$ for each deletable edge, where $m^\star$ is the maximum number of edges present in the graph at any time in the lifespan of the deletable edge; (ii) Inserts non-deletable edges in $O(n)$ amortized time; (iii) Correctly answers* SCC *queries in $O(1)$ time*

Using the above data structure, we can obtain an $O(mn)$ algorithm to compute Step 2 of Cluster. The approach here is to first add to the constraint graph all the $Type0$ and $Type1$ constraints for edges with length greater than D_{min}. We then add $Type1$ constraints for edges of length less than D_{min}, in decreasing order of edge lengths. Whenever we reach an unsatisfiable 2-SAT instance, $Type0$ constraints are deleted in increasing order of edge lengths until satisfiability is restored. By keeping track of the length of the edges whose constraints created unsatisfiability and the length of the edges whose constraints were removed to restore satisfiability, we can obtain all the $(d_0,\ d_1)$ pairs needed in Step 2 of Cluster.

Once again, let $l_1, l_2, ..., l_q$ be the q distinct edge lengths in the cluster graph, and let S_i denote the set of all edges of length l_i in the cluster graph. To simplify the presentation, we define $l_0 = \infty$, $l_{q+1} = 0$ and S_0 and S_{q+1} as the associated (empty) sets. Let $D_{min} = l_p$, $1 \leq p \leq q$.

Algorithm Cluster2

1. Insert all the *Type1* constraints for edges with length greater than $D_{min,}$, into the constraint graph as undeletable edges.
2. Insert all the *Type0* constraints for edges with length greater than $D_{min,}$, as follows:
   $DTS = 1$;
   For $i = 1$ to p-1
     For each edge e_{jk} in S_i that does not complete an even cycle
       $Insert(v_j, u_k, DTS++)$; $Insert(v_k, u_j, DTS++)$
     end for;
   end for;
3. $j = p$-1;
   For $i = p$-1 down to 1
     While (constraint graph is satisfiable)
       $j = j + 1$; Insert *Type1* constraints for edges in S_j;
     end while;
     Record $(l_{i+1},\ l_j)$ as a $(d_0,\ d_1)$ pair.
     Remove *Type0* constraints for all edges in S_i.
   end for;

Theorem 8: *Algorithm Cluster2 correctly computes Step 2 of Cluster in $O(mn)$ time.*

5 Conclusion

We have discussed algorithms that solve the minimum sum-of-diameters clustering problem in $O(n^3)$ and $O(mn)$ time respectively. In practice, the $O(n^3)$ algorithm may be better due to the fact that the only data structure used is an array. It would be interesting to determine experimentally at what value of m the second algorithm becomes more efficient. It is possible to use the ideas described in this paper to obtain an $O(qm)$ algorithm for minimum sum of diameters clustering, where q is the number of distinct edge lengths. This would actually give us better performance if q is $o(n)$. Whether there is a $o(mn)$ algorithm for the general case remains an open question. Another interesting question would be to determine the relative complexities of maintaining the transitive closure vs. maintaining the decomposition into SCCs for general digraphs, in the absence of lookahead.

Acknowledgments. The author would like to thank Pierre Hansen, Venkatesh Raman and S.N. Maheshwari for several useful discussions and references.

References

1. B. Aspvall, M.F. Plass and R.E. Tarjan. A Linear-time Algorithm for Testing the Truth of Certain Quantified Boolean Expressions, IPL, 8, 1979, pp 121-123.
2. S. Cicerone, D. Frigioni, U Nanni and F Pugliese. A uniform approach to semi-dynamic problems on digraphs, TCS, 203, 1998, pp 69-90.
3. S. R. Doddi, M. V. Marathe, S. S. Ravi, D. S. Taylor and P. Widmayer. Approximation algorithms for clustering to minimize the sum of diameters. In Proc. of 7th SWAT, 2000, LNCS vol 1851, pp 237-250.
4. C. Demetrescu and G.F. Italiano. Fully Dynamic Transitive Closure: Breaking Through the $O(n^2)$ barrier, 41st IEEE FOCS pp. 381-389.
5. P. Hansen and B. Jaumard. Minimum Sum of Diameters Clustering, Journal of Classification, 4:215-226, 1987.
6. P. Hansen and B. Jaumard. Cluster analysis and mathematical programming, Mathematical Programming, 79, 1997, pp 191 - 215.
7. S. Khanna, R. Motwani and R.H. Wilson. On certificates and lookahead on dynamic graph problems, Proc 7th ACM-SIAM Symp. Discr. Algthms, 1996, pp222-231.
8. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. Proc. 40th IEEE FOCS, 1999.
9. V. King and G. Sagert. A Fully Dynamic Algorithm for Maintaining the Transitive Closure, ACM STOC 1999, pp 492-498.
10. C. Monma and S. Suri. Partitioning Points and Graphs to Minimize the Maximum or the Sum of Diameters, Proc. 6th Intl. Conf. on Theory and Applications of Graphs, Kalamazoo, Michigan, May 1989, pp 899-912.

Improved Algorithms for the Random Cluster Graph Model

Ron Shamir and Dekel Tsur

School of Computer Science, Tel-Aviv University
{rshamir,dekelts}@tau.ac.il

Abstract. The following probabilistic process models the generation of noisy clustering data: Clusters correspond to disjoint sets of vertices in a graph. Each two vertices from the same set are connected by an edge with probability p, and each two vertices from different sets are connected by an edge with probability $r < p$. The goal of the clustering problem is to reconstruct the clusters from the graph. We give algorithms that solve this problem with high probability. Compared to previous studies, our algorithms have lower time complexity and wider parameter range of applicability. In particular, our algorithms can handle $O(\sqrt{n}/\log n)$ clusters in an n-vertex graph, while all previous algorithms require that the number of clusters is constant.

1 Introduction

Clustering is a fundamental problem that has applications in many areas. We study the clustering problem using a random graph model: A *cluster graph* is a random graph $G = (V, E)$ which is built by the following process: The vertex set V is a union of disjoint sets $V_1, \ldots, V_m$ called *clusters*. Each two vertices from the same set are connected by an edge with probability p, and each two vertices from different sets are connected by an edge with probability $r < p$. The random choices are all independent. The *clustering problem* is, given a cluster graph G, to find the clusters $V_1, \ldots, V_m$.

The random cluster graph models a common situation in experimental data, where noise obscures the true clusters: The probability $1-p$ is the *false negative* probability, i.e., the chance of incorrectly having no edge between two vertices of a cluster. Similarly, r is the *false positive* probability. If errors of each type are independent and identically distributed, one gets the above model.

In this paper we give several algorithms that solve the clustering problem with high probability. When addressing the clustering problem under the random graph model, several parameters are interrelated: Obviously, the smaller the gap $\Delta = p - r$, the harder the problem. Also, the size of the smallest cluster $k = \min_i |V_i|$ is limiting the performance, as very small clusters may be undetectable due to noise. The value k also bounds the number of clusters m. The challenge is to obtain provably good performance for a wide range of values for each parameter. All previous studies addressed the problem when the number

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 230–239, 2002.

of clusters is constant and most studied the case of two equal sized clusters. Our algorithms relax both of these assumptions simultaneously and at the same time achieve better running time.

Let us be more precise. Denote the total number of vertices in the graph by n. We give an $O(mn^2/\log n)$ algorithm that solves the clustering problem with high probability, assuming that $k = \Omega(\Delta^{-1-\epsilon}\sqrt{n \log n})$ for some constant $\epsilon > 0$. The running time improves if k is asymptotically larger than its lower bound. Furthermore, if the sizes of the clusters are equal (or almost equal), we give an $O((m/\log n+1)n^2)$ algorithm that requires only $k = \Omega(\Delta^{-1}\sqrt{n}\log n)$ (here too, the running time improves if k is asymptotically large).

The random graph model was studied by several authors [1,6,2,8,9,5,3,7]. In these papers, the input is a cluster graph, and the goal is to find a vertex partition which minimizes some function, e.g., the number of edges between different sets. It is not hard to show that w.h.p., the partition $V_1, \ldots, V_m$ is optimal, and therefore these problems are asymptotically equivalent to the clustering problem. A comparison between the results of these papers and our results is given in Table 1. The algorithms presented here have a wider range of provable performance than each of the previous algorithms, and are also faster when restricted to the same parameter range. For example, for unequal sized clusters, the algorithm of Ben-Dor et al. [1] requires $k = \Omega(n)$ and $\Delta = \Omega(1)$, while our algorithm can handle instances with $k = \Theta(\sqrt{n \log n})$, and instances with $\Delta = \Theta(n^{-1/2+\epsilon})$. Furthermore, under the requirements of Ben-Dor et al., the running time of our algorithm is $O(n \log n)$. For the case of two equal sized clusters, our algorithm handles almost the same range of Δ as the algorithm of Boppana [2], but our algorithm is faster, and is also more general since it handles as many as $m = \Theta(\sqrt{n}/\log n)$ clusters.

We note that Table 1 cites results as given in the papers, even though in several cases better results can be obtained by improving the analysis or by making small modifications to the algorithms. For example, the algorithm of Condon and Karp can be extended to the case when the number of clusters is non-constant. Also, the running time of the algorithm of Ben-Dor et al. can be improved to $O(n \log n)$ [11].

Due to lack of space, most proofs are omitted or sketched only. Furthermore, we only describe the algorithm for the almost equal sized cluster case.

2 Preliminaries

For a graph $G = (V, E)$, and a vertex $v \in V$, we denote by $N(v)$ the set of neighbors of v. We use $d(v)$ to denote the degree of a vertex v, namely $d(v) = |N(v)|$. For a vertex v, and a set S of vertices, denote $d_S(v) = |N(v) \cap S|$. In particular, for $u \neq v$, $d_{\{u\}}(v)$ is equal to 1 if $(u, v) \in E$ and 0 otherwise. For a set of vertices S, denote by G_S the subgraph of G induced by S.

We use the O-notations O, Ω and o with their usual meaning. However, we write $f \leq O(g)$ instead of the more common $f = O(g)$ to emphasize that an upper bound on f is given. We also use $f \leq \hat{o}(g)$ to denote that $f(n) \leq c \cdot g(n)$

Table 1. Results on the clustering problem (sorted in chronological order). For the comparison, the lower bound $k = \Omega(\Delta^{-1}\sqrt{n}\log n)$ of our algorithm for equal sized clusters was translated to a lower bound on Δ using the fact that $k \leq n/m$. Note that all previous papers assume $m = 2$ or $m = O(1)$ (the requirement $k = \Omega(n)$ in [1] implies that $m = O(1)$), and all except [1] assume equal sized clusters. For the * values, no implicit requirement is made, though some requirement is implied by the bound on the other parameter.

Paper	Requirements		Complexity
	k	Δ	
Ben-Dor et al. [1]	$\Omega(n)$	$\Omega(1)$	$O(n^2 \log^{O(1)} n)$
This paper	$\Omega(\Delta^{-1-\epsilon}\sqrt{n\log n})$	$\Omega(n^{-1/2+\epsilon})^*$	$O(mn^2/\log n)$

(a) General case.

Paper	Requirements		Complexity
	m	Δ	
Dyer & Frieze [6]	2	$\Omega(n^{-1/4}\log^{1/4} n)$	$O(n^2)$
Boppana [2]	2	$\Omega(\sqrt{p}n^{-1/2}\sqrt{\log n})$	$n^{O(1)}$
Jerrum & Sorkin [8]	2	$\Omega(n^{-1/6+\epsilon})$	$O(n^3)$
Jules [9]	2	$\Omega(1)$	$O(n^3)$
Condon & Karp [5]	$O(1)$	$\Omega(n^{-1/2+\epsilon})$	$O(n^2)$
Carson & Impagliazzo [3]	2	$\omega(\sqrt{p}n^{-1/2}\log n)$	$O(n^2)$
Feige & Kilian [7]	2	$\Omega(\sqrt{p}n^{-1/2}\sqrt{\log n})$	$n^{O(1)}$
This paper	$O(\sqrt{n}/\log n)^*$	$\Omega(mn^{-1/2}\log n)$	$O((m/\log n+1)n^2)$

(b) Equal sized clusters.

for some constant c that can be made arbitrarily small, depending on other constants (This imply in particular that $f \leq O(g)$). For example, if we have the requirement $k \geq \Omega(\sqrt{n})$, then we can write that $m \leq n/k \leq \hat{o}(\sqrt{n})$ since the hidden constant in the latter inequality can be made arbitrarily small by increasing the hidden constant in the former inequality.

For proving the correctness of our algorithms we use the following theorems which give estimates on the sum of independent random variables. The following theorem is derived from Chernoff [4].

Theorem 1. *Let $X_1, \ldots, X_n$ be independent Bernoulli random variables, and let $X = \sum_{i=1}^{n} X_i$. Then, $\mathrm{P}\left[|X - \mathrm{E}[X]| \geq a\sqrt{3\mathrm{E}[X]}\right] \leq 2e^{-a^2}$ for any $a \geq 0$.*

Theorem 2 (Esseen's Inequality). *Let $X_1, \ldots, X_n$ be independent random variables such that $\mathrm{E}[X_i] = 0$ and $\mathrm{E}[|X_i|^3] < \infty$, for $i = 1, \ldots, n$. Let $B_n = \sum_{i=1}^{n} \mathrm{E}[X_i^2]$ and $L_n = B_n^{-3/2} \sum_{i=1}^{n} \mathrm{E}[|X_i|^3]$. Then $|\mathrm{P}\left[B_n^{-1/2}\sum_{i=1}^{n} X_i < x\right] -$*

$\Phi(x)| \leq AL_n$ *where* A *is an absolute constant and* $\Phi(x)$ *denotes the normal* $(0,1)$ *cumulative distribution function.*

For a proof of Esseen's inequality see [10, p. 111]. We also use the following lemma.

Lemma 1. *Let* A *be a set with* n *elements, and* B *be a subset of* A *with* k *elements. Let* S' *be a random subset of* A*, and let* S *be a random subset of* $A - S'$ *of size* s*. Then* $\mathrm{P}\left[||B \cap S| - \frac{k}{n}s| \geq a\sqrt{3(k/n)s}\right] \leq 6\sqrt{k}e^{-a^2}$ *for any* $a \geq 0$.

In the following, we say that an event happens with high probability (w.h.p.) if its probability is $1 - n^{-\Omega(1)}$.

3 The Basic Algorithm

In this section we give a top-level description of our algorithms.

Let $G = (V, E)$ be a cluster graph. Denote $A_i = |V_i|$, $a_i = |V_i|/n$ and $\Gamma = \max_i |\frac{|V_i|}{n} - \frac{1}{m}|$. A set $S \subseteq V$ is called a *subcluster* if $S \subseteq V_i$ for some cluster V_i. An induced subgraph G_S is called a *cluster collection* if for all i, either $V_i \subseteq S$ or $V_i \subseteq V - S$. Suppose we have a procedure Find(G, V') that receives a cluster graph $G = (V, E)$ and a set $V' \subseteq V$, and returns a subcluster of size $\Omega(\log n/\Delta^2)$, by only considering vertices and edges in $G_{V'}$. Then, we use the following algorithm for solving the clustering problem: Repeatedly, find a subcluster S in the graph, find the cluster that contains S, and remove the cluster from the graph. Formally, the algorithm, Solve(G), is as follows:

1. Randomly select a set W of $n/2$ vertices.
2. $S_0 \leftarrow$ Find(G, W).
3. Let S be a random subset of S_0 of size $s = \Theta(\log n/\Delta^2)$. Let $v_1, \ldots, v_{n/2}$ be an ordering of $V - W$ such that $d_S(v_1) \geq d_S(v_2) \geq \cdots \geq d_S(v_{n/2})$. Let $D = \Theta(\sqrt{s \log n})$. If $\max_j\{d_S(v_j) - d_S(v_{j+1})\} < D$, output V and stop.
4. Let j be an index such that $d_S(v_j) - d_S(v_{j+1})$ is maximum, and let $S_1 = \{v_1, \ldots, v_j\}$.
5. Let S' be a random subset of S_1 of size s. Let $w_1, \ldots, w_{n/2}$ be an ordering of W such that $d_{S'}(w_1) \geq \cdots \geq d_{S'}(w_{n/2})$. Let j' be an index such that $d_{S'}(w_{j'}) - d_{S'}(w_{j'+1})$ is maximum, and let $S_2 = \{w_1, \ldots, w_{j'}\}$.
6. Output $S_1 \cup S_2$ and delete the vertices in $S_1 \cup S_2$ from G.
7. If G is not empty, goto 1.

Here and in the following, n is the number of vertices in the current (sub)graph G. Note that to avoid dependencies, we can not build the set S_2 by looking at edges whose endpoints are in W. Thus, we use a slightly different procedures to build S_1 and S_2.

Denote by $T(G)$ the running time of procedure Find on an input graph G. We assume that T satisfies $\sum_{i=1}^{l} T(G_i) \leq T(G)$ for any partition of G into vertex disjoint cluster collections $G_1, \ldots, G_l$.

Lemma 2. *If procedure* Find *returns w.h.p. a subcluster of size* $\Omega(\log n/\Delta^2)$*, then w.h.p., algorithm* Solve *solves the clustering problem. The running time of algorithm* Solve *is* $O(mT(G) + mn\log n/\Delta^2)$.

Proof. We shall prove that the failure rate in one iterations is n^{-c} for some constant c. The probability of failure of the algorithm is at most m times the probability of failure in one iteration, and since $m \leq n$ and c can be chosen sufficiently large, we obtain that the overall failure probability is $n^{-\Omega(1)}$. We shall show that if G consists of one cluster, then w.h.p. the algorithm stops at Step 3, and otherwise, if $S_0 \subseteq V_i$ then w.h.p., $S_1 \cup S_2 = V_i$.

W.l.o.g. we assume that $S_0 \subseteq V_1$. Since the building process of S was independent of the edges between $V - W$ and S, by Theorem 1 we have that w.h.p., $|d_S(v) - \mathrm{E}[d_S(v)]| < \frac{1}{2}D$ for all $v \in V - W$. For $v \in V_1$ we have that $\mathrm{E}[d_S(v)] = sp$, and for $v \notin V_1$ we have $\mathrm{E}[d_S(v)] = sr$. Hence, if G consists of one cluster, then w.h.p. $|d_S(v) - d_S(v')| < D$ for all $v, v' \in V - W$, and therefore, algorithm Solve stops at Step 3. Otherwise, for two vertices $v, v' \in V - W$, if either $v, v' \in V_1$ or $v, v' \notin V_1$ then w.h.p. $|d_S(v) - d_S(v')| < D$, and if $v \in V_1$ and $v' \notin V_1$ then w.h.p. $d_S(v) - d_S(v') > s\Delta - D \geq D$ where the last inequality is achieved by choosing appropriate constants for s, D so $s\Delta \geq 2D$. It follows that algorithm Solve does not stop at Step 3, and furthermore, $S_1 = V_1 - W$. Using similar arguments we show that $S_2 = V_1 \cap W$. □

Note that algorithm Solve requires the knowledge of Δ. However, if instead of Δ we use some known lower bound Δ' on Δ, then the algorithm remains correct, and the running time becomes $O(mT(G) + mn\log n/\Delta'^2)$.

Unfortunately, in order to build the procedure Find, we will need the following additional requirements on the input graph: (R1) $k \geq \Omega(\sqrt{n}\log^\alpha n/\Delta^\beta)$ for some $\alpha, \beta > 0$ and (R2) $p, r \in [\frac{1}{4}, \frac{3}{4}]$. While requirement of the type (R1) is understandable, we would like to get rid of (R2). To eliminate (R2), we transform the input graph to a graph that always satisfies the requirement using the following algorithm $\mathrm{Solve}_2(G)$:

1. Build a graph G' on the vertex set of G in the following way: For every pair of vertices u, v, add the edge (u, v) to G' with probability $\frac{3}{4}$ if (u, v) is an edge in G, and with probability $\frac{1}{4}$ otherwise.
2. Run Solve(G').

Lemma 3. *If procedure* Find *returns w.h.p. a subcluster of size* $\Omega(\log n/\Delta^2)$ *on graphs that satisfy (R1) and (R2), then w.h.p., algorithm* Solve_2 *solves the clustering problem on graphs that satisfy (R1). The running time of algorithm* Solve_2 *is* $O(mT(G) + mn\log n/\Delta^2)$.

Proof. Clearly, G' is a cluster graphs on the clusters $V_1, \ldots, V_m$ with edge probabilities $p' = \frac{3}{4}p + \frac{1}{4}(1-p) = \frac{1}{2}p + \frac{1}{4}$ and $r' = \frac{1}{2}r + \frac{1}{4}$. G' satisfies (R1) and furthermore, every cluster collection G'' of G' satisfies (R1) as the size of the smallest cluster in G'' is greater or equal to the size of the smallest cluster in G. As $p', r' \in [\frac{1}{4}, \frac{3}{4}]$ and $p' - r' = \Theta(\Delta)$, by Lemma 2 Solve(G') returns w.h.p. the correct partition. □

4 The Partition Procedure

In the previous section, we presented the basic structure of our algorithms. However, we still need to show how to build procedure Find, namely, how to find a subcluster of size $\Omega(\log n/\Delta^2)$. In this section we give the main ideas, and we will use these ideas in Section 5 to explicitly build procedure Find.

One of the key elements we use is the notion of imbalance which was also used in [8] and [5]. We use here a slightly different definition for imbalance than the one used in these papers: For two disjoint sets L, R of vertices of equal size, we define the L, R-*imbalance of* V_i, denoted $\mathrm{I}(V_i, L, R)$, by

$$\mathrm{I}(V_i, L, R) = \frac{|V_i \cap L| - |V_i \cap R|}{|L|}.$$

The *imbalance of* L, R is the maximum value amongst $\mathrm{I}(V_1, L, R), \ldots, \mathrm{I}(V_m, L, R)$, and the *secondary imbalance of* L, R is the second largest value (the secondary imbalance is equal to the imbalance if the maximum value appears more than once). We will show that given two sets L, R with "large" imbalance, and "small" secondary imbalance, it is possible to generate a large subcluster. Therefore, our goal is to generate the sets L, R with such properties.

Let $f\colon V \to Z^+$ be some function that depends on the edges of G. Since G is a random graph, we have that each $f(v)$ is a random variable. The function f is called a *cluster function* if $\{f(v)|v \in V\}$ are independent random variables, and for any $u, v \in T$ that belong to the same cluster, $f(u), f(v)$ have the same distribution. For example, $f(v) = d(v)$ is a cluster function.

If the values of f for vertices of one cluster are always greater than the values of f for the vertices of the other clusters, then we can easily generate a subcluster by picking $\Theta(\log n/\Delta^2)$ vertices with largest f-value. However, we are able to give such a cluster function f only when Δ is large. For smaller value of Δ, we are able to give a cluster function f with the following property: The expectation of $f(v)$ for vertices of one cluster, say V_1, is larger than the expectation of $f(v)$ for vertices in the other clusters, and the variance of $f(v)$ is "small" for all v. We will use this property of f in the following procedure Partition(G, T, f):

1. Begin with two empty sets L, R. Partition the vertices of T into pairs. For each pair v, w, place the vertex with larger f-value into L and the other into R, breaking ties randomly.
2. Return L, R.

We note that procedure Partition is very similar to the algorithm of Condon and Karp [5], and the cluster function of Lemma 5 is also used in [5].

Let $p_{ij}^{>}(f) = \mathrm{P}\left[f(v) > f(w)|v \in V_i, w \in V_j\right]$ (we will write $p_{ij}^{>}$ if f is clear from the context), and let $p_{ij}^{=}(f)$ and $p_{ij}^{<}(f)$ be the conditional probabilities that $f(v) = f(w)$ and $f(v) < f(w)$, respectively. Let $c_i(f) = 2a_i \sum_{j \neq i} a_j(p_{ij}^{>}(f) - p_{ij}^{<}(f))$. Suppose that L, R are the output of Partition(G, T, f) and denote $b_i = \mathrm{I}(V_i, L, R)$.

Theorem 3. *If T is a random set of size $2l$ and $k \geq \Omega(\log n)$, then w.h.p. $|b_i - c_i(f)| \leq O(\sqrt{a_i l^{-1} \log n}(1 + \sqrt{a_i m}))$ for all i.*

Proof. Since T is a random set, the process of choosing T and then partitioning T into pairs, is equivalent to the process of selecting vertices $u^1, \ldots, u^{2l}$ from V (one after another) and pairing u^{2t-1} and u^{2t} for each t. Denote $a_j^r = \mathrm{P}[u_r \in V_j]$.

Denote by I_i^1 (resp., I_i^2) the number of indices t for which $u^{2t-1} \in V_i$, $u^{2t} \notin V_i$, and u^{2t-1} (resp., u^{2t}) is inserted into L. Similarly, denote by I_i^3 (I_i^4) the number of indices t for which $u^{2t-1} \notin V_i$, $u^{2t} \in V_i$, and u^{2t} (u^{2t-1}) is inserted into L. Clearly, $b_i = (I_i^1 + I_i^3 - I_i^2 - I_i^4)/l$. We will now give an estimate on I_i^1 for some fixed i. Denote by X_t the indicator variable to the event that $u^{2t-1} \in V_i$, $u^{2t} \notin V_i$, and u^{2t} is inserted into L. Let $p_j = p_{ij}^{>}(f) + \frac{1}{2}p_{ij}^{=}(f)$. By Lemma 1, we have w.h.p. that $|a_j - a_j^r| \leq O(\sqrt{a_j n^{-1} \log n})$ for all j and r. Therefore, $\mathrm{P}[X_t = 1] = a_i^{2t-1} \sum_{j \neq i} a_j^{2t} p_j \leq a_i \sum_{j \neq i} a_j p_j + O(\sqrt{a_i n^{-1} \log n}(1 + \sqrt{a_i m})) \doteq L$. Hence, we have that the random variable $I_i^1 = \sum_{t=1}^{l} X_t$ is dominated by a random variable Y, where Y has binomial distribution with l experiments and success probability $\min(1, L)$. Thus, by Theorem 1, we have w.h.p. that $Y \leq \mathrm{E}[Y] + O(\sqrt{\mathrm{E}[Y] \log n}) \leq l a_i \sum_{j \neq i} a_j p_j + O(\sqrt{a_i l \log n}(1 + \sqrt{a_i} m))$. It follows that w.h.p. $I_i^1 \leq l a_i \sum_{j \neq i} a_j p_j + O(\sqrt{a_i l \log n}(1 + \sqrt{a_i} m))$.

Similar arguments gives a lower bound on I_i^1, and lower and upper bounds on I_i^2,I_i^3, and I_i^4 Combining these bounds gives the desired bound on b_i. □

In the next sections, we use procedure Partition in a slightly different scenario than in Theorem 3: We have a random set S of vertices, and then the set T is randomly chosen from $V - S$. It is easy to verify that the result of Theorem 3 is also valid here.

We now give two cluster functions, which will be used later in our algorithms. The first function, given in Lemma 4, will be used for building initial sets L_0, R_0 with large imbalance, and the second function, in Lemma 5, will be used to iteratively build a series of sets L_i, R_i with increasing imbalance.

Lemma 4. *Let u be some vertex from $V_1 - T$, and define $f(v) = d_{\{u\}}(v)$ for all $v \in T$. Then $c_1(f) = 2a_1(1 - a_1)\Delta$ and $c_i(f) = -2a_1 a_i \Delta$ for all $i > 1$.*

Proof. Clearly, $p_{1j}^{>} = p(1 - r)$ and $p_{1j}^{<} = (1 - p)r$, so $c_1 = 2a_1 \sum_{j \neq i} a_j(p(1 - r) - (1 - p)r) = 2a_1(1 - a_1)\Delta$. For $i, j > 1$, $p_{ij}^{>} = p_{ij}^{<} = r(1 - r)$ and $p_{i1}^{>} - p_{i1}^{<} = -(p_{1i}^{>} - p_{1i}^{<}) = -\Delta$, and therefore, $c_i = -2a_1 a_i \Delta$. □

Lemma 5. *Let L', R' be two disjoint subsets of $V - T$ of size l' each, and define $f(v) = d_{L'}(v) - d_{R'}(v)$ for all $v \in T$. Let $b_i' = \mathrm{I}(V_i, L', R')$. Suppose that $b_1' \geq b_2' \geq \cdots \geq b_m'$, $p, r \in [\frac{1}{4}, \frac{3}{4}]$, $k \geq \Omega(\sqrt{n \log n})$, and $|b_i' - b_j'| \leq \alpha/\Delta\sqrt{l'}$ for all i, j where $\alpha \leq 1$. Furthermore, suppose that the building process of L', R' is independent of the edges between $L' \cup R'$ and T. Let $\beta = \Gamma + \sqrt{\frac{1/m + \Gamma}{l'} \log n}$ and $\gamma = \sqrt{\frac{2}{\pi}/(\frac{1}{2m}p(1 - p) + (1 - \frac{1}{2m})r(1 - r))}$. Then, w.h.p.,*

1. $c_1(f) \geq \gamma\Delta\sqrt{l'}a_1(1-O(\beta\Delta)-\frac{2}{9}\alpha^2)(b'_1-\sum_{j=1}^m a_j b'_j)-3\frac{a_1}{\sqrt{l'}}$.
2. $c_i(f) \leq \gamma\Delta\sqrt{l'}a_i\left(b'_i-\sum_{j=1}^m a_j b'_j+(O(\beta\Delta)+\frac{2}{9}\alpha^2)(b'_1-b'_m)\right)+3\frac{a_i}{\sqrt{l'}}$ *and* $c_i(f) \geq \gamma\Delta\sqrt{l'}a_i\left(b'_i-\sum_{j=1}^m a_j b'_j-(O(\beta\Delta)+\frac{2}{9}\alpha^2)(b'_1-b'_m)\right)-3\frac{a_i}{\sqrt{l'}}$ *for all* $i>1$.

Proof. To bound c_1, we need to estimate $p^{>}_{1j}-p^{<}_{1j}$ for some fixed j. We fix some $v \in V_1 \cap T$ and $w \in V_j \cap T$. Then we write $f(v)-f(w)$ as a sum of $4l'$ independent random variables, where each variable depends on one edge in the graph. The expectation of $f(v)-f(w)$ is $\Delta l'(b'_1-b'_j)$, and using Theorem 2 we obtain that $p^{>}_{1j}-p^{<}_{1j} \geq 2\Phi\left(\Delta l'(b'_1-b'_j)/\sqrt{B_{1j}}\right)-1-\frac{3}{2\sqrt{l'}}$ for B_{1j} whose value is close to $B=2\frac{l'}{m}p(1-p)+(4l'-2\frac{l'}{m})r(1-r)$. The lower bound on c_1 follows. The bounds on c_i are proved similarly. □

5 Finding a Subcluster

In this section we present an algorithm for finding a subcluster in the case of almost equal sized clusters: The algorithm requires that $\Gamma \leq O(1/m\log n)$. The procedure Find(G,V') is as follows:

1. Let $\hat{l}=\Theta(m^2\Delta^{-2}\log^2 n+\Delta^2 m^{-1}\log^5 n)$, $l=\Theta(\hat{l}/\log n)$, $l'=\Theta(\hat{l}/\log^2 n)$, $s=\Theta(m\Delta^{-2}\log n)$, and $D=\Theta(\sqrt{\hat{l}\log n})$. Let $l_t=l$ for $0 \leq t \leq 2$ and $l_t=l'$ for $t \geq 3$.
2. Randomly select disjoint sets of vertices $S, \hat{W}, W_0$ from V' of sizes $s, 2\hat{l}, 2l_0$, respectively.
3. Randomly select some unchosen vertex u from V'.
4. $\hat{L}_0, \hat{R}_0 \leftarrow$ Partition$(G, \hat{W}, d_{\{u\}})$ and $L_0, R_0 \leftarrow$ Partition$(G, W_0, d_{\{u\}})$.
5. For $t=0,1,\ldots,\log n$ do:
   a) Let $v_1,\ldots,v_s$ be an ordering of S such that $f(v_1) \geq f(v_2) \geq \cdots \geq f(v_s)$, where $f(v)=d_{\hat{L}_t}(v)-d_{\hat{R}_t}(v)$. If $\max_j\{f(v_j)-f(v_{j+1})\} \geq D$, then let j be the first index for which $f(v_j)-f(v_{j+1}) \geq \frac{1}{3}D$, and return $\{v_1,\ldots,v_j\}$. Otherwise, continue.
   b) Randomly select a set W_{t+1} from V' of $2l_{t+1}$ unchosen vertices.
   c) $\hat{L}_{t+1}, \hat{R}_{t+1} \leftarrow$ Partition$(G, \hat{W}, d_{L_t}-d_{R_t})$.
   d) $L_{t+1}, R_{t+1} \leftarrow$ Partition$(G, W_{t+1}, d_{L_t}-d_{R_t})$.
6. Output 'Failure'.

Note that in the first three iterations of Step 5, the sets L_t, R_t are bigger than in the rest of the steps. The reason is that in this steps, the imbalance is small, so in order to get a small *relative* difference between the imbalances and the expected imbalances, we need to use bigger sets (see Theorem 3). We also note that the $\Omega(\log n)$ size difference between $\hat{L}_t, \hat{R}_t$ and L_t, R_t has an important role: It allows us to ensure that the requirement $|b'_i-b'_j| \leq \alpha/\Delta\sqrt{l'}$ of Lemma 5 is satisfied for a small α.

Denote $b_i^t = \mathrm{I}(V_i, L_t, R_t)$, $\hat{b}_i^t = \mathrm{I}(V_i, \hat{L}_t, \hat{R}_t)$, $r_i^t = b_i^t/b_1^t$, and $\hat{r}_i^t = \hat{b}_i^t/\hat{b}_1^t$. In the following, we assume w.l.o.g. that the vertex u chosen in Step 3 is from V_1. Let T be the value of t when the procedure Find stops. For simplicity, we assume that $T \geq 3$.

Lemma 6. *Suppose that* $\Gamma \leq O(1/m \log n)$, $k \geq \Omega(\Delta^{-1}\sqrt{n}\log n)$, *and* $p, r \in [\frac{1}{4}, \frac{3}{4}]$. *Then, w.h.p.,*

1. *For all* i, $|b_1^t - \hat{b}_1^t| \leq \hat{o}(\frac{\Delta}{m})$ *for* $0 \leq t \leq 2$, *and* $|b_1^t - \hat{b}_1^t| \leq \hat{o}(\frac{\Delta}{m}\sqrt{\log n})$ *for* $3 \leq t \leq T$.
2. $b_1^0, \hat{b}_1^0 \geq (1 - \hat{o}(1))\frac{\Delta}{m}$.
3. *If* $m \geq 3$, *then for all* $i > 1$, $-\frac{1}{2} - \hat{o}(1) \leq r_i^0, \hat{r}_i^0 \leq \hat{o}(1)$.
4. *For* $1 \leq t \leq 3$, $b_1^t, \hat{b}_1^t \geq \log^{t/2} n \cdot \frac{\Delta}{m}$.
5. *For* $4 \leq t \leq T$, $b_1^t, \hat{b}_1^t \geq 2^{t-3}\log^{3/2} n \cdot \frac{\Delta}{m}$.
6. *If* $m \geq 3$, *then for all* $1 \leq t \leq 3$ *and all* $i > 1$, $(1+\delta)^t(\min(0, r_i^0) - \delta t) \leq r_i^t \leq (1+\delta)^t(\max(0, r_i^0) + \delta t)$ *and* $(1+\delta)^t(\min(0, \hat{r}_i^0) - \delta t) \leq \hat{r}_i^t \leq (1+\delta)^t(\max(0, \hat{r}_i^0) + \delta t)$ *where* $\delta \leq \hat{o}(1)$.
7. *If* $m \geq 3$, *then for all* $4 \leq t \leq T$ *and all* $i > 1$, $(1+\delta')^{t-3}(\min(0, r_i^3) - \delta'(t-3)) \leq r_i^t \leq (1+\delta')^{t-3}(\max(0, r_i^3) + \delta'(t-3))$ *and* $(1+\delta')^{t-3}(\min(0, \hat{r}_i^3) - \delta'(t-3)) \leq \hat{r}_i^t \leq (1+\delta')^{t-3}(\max(0, \hat{r}_i^3) + \delta'(t-3))$ *where* $\delta' \leq \hat{o}(1/\log n)$.

Note that the bound $k \geq \Omega(\Delta^{-1}\sqrt{n}\log n)$ implies that the total number of vertices selected by Find is at most $n/2$. Also, as $T \leq \log n$, we have from (6) and (7) that $-\frac{1}{2} - \hat{o}(1) \leq r_i^t \leq \hat{o}(1)$ for all t when $m \geq 3$.

Proof. Item 1 follows from Theorem 3. Items 2 and 3 follows from Lemma 4.

To prove item 4, we shall prove that $b_1^{t+1} \geq 2\sqrt{\log n} \cdot b_1^t$ for some fixed $0 \leq t \leq 2$. Assume w.l.o.g. that $b_1^t \geq b_2^t \geq \cdots \geq b_m^t$. By Theorem 1, w.h.p., $|f(v) - \mathrm{E}[f(v)]| < \frac{1}{2}D$ for all $v \in S$, where $\mathrm{E}[f(v)] = \Delta\hat{l}\hat{b}_i^t$. Since the algorithm didn't stop at iteration t, we have that $\Delta\hat{l}(\hat{b}_1^t - \hat{b}_2^t) \leq 2D$, because otherwise, the difference between the minimum value of $f(v)$ for $v \in V_1 \cap S$ and the maximum value of $f(v')$ for $v' \in S - V_1$ would be at least D. Thus, for all i, j, $|b_i^t - b_j^t| \leq (2 + \hat{o}(1))(\hat{b}_1^t - \hat{b}_2^t) \leq \alpha/\Delta\sqrt{l}$ where $\alpha \leq \hat{o}(1)$. It follows (using Lemma 5) that $b_1^{t+1} \geq (1 - \hat{o}(1))Fb_1^t$ where $F = \gamma\Delta\sqrt{l}/m$ (γ is defined in Lemma 5). This proves item 4. The proof of item 5 is similar.

Using Lemma 5, we show that $(\min(0, r_i^t) - \hat{o}(1))Fb_1^t \leq b_i^{t+1} \leq (\max(0, r_i^t) + \hat{o}(1))Fb_1^t$. Thus, $(\min(0, r_i^t) - \delta)/(1-\delta) \leq r_i^{t+1} \leq (\max(0, r_i^t) + \delta)/(1-\delta)$ for some $\delta \leq \hat{o}(1)$, and item 6 follows by induction. The proof of item 7 is similar. □

Lemma 7. *Suppose that* $\Gamma \leq O(1/m \log n)$, $k \geq \Omega(\Delta^{-1}\sqrt{n}\log n)$, *and* $p, r \in [\frac{1}{4}, \frac{3}{4}]$. *Then, w.h.p., procedure* Find *returns a subcluster of size* $\Omega(m \log n/\Delta^2)$. *The running time of procedure* Find *is* $O(m^3\Delta^{-4}\log^3 n \cdot (m + \log n) + \log^9 n)$.

Proof. By Lemma 6, for all $t \leq T$, $b_1^t > 2^t/n$. Since $b_1^t \leq 1$ by definition, we conclude that $T < \log n$, namely procedure Find stops at Step 5a. We now look at the values of f at the last iteration. W.l.o.g. assume that $\hat{b}_1^T \geq \hat{b}_2^T \geq \cdots \geq \hat{b}_m^T$.

By Theorem 1, w.h.p., $|f(v)-\mathrm{E}[f(v)]| < \frac{1}{6}D$ for all $v \in S$, where $\mathrm{E}[f(v)] = \Delta\hat{l}\hat{b}_i^T$ for $v \in V_i$. As $\max_j\{f(v_j) - f(v_{j+1})\} \geq D$, we have that there is an index i such that $\Delta\hat{l}\hat{b}_i^T - \Delta\hat{l}\hat{b}_{i+1}^T \geq \frac{2}{3}D$. By Lemma 6, $\Delta\hat{l}(\hat{b}_1^T - \hat{b}_2^T) \geq \Delta\hat{l}(\hat{b}_i^T - \hat{b}_{i+1}^T) \geq \frac{2}{3}D$. Therefore, $\min_{v\in V_1\cap S} f(v) - \max_{v\in S-V_1} f(v) > \frac{1}{3}D$. Furthermore, for $v, v' \in V_1 \cap S$, $|f(v)-f(v')| < \frac{1}{3}D$. It follows that $\{v_1, \ldots, v_j\} = V_1 \cap S$, and in particular, $\{v_1, \ldots, v_j\}$ is a subcluster. Finally, by Lemma 1, $|V_1 \cap S| = \Omega(\log n/\Delta^2)$. □

Note that procedure Find requires the knowledge of m and Δ. However, if we use take $\hat{l} = \Theta(M^2 \log^2 n + \log^5 n)$ and $s = \Theta(M^2 \log n)$ where $M \geq m/\Delta$, then procedure Find remains correct, and its time complexity becomes $O(M^4 \log^4 n + \log^{10} n)$. Since the bound $k \geq \Omega(\Delta^{-1}\sqrt{n}\log n)$ implies $m/\Delta \leq O(\sqrt{n}/\log n)$, one can use $M = \Theta(\sqrt{n}/\log n)$, and the time complexity will be $O(n^2)$.

Combining Lemma 7 and Lemma 3 gives the following theorem:

Theorem 4. *The above algorithm solves w.h.p. the clustering problem when* $\Gamma \leq O(1/m\log n)$ *and* $k \geq \Omega(\Delta^{-1}\sqrt{n}\log n)$. *The running time of the algorithm is* $O(m^4\Delta^{-4}\log^3 n \cdot (m + \log n) + m\Delta^{-2}n\log n)$.

To handle the case where cluster sizes are not almost equal, we have to consider the case of very large clusters, and the vertex u in Step 3 of procedure Find must be chosen from a large cluster. Moreover, the analysis of the analogue of Lemma 6 is less tight, which allows only a constant number of iterations of procedure Find.

References

1. A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *J. of Computational Biology*, 6:281–297, 1999.
2. R. B. Boppana. Eigenvalues and graph bisection: An average-case analysis. In *Proc. 28th Symposium on Foundation of Computer Science (FOCS 87)*, pages 280–285, 1987.
3. T. Carson and R. Impagliazzo. Hill-climbing finds random planted bisections. In *Proc. Twelfth Symposium on Discrete Algorithms (SODA 01)*, pages 903–909. ACM press, 2001.
4. H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Statis.*, 23:493–507, 1952.
5. A. E. Condon and R. M. Karp. Algorithms for graph partitioning on the planted partition model. *Lecture Notes in Computer Science*, 1671:221–232, 1999.
6. M. E. Dyer and A. M. Frieze. The solution of some random NP-hard problems in polynomial expected time. *J. of Algorithms*, 10(4):451–489, 1989.
7. U. Feige and J. Kilian. Heuristics for semirandom graph problems. *J. of Computer and System Sciences*, To appear.
8. M. Jerrum and G. B. Sorkin. Simulated annealing for graph bisection. In *Proc. 34th Symposium on Foundation of Computer Science (FOCS 93)*, pages 94–103, 1993.
9. A. Jules. *Topics in black box optimization.* PhD thesis, U. California, 1996.
10. V. V. Petrov. *Sums of independent random variables.* Springer-Verlag, 1975.
11. Z. Yakhini. Personal communications.

Δ-List Vertex Coloring in Linear Time

San Skulrattanakulchai

Department of Computer Science, University of Colorado at Boulder,
Boulder CO 80309 USA
skulratt@cs.colorado.edu

Abstract. We present a new proof of a theorem of Erdős, Rubin, and Taylor, which states that *the list chromatic number (or choice number) of a connected, simple graph that is neither complete nor an odd cycle does not exceed its maximum degree* Δ. Our proof yields the first-known linear-time algorithm to Δ-list-color graphs satisfying the hypothesis of the theorem. Without change, our algorithm can also be used to Δ-color such graphs. It has the same running time as, but seems to be much simpler than, the current known algorithm, due to Lovász, for Δ-coloring such graphs. We also give a specialized version of our algorithm that works on *subcubic* graphs (ones with maximum degree three) by exploiting a simple decomposition principle for them.

1 Introduction

This paper presents efficient algorithms for a list vertex coloring problem. Our main contribution is a new proof of a theorem of Erdős, Rubin, and Taylor [1]. Our proof yields the first-known linear-time algorithm to Δ-list-color graphs that do not contain a complete graph or an odd cycle as a connected component. Without change, our algorithm can also be used to Δ-color such graphs. It has the same running time as, but seems to be much simpler than, the current known algorithm for Δ-coloring such graphs. We also give a specialized version of our algorithm that works on *subcubic* graphs. (A subcubic graph has maximum degree 3.) The notations and the definitions of the graph terms we use are given in Sect. 1.1. We now discuss the problem we solve in detail.

In the *list vertex coloring problem* [2], we are given a graph every vertex of which has been assigned its own list of colors as part of the input. Our task is to assign to every vertex a color chosen from its color list so that adjacent vertices receive distinct colors. A graph is *list-colorable* if the task is possible. The list vertex coloring problem is a generalization of the usual *vertex coloring problem* [2]. In the usual problem, we are given a graph and a constant number k of colors. Our task is to assign to every vertex a color chosen from the given k colors so that adjacent vertices receive distinct colors. A graph is *k-colorable* if the task is possible. To *k-color* the graph is to find such a color assignment. The *chromatic number* of a graph is the least k for which it is k-colorable. A graph is *k-list-colorable* if it is list-colorable given any list assignment as long as the size of each list is at least k. To find such a color assignment is to *k-list-color*

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 240–248, 2002.

the graph. The *list chromatic number* (or *choice number*) of a graph is the least k for which it is k-list-colorable. More generally, for any list size assignment $f : V \to \mathbb{N}$, we say that a graph is *f-choosable* if it is list-colorable given any list assignment as long as the list size of any vertex v is at least $f(v)$. For example, a d-choosable graph is one that is list-colorable whenever every vertex's color list contains at least as many colors as its degree.

Observe that a k-list-colorable graph is always k-colorable since any algorithm for k-list-coloring can be used for k-coloring: simply assign the same list of k colors to every vertex of the input graph, and run the k-list-coloring algorithm on it. Note also that a graph is list-colorable if and only if each of its connected components is. Therefore, one needs only consider list-colorability of connected graphs.

A *Brooks graph* is a connected graph that is neither complete nor an odd cycle. In [1] Erdős et al. prove that a connected graph is d-choosable if and only if at least one of its biconnected components is a Brooks graph. Consequently, they obtain the list version of Brooks' Theorem: *the list chromatic number (or choice number) of a Brooks graph does not exceed its maximum degree.* In other words, a Brooks graph is Δ-list-colorable. In their proof they check biconnectivity of the graph at every inductive step following a sequence of vertex and edge deletions. This is an expensive operation: no algorithm is known that can dynamically compute graph biconnectivity under vertex and edge deletions in $O(m+n)$ time. We give a new proof of their theorem and also derive the first $O(m + n)$-time Δ-list-coloring algorithm. Both their and our proofs rely on existence of some special types of induced subgraphs in a biconnected Brooks graph. They identify two types of special subgraphs. By adding another type to their list we are able to obtain a more efficient algorithm. Our success rests on the simple fact that it is easier to find objects of one's desire when there are more of them around.

Our algorithm naturally specializes to the problem of finding a coloring in the original Brooks' Theorem [3]: *the chromatic number of a Brooks graph does not exceed its maximum degree.* In other words, a Brooks graph is Δ-colorable. The current known efficient algorithm for Δ-coloring Brooks graphs is due to Lovász [4]. (See also Problem 9.13, pp. 67 in [5].) By using the ideas given in Sect. 2.2, it is possible to implement his algorithm to make it run in $O(m + n)$ time. Lovász's algorithm computes a separating pair of vertices [6]. In contrast, our algorithm explores the graph in search of an occurrence of any one of a few simple unavoidable patterns, whose existence gives a straightforward coloring algorithm for the whole graph. Both Lovász's and our algorithms call upon a subroutine to compute biconnected components. This is fine since there exist many simple biconnected components algorithms in the literature. (See [7,8], for example.) Again, our algorithm is simpler than Lovász's in this respect. All it needs from the biconnectivity computation is one biconnected component. In contrast, Lovász's algorithm has to construct the so-called block-cutpoint tree from all the biconnected components. Another difference between the two algorithms concerns recoloring. Lovász's algorithm colors each biconnected compo-

nent separately, then put these colorings together by appropriately "permuting colors" in each component. Our algorithm never recolors.

A class of graphs that has proven itself to be of special interest is the class of subcubic graphs [9]. References [10,11] show that a simple decomposition principle for subcubic graphs give efficient algorithms for solving various coloring problems on them. We will exploit this principle to obtain a version of our algorithm that is specialized to subcubic Brooks graphs and that is simpler than the general version. It even does away with the biconnectivity computation altogether.

In the rest of this section we discuss terminology. In Sect. 2 we develop our proof and algorithm for the theorem of Erdős et al. on general graphs. In Sect. 3 we develop an algorithm for the case of subcubic graphs.

1.1 Terminology

For any positive integer n, the set of all positive integers not greater than n is denoted $[n]$.

We follow the terminology of [12] and consider only undirected graphs without parallel edges. By default, $G = (V, E)$ denotes the graph under consideration with $|V| = n$ and $|E| = m$. The *degree of a vertex* v is usually denoted $d(v)$, and also as $d_G(v)$ whenever G needs to be distinguished from some other graph also under consideration. The *minimum (maximum) degree* of the graph is denoted δ (Δ). By a *path* we mean one with no repeated vertices. A cycle or path is *even (odd)* if it has an even (odd) number of edges. A *wheel* W_k consists of a cycle C_k, and a *hub* vertex w that is not on the cycle but is adjacent to every vertex on the cycle. The cycle vertices are the *rim*; the k edges joining the hub w to the rim are called *spokes*. A *whel* is a wheel W_k with ℓ spokes missing, where $1 \leq \ell < k-1$. In other words, a whel can be obtained from a wheel by removing at least one spoke while keeping at least two spokes. A *theta graph* $\theta(a, b, c)$ consists of 3 internally disjoint paths P_{a+1}, P_{b+1}, and P_{c+1} connecting 2 end vertices, where the path lengths a, b, and c satisfy $0 < a \leq b \leq c$. A *diamond* is the complete graph K_4 with one edge missing. It is at the same time the smallest whel and the theta graph $\theta(1, 2, 2)$. Let H be a subgraph of G. A path *avoids* H if it uses no edge of H. A vertex w is called a *neighbor* of H if w is adjacent in G to some vertex of H but is itself not a vertex of H. We always use the term *"induced"* to mean "induced by its vertex set." For example, C is an induced cycle if C is a subgraph of G isomorphic to a cycle and C is induced by its own vertex set.

We use the term *available color* at various places in the description of our algorithm. During the execution of the algorithm, a *color c is available for a vertex v* if it is in the color list of v and no vertex adjacent to v is already colored c.

In this paper, the term *linear* means $O(m + n)$, i.e., linear in the size and order of the input graph.

2 General Graphs

2.1 Lemmas and Theorem

Lemma 1 (Cycle List Coloring Lemma). *If every vertex of a cycle C has a list of at least 2 colors, then C is list-colorable unless C is odd and every vertex has the same list of 2 colors.*

Proof. Let C have k vertices. First suppose some vertex has more than 2 colors in its list. Label it v_1 and label the other vertices $v_2, \ldots, v_k$ according as they appear on the tour of C starting from v_1. Choose a color c in the list of v_1 so that v_k has at least 2 available colors if v_1 is colored c. Assign color c to v_1. Now color $v_2, \ldots, v_k$ in that order, choosing any color available for each.

Next suppose every vertex has exactly 2 colors in its list but some lists are different. Choose two consecutive vertices with distinct lists and label them v_1 and v_k. Label the other vertices $v_2, \ldots, v_{k-1}$ according as they appear on the tour of C starting from v_1 and ending at v_k. Now assign colors the same way as previous case.

Last suppose C is even and all lists are the same list of 2 colors. Starting from any vertex, tour the cycle and color each vertex in the order visited by the tour using any color available for each.

It is easy to check that in all three cases every vertex has some available color at the time when the algorithm is about to color it. □

Lemma 2 (Theta Graph List Coloring Lemma). *Any theta graph $\theta(a, b, c)$ is list-colorable if each of its two end vertices has a list of at least 3 colors, and each of the rest of its vertices has a list of at least 2 colors.*

Proof. We may assume that all inequalities on the sizes of the lists hold with equality. Let s, t be the end vertices connected by the three paths P_{a+1}, P_{b+1} and P_{c+1}. Let $P_{a+1} = (s, u_1, \ldots, u_{a-1}, t)$, $P_{b+1} = (s, v_1, \ldots, v_{b-1}, t)$, and $P_{c+1} = (s, w_1, \ldots, w_{c-1}, t)$. First color s by a color from its list that is different from any color in the list of v_1. This is always possible since s has 3 colors in its list while v_1 has only 2. Then color $w_1, w_2 \ldots, w_{c-1}$, $u_1, u_2, \ldots, u_{a-1}$, t, $v_{b-1}, v_{b-2}, \ldots, v_1$ in that order, using any color available for each. □

Lemma 3. *A graph that is connected but not regular is Δ-list-colorable.*

Proof. Let v_1 be a vertex whose degree is less than Δ. For $j \leftarrow 2$ to n set v_j to be any vertex in $V \setminus \{v_1, \ldots, v_{j-1}\}$ adjacent to some vertex in $\{v_1, \ldots, v_{j-1}\}$. Color $v_n, \ldots, v_1$ in that order, using any color available for each. This procedure never fails since for $j = n, \ldots, 2$ vertex v_j is adjacent to some uncolored v_i, where $i < j$, and $d(v_j) \leq \Delta$ and v_j has $\geq \Delta$ colors in its list. Moreover, v_1 is colorable because $d(v_1) < \Delta$ and v_1 has $\geq \Delta$ colors in its list. □

Lemma 4. *A connected graph G is Δ-list-colorable if it contains any of (i) an even cycle, (ii) a whel, or (iii) a theta graph, as an induced subgraph.*

Proof. First suppose that G contains an induced even cycle $C = \langle v_1, \ldots, v_{2k} \rangle$. For $j \leftarrow 2k+1$ to n set v_j to be any vertex in $V \setminus \{v_1, \ldots, v_{j-1}\}$ adjacent to some vertex in $\{v_1, \ldots, v_{j-1}\}$. Color $v_n, \ldots, v_{2k+1}$ in that order, using any color available for each; then color C using its vertices' lists of available colors by invoking the *Cycle List Coloring Lemma*. This procedure never fails since for $j = n, \ldots, 2k+1$ vertex v_j is adjacent to some uncolored v_i, where $i < j$, and $d(v_j) = \Delta$ and v_j has $\geq \Delta$ colors in its list. Moreover, C is colorable because right after v_{2k+1} is colored, each vertex on the even cycle C has a list of at least 2 available colors.

Next suppose that G contains an induced whel consisting of w as the hub vertex and cycle $C = \langle v_1, \ldots, v_k \rangle$ as the rim. Set $v_{k+1} \leftarrow w$. For $i \leftarrow k+2$ to n set v_i to be some vertex in $V \setminus \{v_1, \ldots, v_{i-1}\}$ adjacent to some vertex in $\{v_1, \ldots, v_{i-1}\}$. Color $v_n, \ldots, v_{k+2}$ in that order, using any color available for each. Choose a color for v_{k+1} so that the vertices in the resulting uncolored cycle C do not have the same list of 2 available colors. This can always be done since v_{k+1} is adjacent to more than one but not all vertices of C and v_{k+1} has at least 2 available colors. Now cycle C is colorable using its vertices' lists of available colors by the *Cycle List Coloring Lemma*.

Now suppose that G contains an induced theta subgraph $\theta(a, b, c)$. There are $k = a + b + c - 1$ vertices in the θ graph. Label them $v_1, \ldots, v_k$. For $i \leftarrow k+1$ to n set v_i to be some vertex in $V \setminus \{v_1, \ldots, v_{i-1}\}$ adjacent to some vertex in $\{v_1, \ldots, v_{i-1}\}$. Color $v_n, \ldots, v_{k+1}$ in that order, using any color available for each. The uncolored vertices of the theta graph have lists of available colors satisfying the hypotheses of the *Theta Graph List Coloring Lemma*, so they are colorable using colors from these lists. □

Lemma 5. *Let G be a biconnected Brooks graph. Then G contains one of (i) an even cycle, (ii) a whel, or (iii) a theta graph, as an induced subgraph.*

Proof. Since G is biconnected we see that $\Delta \geq \delta \geq 2$ and G contains some cycle, and thus contains some induced cycle. We are done if any of its induced cycles is even. So assume that every induced cycle is odd.

First suppose that G contains a triangle. Let K be a maximal clique containing this triangle. Then K is necessarily a proper subgraph of G since G is not complete. Thus K has some neighbor. Say that $v_1, \ldots, v_k$ are the vertices of K. No neighbor w of K can be adjacent to all the v_i's since otherwise w and all the v_i's would form a clique containing K, contradicting maximality of K. Now if any neighbor w is adjacent to v_i, v_j but not to v_ℓ, where i, j, ℓ are all distinct, then we get an induced diamond on the vertices w, v_i, v_j, v_ℓ, and we are done. So assume that every neighbor is adjacent to exactly one vertex in K. Pick a vertex v_i adjacent to some neighbor of K. Since G is biconnected, there is a shortest path P avoiding K and connecting v_i to some other vertex v_j. Let

ℓ be such that $1 \leq \ell \leq k$ and $\ell \neq i, j$. Vertex v_ℓ together with the vertices on P then induce a whel with v_ℓ as its hub.

Next suppose that G contains an induced odd cycle $C = \langle v_1, \ldots, v_k \rangle$ that is not a triangle. Cycle C cannot be the whole of G since G is not an odd cycle. So C has some neighbor. If some neighbor w of C is adjacent to every v_i, then $\{w, v_1, v_2, v_3\}$ induces a diamond, and we are done. If some neighbor w of C is adjacent to more than one but not all vertices of C, then w and the vertices of C induce a whel, and we are done. So assume that every neighbor of C is adjacent to exactly one vertex of C. Pick a vertex v_i adjacent to some neighbor of C. Since G is biconnected, there is a shortest path P avoiding C and connecting v_i to some other vertex v_j. The vertices of $C \cup P$ then induce a theta graph. □

Theorem 1. *A Brooks graph is Δ-list-colorable.*

Proof. Let G be a Brooks graph. If G is not regular, then it is Δ-list-colorable by Lemma 3. Suppose G is regular and consider its the block-cutpoint tree T. Let H be a biconnected component in G corresponding to a leaf in T. If $H = G$ then H is a Brooks graph by assumption. If $H \neq G$, let v be the only vertex in H that is also a cutpoint in G. Since H corresponds to a leaf in T and v is a cutpoint in the regular graph G, it follows that $d_H(v) < d_H(w)$ for any vertex w in H distinct from v. So H is not regular; and hence cannot be either a complete subgraph or an odd cycle. Therefore, H is a Brooks graph. By Lemma 5, the subgraph H contains either an induced even cycle, or an induced whel, or an induced theta graph; and so does G since it is a supergraph of H. So G is Δ-list-colorable by Lemma 4 since it is also connected. □

2.2 Algorithm

We assume the following of our input. The input Brooks graph $G = (V, E)$ is represented as adjacency lists (see [13]). Each vertex has exactly Δ colors in its color list. (If not, we can throw away all colors in excess of Δ without affecting Δ-list-colorability of G.) Vertices, edges, and colors are numbered using consecutive integers starting from 1. So $V = [n]$, $E = [m]$, and the highest possible color is $n\Delta$. Each color list C_i ($i \in [n]$) is represented as a linked list.

The proof of Theorem 1 suggests the following high-level algorithm for Δ-list-coloring G. If G is not regular, apply the algorithm in the proof of Lemma 3 and we are done. If G is regular, do the following. Execute some biconnected components algorithm [7,8] on G but stop the algorithm as soon as a biconnected component corresponding to a leaf in the block-cutpoint tree of G is identified. So we can stop the algorithm of [7] or [8] as soon as it discovers the first biconnected component. Call this component H. Apply the algorithm in the proof of Lemma 5 on component H to extract an induced even cycle, or an induced whel, or an induced theta graph. Now apply the algorithm from the appropriate part of the proof of Lemma 4.

The biconnected components algorithms [7,8] are known to take $O(m + n)$ time. We will describe how to accomplish these two tasks.

Task A. To implement the algorithm in the proof of Lemma 5 so that it takes $O(m+n)$ time.

Task B. To implement the algorithms in the proof of Lemmas 3 & 4 so that they take $O(m+n)$ time.

Task A. To find an induced cycle, use a modified depth-first search (**DFS**) to explore the graph. Put in the language of the tree-based view of **DFS** [13], we make sure any back edge is explored before any tree edge. This can be done by going through the adjacency list of the currently active vertex v once. If back edges are found and w is the vertex on the recursion stack closest and adjacent to v via a back edge, then all the vertices on the tree path from w to v induce a cycle. If no back edges are found, then continue exploring the graph in the depth-first manner.

Here is how to find a maximal clique containing a triangle. Use a boolean array to keep track of the vertices belonging to the current clique. Also, keep a count of the number of vertices in the current clique. Use another boolean array to keep track of processed vertices. Put all neighbors of the current clique in the queue. Repeat the following until the queue is empty. Delete a vertex v from the queue. If v has already been processed, continue checking the queue. Otherwise, process v as follows. If v is adjacent to all vertices in the current clique, add v to the current clique, and add all neighbors of v not belonging to the current clique to the queue, and mark v processed.

To find a shortest path avoiding H (which is either a maximal clique or an induced non-triangle odd cycle, every neighbor of which is adjacent to exactly one vertex of it) and connecting two vertices of H, use breadth-first search (**BFS**). (See [13]).

This implementation clearly takes $O(m+n)$ time.

Task B. We will describe an implementation of the algorithm in Lemma 3 only since the one in Lemma 4 is similar. Ordering the vertices as $v_1, \ldots, v_n$ can be done by either **DFS** or **BFS**. Preprocess the color lists as follows.

chop C_{v_1} so that it has exactly $d(v_1)+1$ colors
for $i \leftarrow 2$ **to** n **do** chop C_{v_i} so that it has exactly $d(v_i)$ colors

After the preprocessing, we have $\sum_{i=1}^{n} |C_i| = 2m+1$ and $|\bigcup_{i=1}^{n} C_i| \leq 2m+1$. Let k be the highest color to ever appear in any chopped color list. We may assume $\bigcup_{i=1}^{n} C_i = [k]$. For if not, we can renumber the colors by putting all the colors in all the C_i's in a list M, do a radix sort on M, throwing away all duplicate colors, and then get a new number for each color according to its position in the sorted list M. All this work takes $O(m+n)$ time since the highest possible color is $n\Delta$, which is $O(n^2)$. For each vertex i, we will use a list U_i to keep colors made unavailable for i owing to their having been assigned to some of i's neighbors. We will use a boolean array $A[1..k]$ together with list U_i to determine a color available for each vertex i. The coloring pseudocode is as follows.

for $i \leftarrow 1$ **to** n **do** empty list U_i;
for $i \leftarrow n$ **downto** 1 **do** {

for each color c in C_{v_i} **do** $A[c] \leftarrow$ true;
for each color c in U_{v_i} **do** $A[c] \leftarrow$ false;
$c \leftarrow$ first color in C_{v_i}; **while** $A[c] = false$ **do** $c \leftarrow$ next color in C_{v_i};
assign color c to v_i;
for each vertex j on the adjacency list of v_i **do** append color c to U_j }

It is not hard to see that this implementation is correct and takes $O(m + n)$ time.

The foregoing discussion gives our main theorem.

Theorem 2. *There exists an $O(m + n)$-time algorithm to Δ-list-color Brooks graphs.*

3 Subcubic Graphs

Let G be a subcubic graph. Consider a maximal collection $\mathcal{C}$ of edge-disjoint cycles in G. Cycles in $\mathcal{C}$ must be vertex-disjoint since G is subcubic. The graph $\mathcal{T} = G - \mathcal{C}$ is a collection of trees since $\mathcal{C}$ is maximal. This simple fact is the decomposition principle [11] for subcubic graphs: *A subcubic graph G can be decomposed in linear time into edge-disjoint subgraphs $\mathcal{C}$ and $\mathcal{T}$, where $\mathcal{C}$ is a collection of vertex-disjoint cycles, and $\mathcal{T}$ is a forest of maximum degree at most 3. Furthermore, we can choose to decompose G so that all cycles in $\mathcal{C}$ are induced cycles.*

We now describe a particularly simple algorithm to 3-list-color any subcubic, Brooks graph G. First check whether G is cubic or not. If not, then apply the algorithm in the proof of Lemma 3. So assume that G is cubic. Decompose G into trees $\mathcal{T}$ and induced cycles $\mathcal{C}$. If $\mathcal{C}$ contains an even cycle, then apply the algorithm in the first part of the proof of Lemma 4. So assume that every cycle in $\mathcal{C}$ is odd. Suppose there is a cycle $C \in \mathcal{C}$ with a neighbor w that is adjacent to either 2 or 3 vertices of C. Such a neighbor w must be a tree vertex since G is cubic. We claim that w together with the vertices of C induce a whel, so that we can apply the algorithm in the second part of the proof of Lemma 4. This is clear if w is adjacent to 2 vertices of C since C is odd. If w is adjacent to 3 vertices of C, then C cannot be a triangle. For otherwise w together with the vertices of C would induce a K_4 which would necessarily equal G, contradicting G being a Brooks graph. So assume that every neighbor of any cycle $C \in \mathcal{C}$ is adjacent to exactly one vertex of C. Suppose there exist cycles $C_1, C_2 \in \mathcal{C}$ joined by at least 2 edges. Let $C_1 = \langle v_1, \ldots, v_p \rangle$ and $C_2 = \langle w_1, \ldots, w_q \rangle$ and $v_i w_k$ and $v_j w_\ell$ be two edges joining them. We may assume that $i < j$ and no edge joins any of $v_{i+1}, \ldots, v_{j-1}$ to any vertex of C_2. The cycle C_2 consists of two paths joining w_k to w_ℓ. Necessarily one of these paths is even and the other is odd since C_2 is odd. (The odd path may have length 1.) The vertices on one of these paths together with vertices $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$ induce either an even cycle or a theta graph $\theta(1, b, c)$; and we can apply the algorithm in either the first or the third part of the proof of Lemma 4, respectively. So assume that any two cycles in $\mathcal{C}$ are joined by at most 1 edge. Let H be the graph that results from

contracting all cycles in $\mathcal{C}$. Then $\delta(H) \geq 3$ and thus H contains some cycle. Let D be an induced cycle in H. Cycle D necessarily contains some vertex $[X]$ that exists as the result of the contraction of some cycle X. Consider such a vertex. In D, there are two edges e_1, e_2 incident with $[X]$. In G, edge e_1 (e_2) is incident with a vertex x_1 (x_2) on the cycle X. In G, cycle X consists of one odd path and one even path connecting x_1 to x_2. (The odd path may have length 1.) By replacing every vertex of D that was derived from cycle contraction with one of these two paths we can turn the cycle D in H into a cycle C in G. We execute the following replacement procedure. *If vertex* $[X]$ *is not the last one yet to be replaced, then replace* $[X]$ *by the shorter of the two cycle paths in* X *connecting* x_1 *to* x_2*. Otherwise, replace* $[X]$ *by choosing from one of the two cycle paths in* X *the one that will make the resulting cycle* C *even.* The vertices of C then induce either an even cycle or a theta graph $\theta(1, b, c)$, and we can apply the algorithm in either the first or the third part of the proof of Lemma 4, respectively.

Acknowledgments. I am most grateful to Hal Gabow for the Cycle List Coloring Lemma, his careful reading of numerous versions of the manuscript, and helpful discussions. I also thank an anonymous referee whose comments help improve the paper.

References

[1] Erdős, P., Rubin, A.L., Taylor, H.: Choosability in graphs. In: Proceedings of the West-Coast Conference on Combinatorics, Graph Theory and Computing. Volume XXVI of Congressus Numerantium., Arcata, California (1979) 125–157
[2] Jensen, T.R., Toft, B.: Graph Coloring Problems. John Wiley & Sons, Inc., New York, New York (1995)
[3] Brooks, R.L.: On colouring the nodes of a network. Proceedings of the Cambridge Philosophical Society. Mathematical and Physical Sciences **37** (1941) 194–197
[4] Lovász, L.: Three short proofs in graph theory. Journal of Combinatorial Theory Series B **19** (1975) 269–271
[5] Lovász, L.: Combinatorial Problems and Exercises. Second edn. North-Holland Publishing Co., Amsterdam (1993)
[6] Hopcroft, J.E., Tarjan, R.E.: Dividing a graph into triconnected components. SIAM Journal on Computing **2** (1973) 135–158
[7] Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM Journal on Computing **1** (1972) 146–160
[8] Gabow, H.N.: Path-based depth-first search for strong and bi-connected components. Information Processing Letters **74** (2000) 107–114
[9] Greenlaw, R., Petreschi, R.: Cubic graphs. ACM Computing Surveys **27** (1995) 471–495
[10] Skulrattanakulchai, S.: 4-edge-coloring graphs of maximum degree 3 in linear time. Information Processing Letters **81** (2002) 191–195
[11] Gabow, H.N., Skulrattanakulchai, S.: Coloring algorithms on subcubic graphs. (submitted)
[12] Bollobás, B.: Modern Graph Theory. Springer-Verlag, New York (1998)
[13] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. Second edn. McGraw-Hill, New York (2001)

Robot Localization without Depth Perception

Erik D. Demaine[1], Alejandro López-Ortiz[2], and J. Ian Munro[2]

[1] MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA, edemaine@mit.edu
[2] Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {alopez-o,imunro}@uwaterloo.ca

Abstract. Consider the problem of placing reflectors in a 2-D environment in such a way that a robot equipped with a basic laser can always determine its current location. The robot is allowed to swivel at its current location, using the laser to detect at what angles some reflectors are visible, but no distance information is obtained. A polygonal map of the environment and reflectors is available to the robot. We show that there is always a placement of reflectors that allows the robot to localize itself from any point in the environment, and that such a reflector placement can be computed in polynomial time on a real RAM. This result improves over previous techniques which have up to a quadratic number of *ambiguous points* at which the robot cannot determine its location [1, 9]. Further, we show that the problem of optimal reflector placement is equivalent to an art-gallery problem within a constant factor.

1 Introduction

Problem: Robot localization. For a mobile robot to plan its motion, it requires both knowledge of its surrounding environment and accurate information of its current location in this environment. However, the robot's motion is imprecise from such effects as friction, unevenness of the terrain, and inertia, so the robot's location becomes uncertain. Consequently, robots often perform corrective measurements that allow them to rehome their current position (e.g. [5,3,8, 4]). Thus the problem of *robot localization* arises: determine the current location of the robot in its surrounding environment. The basic approach to localization is for the robot to sense its immediate surroundings, and then match this local image against an internal model or *map* of the entire environment. Common sensing devices include vision, radar, sonar, and ladar (laser radar).

Highly detailed information about the environment can be obtained only at the expense of a complex vision system, as well as collection and processing time for the data gathered. An efficient low-cost method of localization would thus allow more accurate motion control for the robot. This paper investigates robot localization with particularly cheap and limited vision systems.

Model: Reflecting Landmarks. Typically robots use landmarks to identify their position [8,3]. These landmarks can either be naturally present (such as a wall or door) or be artificially introduced (magnetic markers, reflectors, beacons). In this paper we follow the model of Sugihara [9], using mutually indistinguishable reflective markers (*reflectors*) that provide angular measurements. This

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 249–259, 2002.

can be realized in a simple and inexpensive form by placing reflective strips or mirrored cylinders in selected positions in the environment. The robot shines a laser in a 360-degree scan and records the angular magnitude of those directions at which a reflection was detected. The result is a star of rays that the robot must match against its given data. What makes this problem difficult is that the robot does not know the distance at which the reflection occurred, nor which reflector caused the reflection, and nor does the robot know of a preferred direction or "true north."

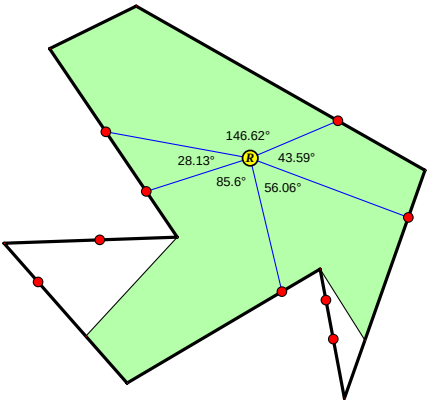


Fig. 1. The robot R knows only the cyclic sequence of angles between visible reflectors (drawn as circles), here placed along the boundary of the polygon. The visibility region is shaded.

Connection to Art Galleries. Sugihara [9] observed that the reflector-placement problem is a generalization of an art-gallery problem. In the classic art-gallery problem, the goal is to choose fixed locations for *guards* (points) such that every point in the environment is visible from at least one guard; equivalently, at least one guard is visible from every point. If we think of reflective strips as guards, certainly the robot needs to see at least one reflective strip at all times. Thus any solution to the reflector-placement problem is also a solution to the art-gallery problem. We establish a connection in the reverse direction.

Previous Work: Ambiguities. Sugihara [9] showed that it is possible to mark the environment in such a way that the robot can localize itself from all but a finite number of *ambiguous points* [9]. Pairs of ambiguous points have the property that the angle readings are the same from either point in the pair, and hence if the robot is placed at either point, it cannot determine at which of the two points it is located. Avis and Imai [1] proved that the total number of degenerate positions for n reflectors, k of which are visible from the robot, is in the worst case $\Theta(n^2/k)$ [1]. Hence, by placing $k = O(n^2)$ reflectors the number of ambiguous points can be reduced to at most a constant number.

More recently, González-Banos and Latombe [5] considered the related problem of finding a minimum set of identifiably distinct reflectors in a given polygon subject to incidence and range constraints. They propose a randomized algorithm which returns, with high probability, a set of guards which is a small non-constant factor away from the minimum number of guards required. The incidence and range constraints model real-life limitations of reflector resolution and sensing devices. However, in either application, ambiguities are never fully resolved, so the robot cannot be guaranteed to be able to localize itself.

Our Results. In this paper, we show that any polygon can be unambiguously marked using at most ten reflective strips per guard using a particular instance of the well-studied family of art-gallery problems. Next, we show that at least four reflectors per guard are needed in the worst case. Lastly, we study changes in the complexity of the localization task when we consider more powerful localization primitives, such as a compass (true north) or a 3-D environment with or without a preferred "up" position.

2 Marking a Single Wall

A simple but important subproblem is when there are no obstacles, and the robot R can be placed anywhere in the plane except on top of one of the reflectors. In this context, we show that two reflectors limit the robot's position to a one-dimensional curve, and three reflectors limit the robot's position to a finite number of points. For two reflectors A, B, the robot's laser scan measures two angles, $\angle ARB$ and $2\pi - \angle ARB$. From elementary geometry it follows that the locus of points forming a fixed angle with two points is an arc of a circle passing through those points together with the reflection of that arc through the line joining the two points. See Figure 2.

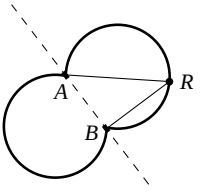


Fig. 2. Two reflectors A, B on a line, and the crescent of points R with fixed angle $\angle ARB$.

Lemma 1. [6] *Given two distinct points A and B on a circle, the interior angle $\angle ARB$ is the same for all R on either of the open arcs connecting A and B* [Euclid's Proposition III.21]. *Furthermore, if C is the center of the circle, then $\angle ARB = \frac{1}{2}\angle ACB$ for R on the longer arc and $\angle ARB = \pi - \frac{1}{2}\angle ACB$ for R on the shorter arc* [Euclid's Proposition III.33].

There are two circles with center C and C' such that $\angle AC'B = \angle ACB = \theta$ for any $0 < \theta \leq \pi$, and these circles are reflections of each other through AB. Thus, the longer arcs of these circles correspond to angles θ satisfying $0 < \theta \leq \pi/2$, and the shorter arcs correspond to angles θ satisfying $\pi/2 \leq \theta < \pi$. Hence, there are precisely two arcs corresponding to each angle θ. Together these arcs are called the *θ-crescent* of A and B.

Lemma 2. *Given an angle $0 < \theta < \pi$, the θ-crescent of points A and B is precisely the locus of points R satisfying $\angle ARB = \theta$.*

For this lemma to hold for $\theta = 0$ and π as well, there are two additional special cases, corresponding to the points along the line AB which we have so far ignored. The points R strictly between A and B satisfy $\angle ARB = \pi$, and the other points R (except A and B) satisfy $\angle ARB = 0$. Thus, we define the π-crescent of A and B to be the open line segment between A and B, and the 0-crescent to be the line AB minus the closed line segment between A and B.

In particular, two reflectors certainly do not suffice to uniquely determine the position of the robot: they leave every point in the plane ambiguous by an uncountably infinite amount.

We now turn to the case of three reflectors A, B, C in an arbitrary position in the plane. At first it might seem that three angles suffice to uniquely determine a position. Indeed, this would be the case if the robot knew the correspondence between reflectors and reflection angles. Because this information is not known, however, we can cyclicly shift this correspondence, compute the corresponding crescents, and take their intersection, as shown in Figure 3.

More precisely, in this figure, we proceed as follows from the triangle ABC and the point R chosen arbitrarily. We draw the circular arc with aperture $\angle ARB$ subtending AC. We repeat this procedure with $\angle BRC$ subtending AB. These two circular arcs intersect at R'. It follows then by construction that $\angle ARC = \angle BR'C$, $\angle ARB = \angle AR'C$, and $\angle BRC = \angle AR'B$, and hence R and R' cannot be distinguished. Therefore, three reflectors in general position do not uniquely determine the position of the robot. However, three reflectors do restrict the position to one of a finite number of locations, at most for each cyclic shift of the angles. The difficulty is to make this number of possible positions equal to exactly one.

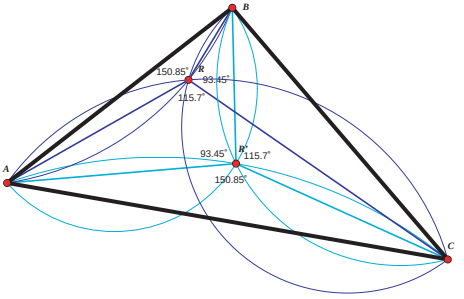


Fig. 3. Positions R and R' cannot be distinguished because the labels of A, B, C are unknown and hence can be shifted. Thin lines show the half-crescents, medium lines show the lines of sight, and thick lines show the triangle.

3 Localization in Polygons

We now focus on the case in which robot motion is limited to a polygon, possibly with polygonal holes. Reflectors are placed on the walls, this means that the robot cannot collide with the reflector and that it must be on the same side of the wall as the reflector. Three reflectors along a common wall will always be seen in the same order so the results of the previous section apply and three reflectors suffice for a convex polygon.

For nonconvex polygons, we convert a particular type of art-gallery guard placement into a collection of reflectors by expanding each guard into a tight cluster of reflectors. For this to work, we consider a slight variation on the art-gallery problem. A *wall guard* is a positive-length (short) interval along an edge of the polygon. A *wall-guard placement* must satisfy that every point in the polygon can see an entire wall guard (strong visibility), and the goal of the art-gallery problem is to minimize the number of wall guards. It is easy to show that the worst-case bounds for wall guards are similar to standard vertex guards: $\Theta(n)$ guards suffice and are sometimes necessary to guard an n-vertex polygon. The main difficulty with obstacles is that (most likely) not all of the reflectors are visible from a given point. Thus a major challenge is to identify *which* reflectors

are those seen. Such an encoding scheme will be the focus of this section. Once we have such a scheme, and we can identify three visible reflectors on a common wall, then we can appeal to the previous section to determine the robot's position relative to the reflectors. As a result, the entire reflector-placement problem will reduce to the art-gallery problem outlined above.

So we turn to the problem of encoding information in the reflectors so that the robot can tell which reflectors it can see. We make use of the *cross ratio* as suggested by Sugihara [9]. This fundamental concept in projective geometry allows us to store a number—represented by four collinear points (reflectors)—that is readable from any point not collinear with the four points.

3.1 The Cross Ratio of a Pencil

Let A, B, C, and D be four collinear points; a, b, c and d denote four concurrent lines and let O be the point of concurrency. We denote by ac the angle between lines a and c.

Definition 1. *The* cross ratio $\{ABCD\}$ of four collinear points A, B, C, D *is defined as the quotient* $\frac{AC}{CB} \Big/ \frac{AD}{DB}$ *where the magnitude of a segment is directed, i.e.* $AC = -CA$. *The* cross ratio of a pencil of lines *is defined as* $\frac{\sin ac}{\sin cb} \Big/ \frac{\sin ad}{\sin db}$. *The cross ratio of four lines is denoted by* $\{abcd\}$. *Given four points A, B, C, D and a point O, we denote by* $O\{ABCD\}$ *the cross ratio of the pencil defined by the lines* $\overline{OA}$, $\overline{OB}$, $\overline{OC}$ *and* $\overline{OD}$.

Theorem 1 ([7]). *Let abcd be a pencil of lines passing through vertex O, and let L be a line transversal of the pencil not passing through O. Let A be the point of intersection of L and a, and analogously define points B, C, and D. Then* $\{abcd\} = \{ABCD\}$. *Conversely, let A, B, C, and D be four collinear points, and O be a point off the line ABCD. Then* $\{ABCD\} = O\{ABCD\}$.

In other words, the cross ratio of four collinear points is the same when viewed from any vantage point off the line.

We can use this principle to label walls in such a way that the label can be read from any robot position O. More precisely, let $G_1, \ldots, G_k$ denote the set of wall guards. Then we place four (collinear) reflectors along wall guard G_i so that those four points have integral cross ratio i. Thus, from any robot position O, the wall guarding guarantees that we see at least one integral cross ratio, and this cross ratio identifies the wall, in principle permitting localization.

Unfortunately, this approach does not suffice, because we do not know which of the visible reflectors form collinear quadruples from a common wall guard. For example, consider a situation in which the robot sees five reflectors, A, B, C, D, X; the first four reflectors A, B, C, D correspond to a single wall guard; and the last reflector X corresponds to another guard whose quadruple of reflectors is partially occluded. Moreover, the robot happens to be positioned in such a way that both $O\{ABCD\}$ and $O\{BCDX\}$ are integers in $\{1, \ldots, k\}$.

In this case, the robot cannot in general distinguish which of the two sequences corresponds to a guard and which is spurious.

Indeed, this scenario is but one of several possible ambiguous configurations. To solve these problems, we use additional reflectors and more careful placements of guard reflectors to ensure that these ambiguities are fully resolved.

3.2 The Cross Ratio of Noncollinear Points

The following theorems from projective geometry [10] will help in the task of disambiguating a given set of angular measurements. First we need to characterize those points from which a given noncollinear quadruple forms an integer cross ratio in $\{1, \ldots, k\}$:

Theorem 2 (Steiner's Theorem [2,10]). *Given four points A, B, C, D, not all collinear, and given a cross ratio r, the locus C_r of points O such that the cross ratio $O\{ABCD\}$ equals r—$C_r(ABCD) = \{O \mid O\{ABCD\} = r\}$—is a conic curve. Conversely five points in the plane, not necessarily in general position, define a unique conic passing through them.*

The conic may be an ellipse, circle, parabola, hyperbola, or the degenerate cases of a point, a line, or two lines. Consider now four reflectors, three of which are collinear. The following two lemmas describe the robot locations O from which the four reflectors would appear to belong to a common wall guard.

Lemma 3. *Given three collinear points and cross ratio r, there is a fourth point on the line (including the projective point at infinity) realizing cross ratio r.*

Proof. We consider the four points to lie along the (projective) real line with C at the origin. Without loss of generality, let $\mathrm{dist}(AC) = 1$. Define $b = \mathrm{dist}(BC)$ and similarly $d = \mathrm{dist}(BD)$ which implies $\mathrm{dist}(AD) = 1 + b + d$. Hence $\frac{AC}{CB} \Big/ \frac{AD}{DB} = \frac{1}{-b} \Big/ \frac{1+b+d}{-d} = \frac{d}{b\,(1+b+d)} = c$. Solving for d we have $d = b\,c\,(1+b)/(1-b\,c)$. □

Given an edge e of the polygon and three points $X, Y, Z \in e$, let I_{XYZ} be the set of points on e realizing an integer ratio, i.e. $I_{XYZ} = \{\, W \in e \mid \{XYZW\} \in \mathbb{Z} \,\}$. If, from a point Q, four reflectors not all collinear form a pencil with an integer cross ratio, then this quadruple is called *a spurious quadruple from Q*. If the location of Q is clear from the context, then we simply refer to the quadruple as *spurious*. In general, given four points A, B, C, D, not all collinear, we denote by $C_{\mathbb{Z}}(ABCD)$ the locus of points that make the quadruple A, B, C, D spurious. More formally, $C_{\mathbb{Z}}(ABCD) = \bigcup_{k \in \mathbb{Z}} C_k(ABCD)$.

Lemma 4. *Given three collinear points A, B, C, a point E not on the line, and a cross ratio r, the locus of points O such that $O\{ABCE\} = r$ is a line.*

Proof. Given the three collinear points A, B, C, let D be the point along that line such that $\{ABCD\} = r$ as per Lemma 3. Then given a point O along the line $\overline{DE}$ we have that $O\{ABCE\} = O\{ABCD\} = r$. □

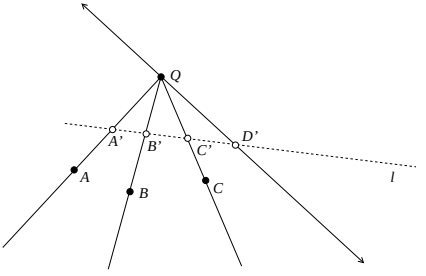


Fig. 4. Point D' such that $Q\{ABCD'\} = k$.

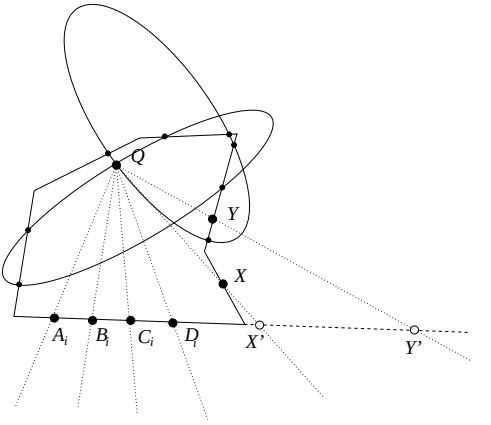


Fig. 5. The view from point Q is ambiguous.

From Steiner's Theorem it follows that $C_{\mathbb{Z}}(ABCD)$ is composed of conics through the points A, B, C, D.

Claim 1. *Given an integer k, a point Q, and three reflectors, the set of fourth reflector points that forms a spurious quadruple from Q with cross ratio k is precisely a line.*

Proof. Using Figure 4 as reference, we are given three reflectors A, B, C and a point Q. First construct the lines $\overline{QA}$, $\overline{QB}$, and $\overline{QC}$. Then use an auxiliary line l not passing through Q which creates the sequence $A'B'C'$ on l. There is a unique point D' on l creating a sequence of cross ratio k as per Lemma 3. Precisely the reflectors D on $\overline{QD'}$ form spurious quadruples with $Q\{ABCD\} = k$. □

Definition 2. *Given a reflector X, we denote by $C_{\mathbb{Z}}(X)$ the union of all conics $C_{\mathbb{Z}}(XYWZ)$ over all reflectors Y, W, Z forming a spurious quadruple. These conics are called conics of ambiguity.*

Definition 3. *A point Q is ambiguous of degree k if it belongs to the intersection of k conics of ambiguity for some reflectors $X_1, \ldots, X_k$, i.e. $Q \in \bigcap_{j=1}^{k} C_{\mathbb{Z}}(X_j)$.*

3.3 A Reflector Placement Algorithm

Because four points alone do not in general uniquely determine the position of the robot, we need to increase the number of reflectors per guard. As we have shown, the existence of ambiguous points of degree 1 is unavoidable because there is a set of conics from which a quadruple becomes spurious. This fact extends to points of degree 2, as in general two conics of ambiguity corresponding to two different quadruples might intersect.

To distinguish from these ambiguous points, we place three quadruples along each guard G_i, realizing the cross ratios $3i$, $3i+1$ and $3i+2$, respectively. Denote by $q_j = \{A_j, B_j, C_j, D_j\}$ the quadruple of reflectors realizing cross ratio j. From any point in the region guarded by G_i, the robot can always see the three quadruples $q_{3i}, q_{3i+1}, q_{3i+2}$ with consecutive integer cross ratios. We arrange so that the points of ambiguity have degree at most 2. Thus, from any point in the polygon, the robot sees at most two spurious quadruples, and at least one consecutive sequence of three quadruples with cross ratios $3i$, $3i+1$ and $3i+2$ which is the actual set corresponding to guard G_i. The robot can localize itself by searching for a consecutive sequence of three quadruples that conclude a common location for the robot, which must then be the correct location because at least one of those three quadruples must not be spurious.

We provide an incremental construction, inserting the quadruples for each guard G_i for i from 0 to n. Denote by g_i the set of reflectors corresponding to guard G_i, i.e. $g_i = q_{3i} \cup q_{3i+1} \cup q_{3i+2}$. At iteration i of the algorithm, the robot can localize itself without ambiguity in any region guarded by a guard G_j for $j < i$. Moreover, as we introduce the reflectors of the set g_i, we ensure that no new ambiguities are introduced in previously guarded regions.

The algorithm maintains sets of relevant geometric objects:

1. A set $\mathcal{M}$ of the reflectors placed thus far.
2. A set $\mathcal{D}$ of all points of degree 2, i.e. $\mathcal{D} = \bigcap_{X \in \mathcal{M}} C_{\mathbb{Z}}(X)$.
3. As we insert A_{i+1}, B_{i+1}, C_{i+1}, and D_{i+1}, the sets of points $I_{B_i C_i D_i}$, $I_{C_i D_i A_{i+1}}$, $I_{C_i D_i A_{i+1}}$, and$I_{D_i A_{i+1} B_{i+1}}$ along edge $e_{\lceil i/3 \rceil}$ respectively.
4. Given a point $Q \in \mathcal{D}$, an integer k, and any three reflectors, as per Claim 1, there is a line along which placement of a reflector would increase the degree of Q. Let $\mathcal{H}$ denote the set of these lines, over all $Q \in \mathcal{D}$, $k \in \mathbb{Z}$, and triples in $\mathcal{M}$. The algorithm avoids ambiguities by keeping track of the intersections of the lines with the boundary of the polygon ∂P.
5. After inserting the first two reflectors A_i, B_i of a quadruple q_i on edge $e_{\lceil i/3 \rceil}$, it computes the set of points $\mathcal{I} = \bigcup_{W,Y \in \mathcal{M};\ X \in \mathcal{H} \cap e_{\lceil i/3 \rceil}} I_{WYX}$. This corresponds to undesirable potential locations for reflector C_i that would force D_i to coincide with a point $X \in \mathcal{H} \cap e_{\lceil i/3 \rceil}$, hence increasing the degree of some $Q \in \mathcal{D}$.

The following theorem gives an upper bound on the size of these sets.

Theorem 3 (Bezout's Theorem for Conics [2]). *Any two distinct conics intersect in at most four points.*

It follows then that there is only a countable set of points of degree 2. In fact, if we restrict ourselves to conics of cross ratio at most $3n$ then there are $O(n^4)$ points in H. With these sets in hand we can now describe the algorithm:

For each $i = 1, \ldots, 3n$, insert one-by-one the reflectors A_i, B_i, and C_i of each quadruple while avoiding the sets I_{XYZ} in (3) above and the sets $\mathcal{H} \cap \partial P$ and $\mathcal{I}$. This can always be done as these sets are discrete and the positions of A_i, B_i and C_i are continuously varying. After this process, the location of D_i is fixed.

Because of step (5) above, we know that for any triple XYZ not containing C_i, the conics of ambiguity through $XYZD_i$ do not increase the degree of points in $\mathcal{D}$. Nevertheless, it is indeed possible for a quadruple of the form C_iD_iXY, with $X, Y \in \mathcal{M}$, to increase the degree of a point in $\mathcal{D}$.

Let Q be the point in $\mathcal{D}$ whose degree is three after inserting D_i (refer to Figure 5). Using the notation of this figure, we see that Q lies in the intersection of two conics (ellipses), and hence is of degree two. To remove this ambiguity, we must move C_i and D_i in such a way that the cross ratio $\{A_iB_iC_iD_i\} = i$ remains constant yet the cross ratio $Q\{C_iD_iXY\}$ is no longer an integer. First note that $Q\{C_iD_iXY\} = \{C_iD_iX'Y'\}$. Without loss of generality, we introduce a real axis coordinate system on edge $e_{\lceil i/3 \rceil}$ such that A_i is the origin and $B_iA_i = 1$. Let c, d, x, y denote the position on the real axis of C_i, D_i, X', Y' respectively. Hence $\frac{1}{c-1} \Big/ \frac{d}{d-c-1} = 3 \implies d = \frac{c+1}{1-3c+3}$. Similarly $\frac{d-c}{x} \Big/ \frac{d-c+x+y}{y} = k\,x \implies d = \frac{yc-kxc+kx^2+kxy}{y-kx}$. Equating the two expressions for d, we obtain a quadratic expression on c with at most two solutions for each integer k. That is, the set of positions C_i that force a D_i to make Q ambiguous is a discrete set and hence it can be avoided by perturbing C_i by an ϵ amount.

Theorem 4. *If g is the number of wall guards required to guard a polygon P, then a robot can localize itself using at most $10g$ reflectors.*

Proof. In the previous algorithm, for a guard g_j we can identify $D_{3j} = A_{3j+1}$ and $D_{3j+1} = A_{3j+2}$, thus reducing reflectors per guard from 12 to 10. □

4 Lower Bound

In this section we prove that, in the worst case, at least $4g - 2$ reflectors are required for unambiguous robot localization for a polygon guardable by g (wall) guards. This lower bound holds even if the reflectors are not on the walls of the polygon. The example is the standard *comb polygon* shown in Figure 6.

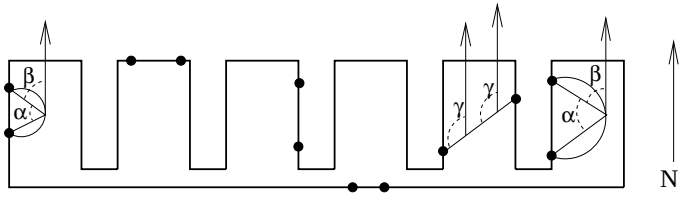


Fig. 6. Two types of ambiguous positions.

Theorem 5. *If g is the number of wall guards required to guard a polygon P, then we need at least $4g - 2$ reflectors to uniquely localize the position of a robot in P.*

Proof. It follows from the discussion in Section 2 that we need at least three reflectors per tine. Suppose that three or more tines had just three reflectors.

Then either two of these tines have the three reflectors collinear, or two of the tines have the three reflectors noncollinear. If the three reflectors are collinear in at least two tines, then for a position arbitrarily close to the midpoint between two reflectors in such a tine, the angles observed are an almost-180° angle and an almost-zero angle. This angular configuration can be realized in both of the tines with three collinear reflectors, and therefore it is ambiguous.

Alternatively, if at least two of the three-reflector sets are not collinear, then drop a perpendicular from the vertex opposite the longest edge of the triangle formed by three reflectors, and obtain a point for the robot that reads angles 90°, 90°, and 180°. This applies to both tines and therefore the position is ambiguous. □

5 Other Localization Primitives

The localization problems may become a simpler task if the robot can benefit from alternative, independent orientation mechanisms. In real life, the robot moves on the floor, which is a 2-D surface embedded in a 3-D space. We can take advantage of this fact by using the third dimension to place reflectors. In this model, the robot can perform 360-degree scans along any given chosen plane through its current position. The robot might also perform a 2-D-like scan by performing a laser sweep along the horizontal plane of height zero. The robot is equipped with a device that indicates the "up" direction (defining the orientation of the floor plane) at all times. We omit proofs in this abstract.

Theorem 6. *If n is the number of vertices in a polygon P, then a robot aided by a compass indicating a North position at a point at infinity requires at least $n/4 - 8$ reflectors in the worst case to uniquely indentify its position in P.*

Theorem 7. *Consider a robot on a plane in a 3-D environment with walls, given an "up" direction and a map of the environment. Let g be the number of guards required to guard such environment. Then at least $4g$ reflectors and at most $6g$ reflectors are needed for the robot to localize itself in the environment.*

6 Conclusions

We have developed a method to remove ambiguities from Sugihara's reflector model for robot localization. This model can be implemented economically both in terms of hardware (laser, reflectors) as well as computational requirements. We have given nearly matching upper and lower bounds on the number of reflectors needed per guard. We also considered alternative scenarios for localization with more general primitives and showed upper and lower bounds in these contexts.

Acknowledgments. We thank Martin Demaine for many helpful discussions.

References

1. D. Avis and H. Imai. Locating a robot with angle measurements. *Journal of Symbolic Computation*, 10(3–4):311–326, 1990.
2. M. Berger. *Geometry*, vol. II. Springer-Verlag, 1987.
3. M. Betke and L. Gurvits. Mobile robot localization using landmarks. In *Proc. IEEE International Conference on Robotics and Automation*, vol. 2, 135–142, 1994.
4. G. Dudek, K. Romanik, and S. Whitesides. Localizing a robot with minimum travel. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, 1995.
5. H. Gonzalez-Banos and J. Latombe. A randomized art-gallery algorithm for sensor placement. In *Proc. 17th ACM Symposium on Computational Geometry*, 2001.
6. S. T. L. Heath. *The thirteen books of Euclid's Elements translated from the text of Heiberg with introduction and commentary.* Dover Publications, New York, 1956.
7. L. S. Shively. *Introduction to Modern Geometry.* John Wiley & Sons, 1939.
8. R. Sim and G. Dudek. Mobile robot localization from learned landmarks. In *Proc. IEEE/RSJ Conference on Intelligent Robots and Systems*, 1998.
9. K. Sugihara. Some location problems for robot navigation using a single camera. *Computer Vision, Graphics, and Image Processing*, 42(1):112–129, 1988.
10. O. Veblen and J. W. Young. *Projective Geometry*, vol. 1. Gin and Company, 1938.

Online Parallel Heuristics and Robot Searching under the Competitive Framework*

Alejandro López-Ortiz[1] and Sven Schuierer[2]

[1] Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1. alopez-o@uwaterloo.ca
[2] Institut für Informatik, Am Flughafen 17, Geb. 051, D-79110, Freiburg, Germany. schuiere@informatik.uni-freiburg.de

Abstract. In this paper we investigate parallel searches on m concurrent rays for a point target t located at some unknown distance along one of the rays. A group of p agents or robots moving at unit speed searches for t. The search succeeds when an agent reaches the point t. Given a strategy S the competitive ratio is the ratio of the time needed by the agents to find t using S and the time needed if the location of t had been known in advance. We provide a strategy with competitive ratio of $1 + 2(m/p - 1)(m/(m - p))^{m/p}$ and prove that this is optimal. This problem has applications in multiple heuristic searches in AI as well as robot motion planning. The case $p = 1$ is known in the literature as the cow path problem.

1 Introduction

Searching for a target is an important and well studied problem in robotics. In many realistic situations such as navigation in an unknown terrain or a search and rescue operation the robot does not possess complete knowledge about its environment. In the earlier case the robot may not have a map of its surroundings and in the latter the location of the target may be unknown [3,11,12,17,18].

The *competitive ratio* [20,11] of a strategy S is defined as the maximum of the ratio of the search cost using S and the optimal distance from the starting point to the target, over all possible positions of the target.

Consider an exhaustive search on m concurrent rays. Here, a point robot or—as in our case—a group of point robots is assumed to stand at the origin of m concurrent rays. One of the rays contains the target t whose distance to the origin is unknown. The robot can only detect t if it stands on top of it. It can be shown that an optimal strategy for one robot is to visit the rays in cyclic order, increasing the step length each time by a factor of $m/(m - 1)$ if it starts with a step length of 1. The competitive ratio C_m achieved by this strategy is given by $C_m = 1 + 2m^m/(m - 1)^{m-1}$ which can be shown to be optimal [1,6, 10,15]. The lower bound in this case has proven to be a useful tool for proving

* This research is partially supported by the DFG-Project "Diskrete Probleme", No. Ot 64/8-2.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 260–269, 2002.

lower bounds for searching in a number of classes of simple polygons, such as star-shaped polygons [13], $\mathcal{G}$-streets [5,14], HV-streets [4], and θ-streets [4,8].

Parallel searching on m concurrent rays has been addressed before in two contexts. In the first context a group of p point robots searches for the target. Neither the ray containing the target nor the distance to the target are known. The robots can communicate only when they meet. The search concludes when the target is found and all robots are notified thus. Baeza-Yates and Schott investigated searching on the real line [2] and Hammar, Nilsson and Schuierer considered the same case for m concurrent rays [7].

The second context is the on-line construction of hybrid algorithms. In this setting we are given a problem Q and m heuristics or approaches to solving it. The implementation of each approach is called a *basic* algorithm. We are given a computer with $k < m$ disjoint memory areas which can be used to run one basic algorithm and to store the results of its computation. Only a single basic algorithm can be run on the computer at a given time. It is not known in advance which of the algorithms solves the problem Q—although we assume that there is at least one—or how much time it takes to compute a solution. In the worst case only one algorithm solves Q whereas the others do not even halt on Q. One way to solve Q is to construct a hybrid algorithm in the following way. A basic algorithm is run for some time, and then a computer switches to another basic algorithm for some time and so on until Q is solved. If $k < m$, then there is not enough memory to save all of the intermediate results. Hence, the current intermediate results have to be discarded and later recomputed from scratch. An alternative way to look at this problem is to assume that we are given k robots searching on m rays for a target. Each ray corresponds to a basic algorithm and a robot corresponds to a memory area, with only one robot moving at any given time. Discarding intermediate results for an algorithm A is equivalent to moving the robot on the ray corresponding to A back to the origin. Kao et al. [9,21] gave a hybrid algorithm that achieves an optimal competitive ratio of $k + 2(m-k+1)^{m-k+1}/(m-k)^{m-k}$.

A generalization of this context is to consider a distributed setting in which more than one computer or robot perform a simultaneous search. In this case at least one of the robots must reach the target, at which time the search is considered complete. The robots move at unit speed and the competitive ratio is defined as the ratio between the search time and the shortest distance to the target. Under this framework López-Ortiz and Sweet show that the integer lattice on the plane can be searched efficiently in parallel [16].

In this paper we study searches in m concurrent rays which also correspond to an m heuristic problem with $k = p$ memory states and p processors or computers. The "terminate-on-success" framework models search and rescue operations as well as the multiple heuristic searches. We provide an optimal strategy with competitive ratio of $1 + 2(m/p - 1)(m/(m-p))^{m/p}$ for searching m rays with $p < m$ robots in parallel. The case $p = 1$ is sometimes referred in the literature as the cow path problem [10]. A variation of the newly proposed strategy can be applied to other graphs such as trees, resulting in an iterative deepening

scheme which is optimal to within a constant factor. This shows that neither BFS nor DFS are optimal —absent any other information. This has relevance in distributed computing searches in game spaces and automated theorem proving.

The paper is organized as follows. In the next section we present some definitions and preliminary results. In Section 3 we present a lower bound for the problem of searching on m rays with p robots. In Section 4 we then present an algorithm that achieves this bound.

2 Preliminaries

In the following we consider the problem of a group of p robots searching for a target of unknown location on m rays in parallel. The competitive ratio is defined as the quotient of the search time over the shortest distance to the target. In this case we consider robots that have the same maximal speed, which is assumed to be, without loss of generality, one unit of distance per unit of time.

Given a strategy S, at a time T the *snapshot* of S is given by $m + 2p$ values $(s_1, \ldots, s_m, d_1, I_1, d_2, I_2, \ldots, d_p, I_p)$ where s_i is the distance up to which ray i is explored, d_i is the distance of robot i to the origin, and I_i is the index of the ray that robot i is located on. Consider now a strategy X to search on m rays with p robots, in which the robots repeatedly travel one ray for a certain distance and then return to the origin to choose another ray. Let $X_S = (x_0, x_1, \ldots)$ be the collection of distances at which the robots change direction to return to the origin, ordered by the time at which the robots turn around.

Let r_i be the ray on which the robot that turns at x_i is located and T_i be the first time that a robot passes x_i on ray r_i again. Assume that this robot turns around again after having traveled a distance of x_k, where $k > i$. If the target is placed on r_i between x_i and x_k, say at distance d after x_i, then the competitive ratio of the strategy for this placement is

$$\frac{T_i + d}{x_i + d}.$$

Since the competitive ratio is a worst case measure, we see that the competitive ratio C_S of S is at least

$$C_S \geq \sup_{d>0} \left\{ \frac{T_i + d}{x_i + d} \right\} = \frac{T_i}{x_i}. \tag{1}$$

As the target is necessarily found at some point along a step, we obtain

$$C_S = \sup_{i \geq 0} \left\{ \frac{T_i}{x_i} \right\}.$$

We say a ray r is *occupied* at time T if there is a robot on r at this time. We say a ray r is *busy* at time T if there is a robot on r that is moving away from the origin at this time. Let the *schedule* of robot R be the sequence of rays in the order in which they are explored by R together with the distance

to which they are explored, i.e. $Sch_R = (d_1, I_1, d_2, I_2, \ldots)$. Given two strategies we say that S_1 is *contained* in S_2 up to time T, denoted $S_1 \subseteq_T S_2$, if the snapshots of both strategies coincide for all $t \leq T$. Given a sequence of strategies $\mathcal{V} = (S_1, S_2, \ldots)$, we say that the sequence $\mathcal{V}$ converges to a limit strategy S if there is a strictly increasing function $T(n)$ with $\lim_{n\to\infty} T(n) = \infty$ such that for each n, $S_m \subseteq_{T(n)} S_{m+1}$ for all $m \geq n$. The limit strategy S is defined in the obvious way.

3 A Lower Bound

We are interested in proving a lower bound on C_S for any strategy S.

Lemma 1. *Let S be a strategy to search on m rays with p robots. Then there exist a strategy S' with the same competitive ratio or better such that*

1. *At any time t, there is at most one robot on a given ray.*
2. *If a robot moves towards the origin on some ray, then it continues until it has reached the origin.*
3. *All robots are moving at all times.*

Proof. Assume that there are at least two robots on a given ray. Either the paths of these two robots cross in opposing directions along the ray or not. In the latter case, this means that one robot trails the other along that ray and, hence, has no net effect in the exploration. Clearly a modified strategy S' in which the trailing robot stays put in the origin has the same competitive ratio as S. Alternatively, if the robot paths cross in opposing directions consider a strategy S'' which replaces the cross-paths with a "bounce", in which both robots change direction at the point of intersection of their paths. The robots also exchange schedules from that point onwards. S'' is now a strategy in which the robots never properly cross in opposing directions, and hence itself can be replaced with a strategy S' in which one of the robots stays in the origin. S' is a strategy with the same competitive ratio as S in which robots do not change direction away from the origin.

Similarly, if the robot is moving towards the origin and then changes direction, we can create a strategy S' in which the robot stays put rather than moving toward the origin and then backtracking its steps. The strategy S' has the same competitive ratio as S, but no changes in direction away from the origin along a ray.

Lastly if we consider a robot whose sequence of moves includes a stand-still period, clearly removing those idle periods can only decrease the competitive ratio. Let R be a robot that is idle at step i. Then R moves ahead to explore ray I_i in its schedule Sch_R. However this ray might presently be occupied in which case R exchanges schedule with the robot R' occupying the ray and moves ahead to the next ray in $Sch_{R'}$. In turn, this ray might also be occupied, and the robot exchanges schedules yet again, and so on. Note that a swap on a given ray monotonically increases the distance to be traversed on that ray by

it's occupant. Hence this defines a sequence of strategies whose limit strategy S' is well defined. Moreover, S' satisfies all three properties required in the lemma and has competitive ratio no larger than the original strategy S. □

Lemma 2. *There is an optimal strategy to search on m rays with p robots that satisfies Lemma 1 such that if a robot is located at the origin at time T, then it chooses to explore the ray that has been explored the least among all non-busy rays.*

Proof. Let S be an optimal strategy to search on m rays with p robots that satisfies Lemma 1. Assume that robot R is located at the origin at time T and chooses to explore ray r which is explored up to distance d_r. Assume that there is a non-busy ray r' that is explored up to distance $d_{r'} < d_r$. Now consider the strategy S' where the robot chooses to explore the ray r' and the robot that explores ray r' after T in S explores ray r in S'. Each of these rays is explored in S' to its originally scheduled distance in S, only the order changes. Everything else remains the same.

The only difference in competitive ratio between the strategies S and S' is the time when the point located at a distance d_r on ray r is passed the first time by a robot and the time when $d_{r'}$ is passed the first time by a robot on ray r'.

Assume that in Strategy S a robot passes $d_{r'}$ on ray r' at time $T' + d_{r'}$. Since r is explored before ray r', we have $T' > T$. Hence, the competitive ratio of S for those two steps is

$$\max\left\{\frac{T+d_r}{d_r}, \frac{T'+d_{r'}}{d_{r'}}\right\} = 1 + \max\left\{\frac{T}{d_r}, \frac{T'}{d_{r'}}\right\}$$

whereas the competitive ratio of S' for those two steps is

$$\max\left\{\frac{T+d_{r'}}{d_{r'}}, \frac{T'+d_r}{d_r}\right\} = 1 + \max\left\{\frac{T}{d_{r'}}, \frac{T'}{d_r}\right\}.$$

Since $T' > T$ and $d_r > d_{r'}$, $T'/d_{r'} > \max\{T/d_{r'}, T'/d_r\}$ and the competitive ratio of S' is no greater than the competitive ratio of S.

This shows that switching the searching order to favour the least explored ray has no negative effect on the competitive ratio. However if the non-busy ray r' was occupied, then S' violates condition (1) of Lemma 1. In this case, rather than R exploring the occupied ray r' it exchanges schedule from that point onwards with the occupant of r' as in the proof of Lemma 1. First we observe that after the exchange of schedules, r' is no longer the least explored non-busy ray as it either has been explored to a distance $d > d_r > d'_r$ which is further than ray r or it is in the process of being explored to that distance and hence is busy. In this case, we have a new strategy S' in which robot R is about to explore a ray r' which might or might not be non-busy and occupied. We apply the same procedure to what would be the least explored ray r'' in the new strategy S' and we obtain a new strategy S'' in which ray r'' is about to be explored. Note that the distance to which r' is explored increased. Hence this creates a

sequence of strategies $(S, S', S'', \ldots)$ whose limit strategy has competitive ratio no larger than S. Moreover this new strategy satisfies the properties of Lemma 1 and robots explore the least explored non-busy ray in sequence. □

Corollary 1. *There is an optimal strategy to search on m rays with p robots such that at any time the explored distances of all occupied, but not busy rays are larger than the minimum of the explored distances of all unoccupied rays.*

Proof. By Lemma 2 there is an optimal strategy such that a robot at the origin always chooses to explore the non-busy ray that is explored the least. If this ray is occupied, then there is a time at which two robots are on the same ray—a contradiction to Lemma 1. □

A strategy satisfying Lemmas 1 and 2 is termed a *normalized strategy*. The next lemma provides a lower bound for normalized optimal strategies.

Lemma 3. *The competitive ratio C_S of an optimal* normalized *strategy S with turn point sequence $X = (x_0, x_1, \ldots)$ is at least*

$$C_S \geq \sup_{k \geq 0} \left\{ 1 + 2 \sum_{i=0}^{k+m-p} x_i^s \Big/ \sum_{i=k-p+1}^{k} x_i^s \right\} \tag{2}$$

where $X^s = (x_0^s, x_1^s, \ldots)$ is the sequence of the sorted values of X.

Proof. Let S be an optimal normalized strategy. Consider a time T such that robot R_0 is located at the origin. By Lemma 2 we can assume that it chooses to explore the ray that has been explored the least among all unoccupied rays, say this is ray r_0. In general, let r_j be the current ray of robot R_j at time T, for $0 \leq j \leq p-1$.

Now consider the sequence of turn points taken by a robot R_j up to—but not including—time T. These turn points are elements in the sequence X^s; let I_j be the set of indices in X^s of these turn points of robot R_j.

Let the distance up to which ray r_0 is explored at time T be d_0. Note that $d_0 = x_{k_0}^s$, for some $k_0 \geq 0$. Furthermore, let d_j be the distance up to which ray r_j was explored before the robot R_j entered ray r_j. Note that $d_j = x_{k_j}^s$, for some $0 \leq k_j$ where $k_j < k_0$ by Lemma 2. Hence $d_j \leq d_0$. When the robot R_j passes d_j at time $T_j \leq T + d_j \leq T + d_0$ and the target is placed right after d_j on ray r_j, then the competitive ratio for this placement of the target is given by

$$\frac{2\sum_{i \in I_j} x_i^s + x_{k_j}^s}{x_{k_j}^s} = 1 + 2\frac{\sum_{i \in I_j} x_i^s}{x_{k_j}^s},$$

according to Equation 1, for $0 \leq j \leq p-1$. The factor 2 comes from the fact that the robot has traveled to and fro from the origin to each turn point. Hence, the competitive ratio at time $T + d_0$ is at least

$$C_S \geq \max_{0 \leq j \leq p-1} \left\{ 1 + 2\frac{\sum_{i \in I_j} x_i^s}{x_{k_j}^s} \right\} \geq 1 + 2\frac{\sum_{j=0}^{p-1} \sum_{i \in I_j} x_i^s}{\sum_{j=0}^{p-1} x_{k_j}^s}.$$

Here, we make use of the fact that $\max\{a/c, b/d\} \geq (a+b)/(c+d)$, for all $a, b, c, d > 0$. Note that the sum $A = \sum_{j=0}^{p-1} \sum_{i \in I_j} x_i^s$ contains as summands all x_i^s that have been explored up to time T. In particular, A includes all x_i^s that are smaller than $x_{k_0}^s$, as otherwise the robot R_0 would have explored a ray different from r_0 by Lemma 2. Similarly, there are at least $m - p + 1$ unoccupied rays at time T, one of which is r_0. These rays have each been explored to a distance $x_{l_i} \geq x_{k_0}$, for $1 \leq i \leq m - p$ since otherwise robot R_0 would have chosen one of these for exploration at time T. The smallest choice for these $m - p$ values is $x_{k_0+1}^s, \ldots, x_{k_0+m-p}^s$. Hence,

$$\sum_{j=0}^{p-1} \sum_{i \in I_j} x_i^s \geq \sum_{i=0}^{k_0+m-p} x_i^s.$$

Now consider the values d_j, for $1 \leq j \leq p-1$. The value d_j is the distance up to which ray r_j was explored before robot R_j entered it. Since robot R_j chose ray r_j and not ray r_0, Lemma 2 implies that $d_j \leq d_0 = x_{k_0}^s$. The $p-1$ largest such values are $x_{k-p+1}^s, \ldots x_{k-1}^s$ and

$$\sum_{j=1}^{p-1} d_j \leq \sum_{i=k_0-p+1}^{k_0} x_i^s.$$

Hence,

$$C_S \geq 1 + 2\frac{\sum_{j=0}^{p-1} \sum_{i \in I_j} x_i^s}{\sum_{j=0}^{p-1} x_{k_j}^s} \geq 1 + 2\frac{\sum_{i=0}^{k_0+m-p} x_i^s}{\sum_{i=k_0-p+1}^{k_0} x_i^s},$$

for all $k \geq p$. □

In order to prove a lower bound on Expression 2 we make use of the results by Gal [6] and Schuierer [19] which we state here without proof and in a simplified form for completeness. Let $G_a = (1, a, a^2, \ldots)$ be the geometric sequence in a and $X^{+i} = (x_i, x_{i+1}, \ldots)$ the suffix of sequence X starting at x_i.

Theorem 1 ([19]). *Let $X = (x_0, x_1, \ldots)$ be a sequence of positive numbers, r an integer, and $a = \overline{\lim}_{n\to\infty}(x_n)^{1/n}$, for $a \in \mathbb{R} \cup \{+\infty\}$. If F_k, $k \geq 0$, is a sequence of functionals which satisfy*

1. *$F_k(X)$ only depends on $x_0, x_1, \ldots, x_{k+r}$,*
2. *$F_k(X)$ is continuous, for all $x_i > 0$, with $0 \leq i \leq k + r$,*
3. *$F_k(\alpha X) = F_k(X)$, for all $\alpha > 0$,*
4. *$F_k(X + Y) \leq \max(F_k(X), F_k(Y))$, and*
5. *$F_{k+i}(X) \geq F_k(X^{+i})$, for all $i \geq 1$,*

then

$$\sup_{0 \leq k < \infty} F_k(X) \geq \sup_{0 \leq k < \infty} F_k(G_a).$$

In particular, in our case it is easy to see that, if we set

$$F_k(X^s) = 1 + 2 \sum_{i=0}^{k+m-p} x_i^s \Big/ \sum_{i=k-p+1}^{k} x_i^s,$$

then F_k satisfies all conditions of Theorem 1. Hence,

$$C_S \geq \sup_{0\leq k<\infty} F_k(X^s) \geq \sup_{0\leq k<\infty} F_k(G_a) = \sup_{0\leq k<\infty} 1 + 2\frac{\sum_{i=0}^{k+m-p} a^i}{\sum_{i=k-p+1}^{k} a^i}.$$

Note that if $a \leq 1$, then the above ratio tends to infinity as $k \to \infty$. Hence, we can assume that $a > 1$ and obtain

$$\begin{aligned} C_S &\geq \sup_{0\leq k<\infty} 1 + 2\frac{(a^{k+m-p+1}-1)/(a-1)}{(a^{k+1}-a^{k-p+1})/(a-1)} \\ &= \sup_{0\leq k<\infty} 1 + 2\frac{a^{k+m-p+1}-1}{a^{k+1}-a^{k-p+1}} \\ &\overset{(a>1)}{=} 1 + 2\frac{a^{m-p}}{1-a^{-p}} = 1 + 2\frac{a^m}{a^p-1}. \end{aligned}$$

The above expression is minimized for $a = (m/(m-p))^{1/p}$ and the competitive ratio is bounded from below by

$$C_S \geq 1 + 2\frac{\left(\frac{m}{m-p}\right)^{m/p}}{\frac{m}{m-p}-1} = 1 + 2\left(\frac{m}{p}-1\right)\left(\frac{m}{m-p}\right)^{m/p}.$$

Theorem 2. *There is no search strategy for a target on m rays using p robots with a competitive ratio of less than*

$$1 + 2\left(\frac{m}{p}-1\right)\left(\frac{m}{m-p}\right)^{m/p}.$$

Note that the above expression interpolates nicely between the various special cases that may occur. For instance, if $p = 1$, then we obtain $1+2\,m^m/(m-1)^{m-1}$ as previously shown [1,6]. If there is an integer number of rays per robot, say $m = kp$ for some integer constant k, then we obtain

$$1 + 2\left(\frac{kp}{p}-1\right)\left(\frac{kp}{kp-p}\right)^{kp/p} = 1 + 2(k-1)\frac{k^k}{(k-1)^k} = 1 + 2\frac{k^k}{(k-1)^{k-1}},$$

that is, the same competitive ratio as if each of the robots searches on a separate subset of k rays.

4 An Optimal Strategy

We now present a strategy that achieves a competitive ratio matching the lower bound we have shown above. The strategy works as follows. The robots explore the rays in a fixed cyclic order. Let $a = (m/(m-p))^{1/p}$. The sequence of return distances of the robots is given by $x_i = a^i$. The kth time that robot R returns to the origin it chooses to explore ray $(kp + R) \bmod m$ up to distance x_{kp+R}. Obviously, the ith time ray r is explored, the robot explores it up to distance x_{im+r}.

So let r be a ray that is explored by robot R after it has returned the kth time to the origin. Hence, $kp + R = r \bmod m$, or equivalently $kp + R = im + r$. The total distance traveled thus far by robot R is $2\sum_{j=0}^{k-1} x_{jp+R}$. Clearly, the robot that explored ray r up to distance $x_{(i-1)m+r}$ reached the origin before robot R. Hence, r has been explored up to distance $x_{(i-1)m+r}$ when robot R travels on it and the competitive ratio in this step is given by

$$1 + 2\frac{\sum_{j=0}^{k-1} x_{jp+R}}{x_{kp+R-m}} = 1 + 2\frac{a^R \sum_{j=0}^{k-1}(a^p)^j}{a^R a^{kp-m}} = 1 + 2\frac{a^{kp}-1}{(a^p-1)a^{kp-m}}$$
$$\leq 1 + 2\frac{a^m}{a^p-1} = 1 + 2\left(\frac{m}{p}-1\right)\left(\frac{m}{m-p}\right)^{m/p}.$$

Since the bound is independent of the robot R, the ray r and the number of times the ray was visited, we obtain the following theorem.

Theorem 3. *There exists a strategy for searching for a target on m rays using p robots with a competitive ratio of*

$$1 + 2\left(\frac{m}{p}-1\right)\left(\frac{m}{m-p}\right)^{m/p}$$

which is optimal.

5 Conclusions

We present an optimal strategy for searching for a target on m concurrent rays in parallel using p robots. This strategy has a competitive ratio of

$$1 + 2\left(\frac{m}{p}-1\right)\left(\frac{m}{m-p}\right)^{m/p}.$$

This is a generalization of the on-line construction of on-line heuristics to a distributed model. It also extends the cow path problem to multiple searchers on m concurrent rays, which has proven to be a basic primitive in the exploration of certain classes of polygons. Furthermore, it expands the field of target searching to multiple robots; a setting that more closely reflects real-world scenarios. An open problem is to generalize this algorithm to randomized or average case strategies. In similar settings, a trade-off theorem between average and worst case performance of search strategies is known. It is natural to expect that a similar result might hold for parallel searches.

References

1. R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106:234–252, 1993.
2. R. Baeza-Yates and R. Schott. Parallel searching in the plane. *Comput. Geom. Theory Appl.*, 5:143–154, 1995.
3. Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal learning of an unknown environment. In *Sixth ACM Conference on Computational Learning Theory (COLT 93)*, pages 277–286, July 1993.
4. A. Datta, Ch. Hipke, and S. Schuierer. Competitive searching in polygons—beyond generalized streets. In *Proc. Sixth Annual International Symposium on Algorithms and Computation*, pages 32–41. LNCS 1004, 1995.
5. A. Datta and Ch. Icking. Competitive searching in a generalized street. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 175–182, 1994.
6. S. Gal. *Search Games*. Academic Press, 1980.
7. M. Hammar, B. Nilsson, and S. Schuierer. Parallel searching on m rays. In *Symp. on Theoretical Aspects of Compute Science*, pages 132–142. LNCS 1563, 1999.
8. Ch. Hipke. Online-Algorithmen zur kompetitiven Suche in einfachen Polygonen. Master's thesis, Universität Freiburg, 1994.
9. M.-Y. Kao, Y. Ma, M. Sipser, and Y. Yin. Optimal constructions of hybrid algorithms. *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pp. 372–381, 1994.
10. M.-Y. Kao, J. H. Reif, and S. R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 441–447, 1993.
11. R. Klein. Walking an unknown street with bounded detour. *Comput. Geom. Theory Appl.*, 1:325–351, 1992.
12. J. M. Kleinberg. On-line search in a simple polygon. In *Proc. of 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 8–15, 1994.
13. A. López-Ortiz. *On-line Searching on Bounded and Unbounded Domains*. PhD thesis, Department of Computer Science, University of Waterloo, 1996.
14. A. López-Ortiz and S. Schuierer. Generalized streets revisited. In M. Serna J. Diaz, editor, *Proc. 4th European Symp. on Algorithms*, pp. 546–558. LNCS 1136, 1996.
15. A. López-Ortiz and S. Schuierer. The ultimate strategy to search on m rays? *Theoretical Computer Science*, 261(1):267–295, 2001.
16. A. López-Ortiz and G. Sweet. Parallel searching on a lattice. In *Proc. 13th Canadian Conference on Computational Geometry*, pages 125–128, 2001.
17. A. Mei and Y. Igarashi. Efficient strategies for robot navigation in unknown environment. In *Proc. of 21st Intl. Colloquium on Automata, Languages and Programming*, pages 51–56, 1994.
18. C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. In *Proc. 16th Internat. Colloq. Automata Lang. Program.*, volume 372 of *Lecture Notes in Computer Science*, pages 610–620. Springer-Verlag, 1989.
19. S. Schuierer. Lower bounds in on-line geometric searching. *Computational Geometry: Theory and Applications*, 18(1):37–53, 2001.
20. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
21. Y. Yin. *Teaching, Learning, and Exploration*. PhD thesis, Department of Mathematics and Laboratory for Computer Science, MIT, 1994.

Analysis of Heuristics for the Freeze-Tag Problem

Marcelo O. Sztainberg[1], Esther M. Arkin[1], Michael A. Bender[2], and Joseph S.B. Mitchell[1]

[1] Dept. of Applied Mathematics and Statistics, SUNY, Stony Brook, NY 11794, USA
{mos,estie,jsbm}@ams.sunysb.edu
[2] Dept. of Computer Science, SUNY, Stony Brook, NY 11794, USA
bender@cs.sunysb.edu

Abstract. In the Freeze Tag Problem (FTP) we are given a swarm of n asleep (*frozen* or *inactive*) robots and a single awake (*active*) robot, and we want to awaken all robots in the shortest possible time. A robot is awakened when an active robot "touches" it. The goal is to compute an optimal *awakening schedule* such that all robots are awake by time t^*, for the smallest possible value of t^*. We devise and test heuristic strategies on geometric and network datasets. Our experiments show that all of the strategies perform well, with the simple greedy strategy performing particularly well. A theoretical analysis of the greedy strategy gives a tight approximation bound of $\Theta(\sqrt{\log n})$ for points in the plane. We show more generally that the (tight) performance bound is $\Theta((\log n)^{1-1/d})$ in d dimensions. This is in contrast to general metric spaces, where greedy is known to have a $\Theta(\log n)$ approximation factor, and no method is known to achieve an approximation bound of $o(\log n)$.

1 Introduction

We consider a natural problem that arises in the study of *swarm robotics*. Consider a set of n *robots*, modeled as points in some metric space. There is one "awake" *source* robot; all other robots are *asleep* (inactive). In order to awaken a sleeping robot, an active robot travels to it and touches it; then, that robot can assist the set of active robots in awakening other asleep robots. Our goal is to activate (wake up) all of the robots as quickly as possible; i.e., we want to minimize the *makespan*, which is the time when the last robot is awakened.

This problem has been coined the *freeze-tag problem* (FTP) [1] because of its similarity with the children's game of "freeze-tag." In the game, the person who is "it" tags a player, who becomes "frozen" until another player (who is not "it" and not "frozen") tags him to unfreeze him. The FTP arises when there are a large number of players that are frozen, and one (not "it") unfrozen player, whose goal it is to unfreeze the rest of the players as quickly as possible. Once a player gets unfrozen, s/he is available to assist in unfreezing other frozen players, who can then assist, etc. Other applications of the FTP arise in the context of distributing data (or some other commodity), where physical proximity is required

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 270–279, 2002.

for distribution. This proximity may be necessary because wireless communication has too high a bandwidth cost or security risk. How does one propagate the data to the entire set of participants in the most efficient manner? Prior work on the dissemination of data in a graph includes the *minimum broadcast time problem*, the *multicast problem*, and the related *minimum gossip time problem*; see [4] for a survey and [2,6] for recent approximation results.

The FTP is expressed as a combinatorial optimization problem as follows: Given a set of points in a metric space, find an arboresence (*awakening tree*) of minimum height where every node has out-degree at most two.

What makes the freeze-tag problem particularly intriguing is that *any* reasonable ("nonlazy") strategy yields an $O(\log n)$-approximation (Proposition 1.1 of [1]), whereas no strategy is known for general metric spaces that yields a $o(\log n)$-approximation. Arkin et al. [1] show that even very simple versions of the problem (e.g., on star metrics) are NP-complete. They give an efficient polynomial-time approximation scheme (PTAS) for geometric instances on a set of points in any constant dimension d. They also give results on star metrics, where $O(1)$-approximation is possible, and an $o(\log n)$-approximation for the special case of *ultrametrics*.

In this paper, our main results include the following:

(1) A proof that the natural greedy heuristic applied to geometric instances gives an $O((\log n)^{1-1/d})$-approximation in d dimensions. Thus, in one dimension, the greedy heuristic yields an $O(1)$-approximation, and in the plane the greedy heuristic yields an $O(\sqrt{\log n})$-approximation. We prove that this analysis is tight by supplying matching lower bounds.
(2) We perform an experimental investigation of heuristic strategies for the FTP, comparing the different choices of greedy strategies and comparing these greedy strategies with other heuristics.

Notation. We let S denote the *swarm*, the set of n points in a metric space (often Euclidean d-space, denoted $\Re^d$) where the initially asleep robots are located. We let s denote the *source point* where an initially active source robot is placed. We assume that any active robot in motion travels with unit speed. We let R denote the *radius* of the swarm S with respect to s; i.e., $R = \max_{p \in S} \mathrm{dist}(s, p)$. We let $D = \max_{p,q \in S \cup \{s\}} \mathrm{dist}(p, q)$ denote the diameter of the set of robots. We let t^* denote the minimum makespan. Note that, trivially, $t^* \geq R$ (since robots move with unit speed).

2 Wakeup Strategies for the FTP

2.1 Greedy Strategies

A natural strategy for the FTP is the greedy strategy, where an awake robot chooses the nearest asleep robot to awaken. The motivation for awakening nearby robots first is that parallelism is generated early on: the first robot awakens its closest neighbors, these newly awakened robots awaken their closest sleeping neighbors; thus there is rapid exponential growth in the number of awake robots.

In fact the greedy strategy is not a fully defined heuristic. What remains to be specified is how conflicts among robots are resolved, since two robots may have the same closest neighbor. We now describe three methods for resolving these conflicts: claims, refresh, and delayed target choice.

Claims. In order to guarantee that work is not duplicated, an awake robot *claims* the sleeping robot that it intends to awaken. Once a sleeping robot is claimed, no other robot is allowed to claim or awaken it.

Refresh. It may be beneficial for newly awakened robots to "renegotiate" the claims, since the set of awake robots changes. We refer to the ability to reassign active robots to asleep robots as the option to *refresh* claims.

We first consider the case in which claims are *not* adjusted. Thus, once an awake robot A claims an asleep robot B, A awakens B before making any other algorithmic decisions. The algorithm is now well defined because, without loss of generality, at most one robot is awakened at a time, and each time a robot is awakened it claims the nearest asleep robot.

Consider now the case in which claims *are* renegotiated, or *refreshed*, each time a new robot awakens. For motivation, consider the following scenario. An awake robot A is heading toward a sleeping robot B, which A has claimed. Before A reaches B, another robot C awakens. Now, both A and C would like to claim B, but since B is closer to C than to A, C takes over the responsibility of awakening B.

In our experiments, we assign claims by finding a matching between the awake robots and the asleep robots. We use a (potentially suboptimal) greedy strategy to compute a matching, rather than applying a more complex optimization algorithm. We order, by length, the potential matching edges between the set of currently awake and currently sleeping robots. We iteratively add the shortest edge e to the matching, and remove from consideration those edges that are incident to either of e's endpoints. The resulting matching has the property of giving priority to the short matching edges, which is faithful to the greedy heuristic.

Delayed Target Choice. Refresh introduces several anomalies. In particular, an (awake) robot may repeatedly change directions and oscillate without awakening any robots. This happens when an (awake) robot chooses an asleep target, but before the robot reaches its target, another robot claims the target. With the *delayed target choice* option we avoid these oscillations by making the solution "more off-line". Specifically, we avoid committing to the direction a robot is heading until that robot has traveled far enough to awaken its target, at which point the target (and its position) is fully determined.

Lower Bounds on the Performance of the Greedy Heuristic. Our lower bounds hold for all variations of the greedy heuristic described above, since, in our lower bound examples, all awake robots will travel together in a "pack".

Theorem 1. *For any $\epsilon > 0$, there exists an instance of the FTP for points $S \subset \Re^1$ on a line ($d = 1$) for which the greedy heuristic results in a makespan that is at least $4 - \epsilon$ times optimal.*

Theorem 2. *The greedy heuristic is an $\Omega(\sqrt{\log n})$-approximation to the FTP in the plane.*

Proof: We arrange $\log n$ disks, each of radius 1, along a "zig-zag" path having $\sqrt{\log n}$ rows each having $\sqrt{\log n}$ disks, with disks touching but not overlapping along the path. Refer to Figure 1. The source robot is at the center, s, of the first disk. There is one asleep robot at the center of the second disk, then two at the center of the next, then four, etc, with the number of asleep robots at the center of each disk doubling as we advance along the path of touching disks. The last disk has (about) $\frac{n}{2}$ robots. When the greedy strategy is applied to this example, the awakening happens in sequence along the zig-zag path, with all of the newly awakened robots at the center of one disk targeting the robots at the center of the next disk. An optimal strategy, however, sends the source robot vertically upwards, awakening one cluster of robots at the center of each disk it passes along the way. These robots are then available to travel horizontally, awakening the robots along each row. This strategy yields makespan $t^* = O(\sqrt{\log n})$. Hence, greedy is an $\Omega(\sqrt{\log n})$-approximation. □

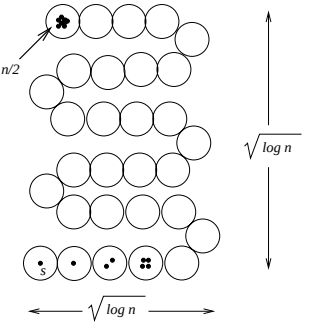


Fig. 1. The lower bound construction in $\Re^2$.

Theorem 3. *The greedy strategy is an $\Omega((\log n)^{1-1/d})$-approximation to the FTP in $\Re^d$.*

Upper Bounds for the Greedy Heuristic. We first show that the lower bound of Theorem 1 is essentially tight for the one-dimensional case.

Theorem 4. *The greedy heuristic for a swarm $S \subset \Re$ of points on a line ($d = 1$) is a 4-approximation.*

The following theorems show that, in contrast with arbitrary metric spaces (where greedy gives an $O(\log n)$-approximation for the FTP), greedy is an $o(\log n)$-approximation for geometric instances. Combined with our lower bounds, we get a *tight* analysis of the approximation factor for greedy in all dimensions $d \geq 1$.

Theorem 5. *The greedy strategy is an $O(\sqrt{\log n})$-approximation to the FTP in the plane. One can compute the greedy awakening tree in time $O(n \log^{O(1)} n)$.*

Theorem 6. *The Greedy strategy is an $O((\log n)^{1-1/d})$ approximation to the FTP in d dimensions. One can compute the greedy awakening tree in time $O(n \log^{O(1)} n)$.*

In both theorems, the algorithmic complexity follows from applying dynamic methods for nearest neighbor search (or approximate nearest neighbor search [3, 5]) in order to identify which asleep robot is closest to a newly awakened robot. The nearest neighbor search requires deletions, since robots get deleted from the set of candidate targets when they awaken.

We concentrate on proving the approximation bound for the 2-dimensional case (Theorem 5); the d-dimensional case is a fairly direct generalization.

Suppose that we are given a greedy awakening tree for a swarm of size $|S| = n$. We can convert this tree into an awakening tree (at no increase in makespan) having the following property: on any root-to-leaf path there is at most one non-leaf node having out-degree 1 (rather than 2). Based on this property, and a simple counting argument, there exists a root-to-leaf path, $P = (p_0, p_1, \ldots, p_K)$, having $K \leq \log n$ edges of lengths $r_i = \text{dist}(p_i, p_{i+1})$. At the time t when the last node p_K is reached, all of the remaining asleep robots are robots that will be awakened by other branches of the awakening tree (by definition of the awakening tree). The makespan, therefore, must be at most $t + D \leq t + 2R$. We will show that $t = O(\sqrt{\log n} \cdot t^*)$, implying that the makespan is $O(\sqrt{\log n} \cdot t^*)$

Fixing attention now on one root-to-leaf path, P, in the awakening tree, we define the *outer circle* C_i to be the circle centered at p_i with radius $r_i = \text{dist}(p_i, p_{i+1})$; the *inner circle* c_i is the circle centered at p_i with radius $\frac{1}{3}r_i$. The properties of the greedy heuristic ensure the following claim:

Claim **1** *None of the points $p_{i+1}, \ldots, p_K$ lie inside circle C_i.*

The proof of Theorem 5 is based on an *area-covering argument*. Specifically, we provide a bound on the area covered by the circles $C_0, C_1, \ldots C_{K-1}$, showing that $\sum_{i=0}^{K-1} r_i^2 = O(R^2)$, where R is the radius of the swarm. The subtlety of the proof is that neither the outer circles $C_0, \ldots, C_{K-1}$ nor even the inner circles $c_0, \ldots, c_{K-1}$ are disjoint. The proof is based on the following lemma:

Lemma 1. *The combined area covered by the (inner or outer) circles is $O(R^2)$;*

$$\sum_{i=0}^{K-1} \textit{area}(C_i) = O\left(\sum_{i=0}^{K-1} \textit{area}(c_i)\right) = O\left(\sum_{i=0}^{K-1} r_i^2\right) = O(R^2).$$

Proof of Theorem 5: The length, L, of the root-to-leaf path P is simply $L = \sum_{i=0}^{K-1} r_i$. By the Cauchy-Schwartz inequality, the fact that $K \leq \log n$, and Lemma 1 we get

$$L = \sum_{i=0}^{K-1} r_i \leq \sqrt{K}\sqrt{\sum_{i=0}^{K-1} r_i^2} = O(R\sqrt{\log n}).$$

Since this result holds for any root-to-leaf path in the awakening tree, the approximation bound follows. □

Proof of Lemma 1: Each (outer,inner) circle pair, (C_i, c_i), is assigned to a *circle class* according to its radius. Without loss of generality, let the smallest

outer circle have radius 1. Define *class i* to be the set of (outer,inner) circle pairs such that the radius of the outer circle has radius r, with $2^{i-1} \leq r < 2^i$.

While pairs of outer circles (and pairs of inner circles) may overlap, using the property of circle classes, we show the following claim:

Claim **2** *Any pair of inner circles belonging to the same class are disjoint.*

Let area A_i be the area covered by inner circles of class i. We claim that

Claim **3** *In order for the inner circles of class i associated with path P to cover area A_i, the corresponding edges of P (associated with circle pairs of class i) must have total length ℓ_i satisfying $\frac{9A_i}{2^{i+1}\pi} \leq \ell_i \leq \frac{9A_i}{2^{i-2}\pi}$.*

Since along path P of the awakening tree we have circles of different classes appearing, not necessarily in order of size, we may have inner circles overlapping, thus not covering "new" area. We need the following claim:

Claim **4** *The length of P that does* not *correspond to edges whose inner circles cover new area is at most a constant fraction of the length that corresponds to edges whose inner circles do cover new area.*

Thus, by Claim 4, the distance traveled along P in which no new area is covered is of the order of the distance traveled in which new area is covered. Since the total area covered is $O(R^2)$, the claim of Lemma 1 follows. □

2.2 Other Heuristic Wakeup Strategies

The greedy strategy has the following weakness: it may be preferable for a robot to travel a longer distance to obtain a better payoff. In this section we examine alternative strategies in an attempt to overcome this weakness.

We design our alternative strategies while keeping in mind the actual application that motivated our study: the need to activate a swarm of small experimental robots, each equipped with certain sensors. The sensors on the actual robots in our project (joint with HRL Labs) have the feature that they sense other robots (or obstacles) in each of k sectors, evenly distributed around the (circular) robot. (Our robots have $k = 8$, implying 45-degree sectors.) Within each of its sectors, a robot can detect only the closest other robot. Thus, there are at most k options facing a robot once it is activated: Which sector should be selected? Once the sector is selected, the robot heads for the closest asleep robot it has sensed in that sector. A greedy strategy that uses sensor sectors will select the sector whose closest robot is closest.

Bang-for-the-Buck. A natural strategy, which we call "bang-for-the-buck", is to choose the next target to awaken based on maximizing the ratio of "value" ("bang") to "cost" ("buck"). There are a variety of heuristic measures of potential value; a particularly simple one is to consider the value of a sector to be the number of currently asleep robots in the sector. The cost of a sector is chosen to be the distance to the nearest asleep robot in the sector.

Random Sector Selection. The goal of this strategy is to "mix up" at random the directions in which robots head to awaken other robots. When a robot awakens, it selects, at random, a sector and chooses its next target to be the closest asleep robot in that sector.

Opposite Cone. In this strategy the goal is to enforce a certain amount of "mixing up" of directions that robots head, by sending a newly awakened robot in a nearly opposite direction from that of the robot that awakened it. In particular, suppose robot A heads due east to awaken robot B at point p; then, the next target that robot A selects will be chosen to be the closest asleep robot in a cone centered on a vector to the west, while the target for robot B to awaken next will be selected from a cone centered on a vector to the east.

3 Experiments

3.1 Experimental Setup

Our experiments are based on a Java simulation of our various strategies. All tests were performed on a PC running Linux OS. The graphical user interface permits the user to select the choice of strategy, the parameters, and the input dataset, and it optionally shows a graphical animation of the simulation.

Datasets. We tested our strategies on both geometric and non-geometric datasets. We investigated four classes of geometric datasets: (1) uniform over a large square (600-by-600); (2) cluster ($\sqrt{n}$ clusters, each of a random size, generated uniformly over a square of side length $c = 2\sqrt{n}$, whose upper left corner is uniformly distributed over the square); (3) square grid; and (4) a regular hexagonal grid. Since our non-greedy strategies are specified geometrically, they were applied only to the geometric data.

The greedy strategies were applied to *non*-geometric datasets, including: (1) Star metrics: the $n - 1$ asleep robots are at the leaves of a star (spoke lengths are chosen uniformly between 1 and n), the source robot is at the root. There is either one asleep robot per leaf (case "1-1") or many asleep robots per leaf (case "1-m": we generate $O(\sqrt{n})$ spokes and randomly assign m robots to each spoke, with m chosen uniformly between 1 and $O(\sqrt{n})$). (2) TSPLIB: symmetric traveling salesman files, with data of type MATRIX (14 instances).

Performance Measures. We maintain performance statistics, including: (1) *total time* of the simulation; (2) *total distance* traveled by all robots; and (3) *average distance* traveled by robots that do any traveling. We found that total distance and average distance were tightly correlated with the total time of the simulation; thus, here we report results only on the total time.

Parameter Choices. For each of the strategies, we considered each possible setting of the set of parameter choices (Claims, Refresh, or Delayed Target Choice) discussed in Section 2.1.

3.2 Experimental Results

The experiments on synthetically generated datasets were conducted as follows. For each choice of wakeup strategy, parameters, and dataset, a set of 100 runs was performed, with 10 runs for each value of $n \in \{100, 200, 300, \ldots, 1000\}$.

The experiments on datasets from the TSPLIB (EUC_2D or MATRIX) were done once per dataset, with one robot per point.

We performed runs for the following combinations of strategies and datasets: Greedy was run on all datasets (Uniform, Cluster, Grid, Hexagonal Grid, Stars 1-1, Stars 1-m, TSPLIB (EUC_2D and MATRIX)), while Bang-for-the-Buck, Random Sector Selection, and Opposite Cone were run on only the geometric instances (Uniform, Cluster, Grid, Hexagonal Grid, TSPLIB (EUC_2D)).

In our plots, the horizontal axis corresponds to the swarm size n, the vertical axis to the ratio of the makespan to the lower bound on makespan (the radius, R, except in some star-metric cases). Thus, the vertical axis shows an upper bound on the approximation ratio.

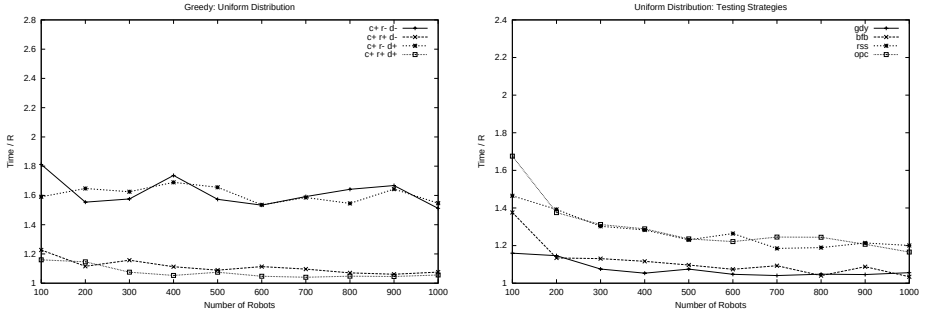


Fig. 2. Left: Greedy on uniform distributions: Choices of parameters for claims (c), refresh (r), and delayed target choice (d). A "+" ("-") indicates the parameter is set to *true* (*false*). **Right:** Greedy (gdy), Bang-for-the-Buck (bfb), Random Sector Selection (rss), and Opposite Cone (opc) tested on uniformly distributed swarms.

Parameter Choices. We experimented with various parameter choices over a common dataset. The claims option is very significant; without it the robots travel in groups rather than spreading out. There is a significant advantage in using the refresh option. While less significant, the delayed target choice is also advantageous. Figure 2 shows the result of running greedy on uniformly distributed points; other datasets yield similar results, with the delayed target choice showing a more pronounced advantage in the case of cluster datasets.

Wakeup Strategy Comparison. The main conclusion we draw from our experiments is that the greedy strategy is a very good heuristic, most often outperforming the other strategies in our comparisons. See Figure 3. As the size (n) of the swarm increases, the approximation ratios tend to stay about the same or decrease; we suspect this is because R becomes a better lower bound

for larger swarms. The upper bounds (Time/R) on approximation factors in the geometric instances are between 1.0 and 1.5. For star metrics, the ratio Time/R is significantly higher (between 2 and 3), but this is due to the fact that R is a poor lower bound in the case of stars. In order to verify this, we computed an alternative lower bound specifically for star metrics having one robot per leaf. (The lower bound is $2L_{min}(\lceil \log(n+1) \rceil - 1) + L_{max}$, where L_{min} (resp., L_{max}) is the length of the shortest (resp., longest) spoke of the star.) Figure 4(right) shows the results of greedy on stars 1-1 datasets (of various spoke length distributions) using this lower bound in computing the approximation ratio; we see that there is a striking improvement over using the lower bound of R (which is particularly poor for star metrics). We also give tables (Tables 1,2) showing the percentage of wins for each strategy, and, finally, report in Table 3 the results for greedy on the non-geometric TSPLIB instances.

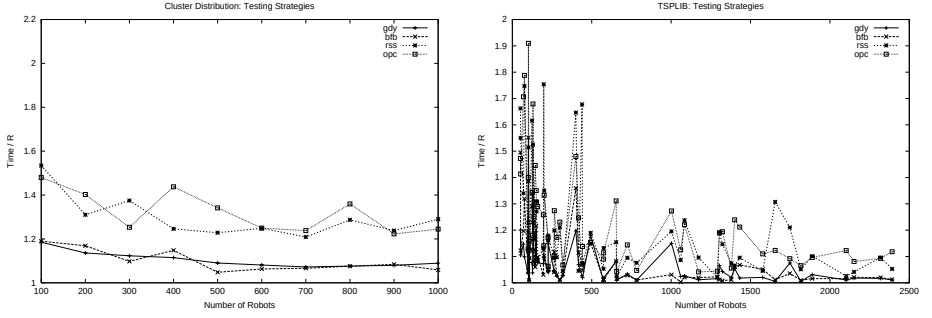


Fig. 3. Strategy comparison on cluster data (left) and TSPLIB EUC_2D data (right).

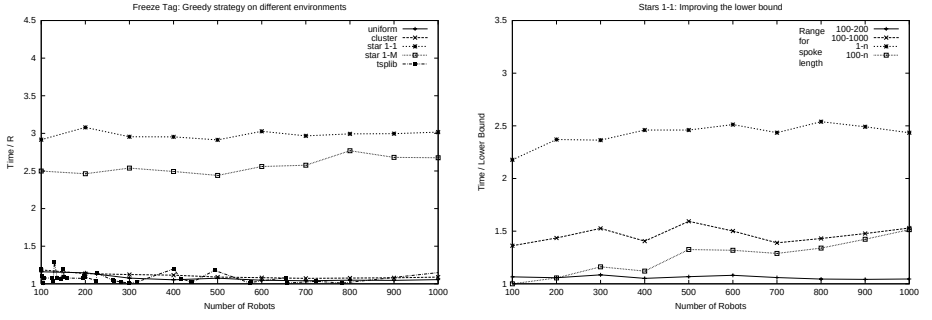


Fig. 4. Left: Comparing greedy on five different datasets. **Right**: Comparing greedy on star 1-1 datasets, *using the improved lower bound* of $2L_{min}(\lceil \log(n+1) \rceil - 1) + L_{max}$. Each curve corresponds to a randomly generated dataset having spoke lengths uniformly generated in the indicated interval.

Acknowledgements. We thank Doug Gage for discussions motivating this research. This research was partially supported by HRL Labs (DARPA subcon-

tract), NASA Ames Research, NSF, Sandia, and the U.S.-Israel Binational Science Foundation.

Table 1. Left: Comparing strategies on the 68 TSPLIB (EUC_2D) datasets. Right: Winning strategies for the TSPLIB (EUC_2D) datasets: For those runs in which a strategy outperformed the others, we compute the maximum, minimum and average approximation factor and percent by which it lead over the second-place strategy.

Algorithm	Wins	%
GDY	45	66.20
BFB	22	32.35
RSS	1	1.45
OPC	0	0.00

Algorithm	Approx. Rate			Winning %		
	Best	Worst	Avg	Max	Min	Avg
GDY	1.00	1.29	1.06	41.13	0.01	6.20
BFB	1.00	1.36	1.07	18.71	2.24	3.97
RSS	1.04	1.04	1.04	2.24	2.24	2.24

Table 2. Comparing strategies on uniform datasets.

Algorithm	Win %
GDY	62
BFB	37
RSS	0
OPC	1

Algorithm	Approx. Rate			Winning %		
	Best	Worst	Avg	Max	Min	Avg
GDY	1.01	1.31	1.06	57.32	0.33	8.83
BFB	1.00	1.16	1.04	27.23	0.13	4.04
OPC	1.23	1.23	1.23	5.25	5.25	5.25

Table 3. Results of greedy strategy for TSPLIB MATRIX (non-geometric) instances.

# of Robots	17	21	24	26	42	42	48	48	58	120	175	535	561	1032
Approx. Rate	1.06	1.04	1.16	1.36	1.08	1.12	1.03	1.24	1.19	1.12	3.26	3.01	1.17	3.19

References

1. E. M. Arkin, M. A. Bender, S. P. Fekete, J. S. B. Mitchell, and M. Skutella. The freeze-tag problem: How to wake up a swarm of robots. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pp. 568–577, 2002.
2. A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Multicasting in heterogeneous networks. In *Proc. 30th ACM Sympos. Theory of Comput.*, pp. 448–453, 1998.
3. S. N. Bespamyatnikh. Dynamic algorithms for approximate neighbor searching. In *Proc. 8th Canad. Conf. Comput. Geom.*, pp. 252–257, 1996.
4. S. M. Hedetniemi, T. Hedetniemi, and A. L. Liestman. A Survey of Gossiping and Broadcasting in Communication Networks. *NETWORKS*, 18:319–349, 1988.
5. S. Kapoor and M. Smid. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Comput.*, 25:775–796, 1996.
6. R. Ravi. Rapid rumor ramification: Approximating the minimum broadcast time. *Proc. 35th Ann. Sympos. on Foundations of Computer Sci.*, pp. 202–213, 1994.
7. `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`.

Approximations for Maximum Transportation Problem with Permutable Supply Vector and Other Capacitated Star Packing Problems

Esther M. Arkin[1], Refael Hassin[2], Shlomi Rubinstein[2], and Maxim Sviridenko[3]

[1] Department of Applied Mathematics and Statistics, SUNY Stony Brook, Stony Brook, NY 11794-3600, estie@ams.sunysb.edu.†

[2] Department of Statistics and Operations Research, Tel-Aviv University, Tel-Aviv 69978, Israel, {hassin,shlomiru}@post.tau.ac.il

[3] IBM T. J. Watson Research Center, Yorktown Heights, P.O. Box 218, NY 10598, USA; sviri@us.ibm.com

1 Introduction

The input to the TRANSPORTATION PROBLEM consists of a complete weighted bipartite graph $G = (V_1, V_2, w)$, integer *supplies* $a_i \geq 0$, $i \in V_1$, and integer *demands* $b_j \geq 0$, $j \in V_2$, where w.l.o.g. $\sum_{i \in V_1} a_i = \sum_{j \in V_2} b_j$. The problem is to compute *flows* x_{ij} $i \in V_1$, $j \in V_2$ such that $\sum_j x_{ij} = a_i$ for every $i \in V_1$, $\sum_i x_{ij} = b_j$ for every $j \in V_2$, and $\sum_E w_{ij} x_{ij}$ is maximized (or minimized). The transportation problem is polynomially solvable even when the flows are required to be integers.

The MAXIMUM TRANSPORTATION PROBLEM WITH PERMUTABLE SUPPLY VECTOR (MTPPSV) is a variation of the transportation problem, where supplies are not attached to the vertices of V_1, but rather to a set of *facilities* that have to be located at the vertices of V_1, one facility at each vertex. Thus, the problem is both to decide on the location of the facilities associated with the given set $\{a_i\}$ to V_1 and on the flows between V_1 and V_2 so as to maximize the total profit. The problem is NP-hard. See [6,7] for applications.

A closely related problem is MAXIMUM CAPACITATED STAR PACKING. Its input consists of a complete undirected graph $G = (V, E)$ with a weight function $w : E \to I\!R_+$ and a vector $c = (c_1, ..., c_p)$ of integers. We use $w(E')$ to denote the total weight of a subset E' of edges. A *star* is a subset of edges with a common vertex called the *center* of the star. The other vertices are *leaves* of the star. The *size*, or *capacity* of a star is its number of edges. The *weight* of a star is the total weight of its edges. The MAXIMUM CAPACITATED STAR PACKING problem requires to compute a set of vertex-disjoint stars in G of sizes $c_1, ..., c_p$, so as to maximize their total weight, where $\sum_{i=1}^{p} c_i = |V| - p$. The problem can be thought of as a facility location problem. Facilities of given sizes c_i are to be

† Partially supported by NSF (CCR0098172).

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 280–287, 2002.

located at vertices to serve customers, where the profit is given by the weight of the edge.

Previous work. The complexity of minimum transportation problem with permutable demand vector was studied by Meusel and Burkard [6] and Hujter, Klinz and Woeginger [4]. Most of their results are valid for maximization problem, too, e.g. the MTPPSV with $a_i \in \{0, 1, 2\}$ is polynomially solvable by reduction to the maximum weight f-factor problem [4]. Wolsey [8] analyzed 'greedy' heuristics for several discrete facility location problems in which monotone submodular functions are maximized. In particular his results imply $(1 - e^{-1})$-approximation for the special case of MTPPSV with $b_j = 1, j \in V_2$ and $a_i \in \{0, A\}, i \in V_1$ for some positive integer A. Unfortunately his approach cannot be generalized for the general MTPPSV since the objective function of MTPPSV is not necessary submodular.

The MAXIMUM QUADRATIC ASSIGNMENT PROBLEM is a generalization of the MAXIMUM CAPACITATED STAR PACKING problem as well as many other problems. Three $n \times n$ nonnegative symmetric matrices $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$ are given and the objective is to compute a permutation π of $V = \{1, \ldots, n\}$ so that $\sum_{\substack{i,j \in V \\ i \neq j}} a_{\pi(i),\pi(j)} b_{i,j} + \sum_{i \in V} c_{i,\pi(i)}$ is maximized. A $\frac{1}{4}$-approximation algorithm for the MAXIMUM QUADRATIC ASSIGNMENT problem assuming the triangle inequality on matrix B was given in [3]. The problem is also a special case of the MAXIMUM k-SET PACKING PROBLEM where k is a maximum star size. The results of Arkin and Hassin [2] imply that a local search algorithm is almost a $\frac{1}{k-1}$-approximation algorithm for this problem.

Our results. We prove that

- A greedy type algorithm is a $\frac{1}{2}$-approximation for MTPPSV.
- A local search algorithm can be made arbitrarily close to be a $\frac{1}{2}$-approximation algorithm for the MTPPSV.
- A low-depth local search algorithm is a $\frac{1}{3}$-approximation algorithm for MAXIMUM CAPACITATED STAR PACKING.
- A matching-based algorithm is a $\frac{1}{2}$-approximation algorithm for MAXIMUM CAPACITATED STAR PACKING if the edge weights satisfy the triangle inequality.

2 Star Packing in Bipartite Graphs

In this section we consider the MAXIMUM CAPACITATED STAR PACKING PROBLEM IN BIPARTITE GRAPHS. An instance of this problem consists of a weighted bipartite graph $G = (V_1, V_2, E, w)$. We must locate p centers of stars of cardinality $c_1 \geq \ldots \geq c_p$ at the vertices of V_1 and assign vertices of $V_2, |V_2| = \sum_{i=1}^{p} c_i$ to star centers satisfying the cardinality constraints on sizes of stars. This problem can be represented as MTPPSV where V_1 are supply vertices with supply vector $c_1, \ldots, c_p, 0, \ldots, 0$ and demand vertices V_2 with demand vector $1, \ldots, 1$.

2.1 Greedy Algorithm

Algorithm GR in Figure 1 is a greedy algorithm that modifies the weights of edges. It selects in each iteration a maximum weight star with respect to modified weights which reflect a deletion of an edge from the partial solution, when a new edge entering the same vertex is selected. GR outputs a partial solution, i.e. a set of stars $S_1, \ldots, S_p$ such that $|S_i| \leq c_i$. This solution can be completed to the feasible capacitated star packing without decreasing its value. The modified

GR
begin
Let $\bar{w}$ *be a weight function on edges such that* $\bar{w} = w$.
for $i = 1, ...p$
Let $S_i = (v_i, X_i)$ *be a star of maximum weight with respect to* $\bar{w}$ *such that* $v_i \in V_1$, $X_i \subseteq V_2$, $|X_i| = c_i$.
Delete from $S_1, ..., S_{i-1}$ *the edges which enter* X_i, *delete* v_i *from* V_1.
for every $x \in X_i$ *and* $y \in V_1$
$\bar{w}_{yx} := \bar{w}_{yx} - \bar{w}_{v_i x}$.
return $S_1, ..., S_p$.
end *GR*

Fig. 1. Algorithm GR

weight of the edge (y, x) in Step i is equal to its original weight minus the weight of the edge from the current approximate solution touching it. Therefore, the total increase in objective function in Step i is equal to the sum of original weights of the star S_i minus the sum of original weights of edges deleted on this step.

Theorem 2.1. *Let apx be the weight of an arbitrary completion of the partial solution returned by GR. Then* $apx \geq opt/2$.

Proof: In Step i of GR we have stars $S_1, \ldots, S_i$ in our approximate solution where $|S_k| \leq c_k, k = 1, \ldots, i$. The size of S_k is exactly c_k during the k-th step of the algorithm but it can be decreased in future steps. On each step the set V_1 and the weight function $\bar{w}$ are modified, so at the end of Step i of the algorithm we have an instance I_i of the problem on the bipartite graph (V_1, V_2, E) and weight function $\bar{w}_i$. Let OPT_i be an optimal solution of the STAR PACKING IN BIPARTITE GRAPHS with star sizes $c_{i+1}, \ldots, c_p$ on the graph $(V_1, V_2, E, \bar{w}_i)$. We will prove that for $i = 1, \ldots, p$,

$$2\bar{w}_{i-1}(S_i) \geq \bar{w}_{i-1}(OPT_{i-1}) - \bar{w}_i(OPT_i). \tag{1}$$

By convention we assume that $\bar{w}_p(OPT_p) = 0$. By summing (1) over $i = 1, ..., p$ we get

$$2\sum_{i=1}^{p} \bar{w}_{i-1}(S_i) \geq \sum_{i=1}^{p} [\bar{w}_{i-1}(OPT_{i-1}) - \bar{w}_i(OPT_i] = \bar{w}_0(OPT_0) - \bar{w}_p(OPT_p) = opt.$$

Note that $\bar{w}_{i-1}(S_i)$ is the weight of star S_i in Step i, we emphasize it since some edges of S_i could be removed in the future but their weight is compensated by changing the weight function, i.e. sum of original edge weights of final stars is at least $\sum_{i=1}^{p} \bar{w}_{i-1}(S_i)$ and therefore, $apx \geq \sum_{i=1}^{p} \bar{w}_{i-1}(S_i) \geq opt/2$. To prove (1) we construct a solution SOL_i to I_i such that

$$2\bar{w}_{i-1}(S_i) \geq \bar{w}_{i-1}(OPT_{i-1}) - \bar{w}_i(SOL_i). \tag{2}$$

Assume that GR chooses the star S_i with the center v_i on the Step i. If OPT_{i-1} has the star with same index (i.e. the star with cardinality c_i) located at v_i then we define SOL_i from OPT_{i-1} by deleting vertex v_i from V_1 and star of cardinality c_i located at this vertex from OPT_{i-1}. In this case

$$\bar{w}_{i-1}(S_i) \geq \bar{w}_{i-1}(OPT_{i-1}) - \bar{w}_{i-1}(SOL_i),$$

and we get the inequality (2) by noticing that $\bar{w}_i(SOL_i) + \bar{w}_{i-1}(S_i) \geq \bar{w}_{i-1}(SOL_i)$ since $\bar{w}_{i-1}(S_i)$ is the total change in weight in Step i.

In general OPT_{i-1} can have another star S located at v_i. In this case OPT_{i-1} has the star of cardinality c_i located at another vertex, say v, we will call this star $S_i(v)$. Let $S' = S \setminus S_i$ be the part of star S which was not used in the star S_i. Since $c_1 \geq \cdots \geq c_p$, we know that $|S_i| \geq |S| \geq |S'|$. We construct SOL_i from OPT_{i-1} as follows. First we delete stars $S_i(v)$ and S from OPT_{i-1}. We do that since we cannot have star with index i in SOL_i and since vertex v_i is busy by S_i. After that we are trying to recover the weight we lost when we deleted S' (generally speaking, we didn't loose $S \setminus S' = S \cap S_i$ since these edges are used in the greedy solution) by moving the center of S' from v_i to v and using $|S'|$ heaviest edges with respect to the weight function $\bar{w}_{i-1}$ from the $|S_i|$ available edges at v. We call this new star $S'(v)$. We define SOL_i from OPT_{i-1} by removing vertex v_i from OPT_{i-1} and by defining new star $S'(v)$ at the vertex v instead of $S_i(v)$.

Denote by $av(C)$ the average weight, with respect to $\bar{w}_{i-1}$, of an edge in the star C. By the greedy property, S_i uses the heaviest edges touching vertex v_i. Also their total weight is at least the weight of $S_i(v)$. Therefore, $av(S_i) \geq av(S')$ and also $av(S_i) \geq av(S_i(v))$. Since $S'(v)$ consists of the heaviest edges in $S_i(v)$, it also follows that $av(S_i) \geq av(S_i(v)) \geq av(S_i(v) \setminus S'(v))$. By construction, $|S_i| = |S'| + |S_i(v) \setminus S'(v)|$. Therefore,

$$\bar{w}_{i-1}(S_i) = av(S_i)\ |S_i| \geq av(S')\ |S'|\ +\ av(S_i(v) \setminus S'(v))\ |S_i(v) \setminus S'(v)| = \bar{w}_{i-1}(S')\ +\ \bar{w}_{i-1}(S_i(v) \setminus S'(v)) = \bar{w}_{i-1}(S')\ +\ \bar{w}_{i-1}(S_i(v))\ -\ \bar{w}_{i-1}(S'(v)),$$

where the last equality follows since $S'(v) \subseteq S_i(v)$. Then,

$$\begin{aligned}\bar{w}_{i-1}(SOL_i) = w_{i-1}(OPT_{i-1}) - \bar{w}_{i-1}(S) - \bar{w}_{i-1}(S_i(v)) + \bar{w}_{i-1}(S'(v)) = \\ \bar{w}_{i-1}(OPT_{i-1}) - \bar{w}_{i-1}(S \cap S_i) - \bar{w}_{i-1}(S') - \bar{w}_{i-1}(S_i(v)) + \bar{w}_{i-1}(S'(v)) \geq \\ \bar{w}_{i-1}(OPT_{i-1}) - \bar{w}_{i-1}(S_i) - \bar{w}_{i-1}(S \cap S_i).\end{aligned}$$

Equation (2) follows now from the fact that $\bar{w}_i(SOL_i) \geq \bar{w}_{i-1}(SOL_i) - \bar{w}_{i-1}(S_i \setminus S)$ since only those edge of SOL_i decrease their weights which touch vertices from $S_i \setminus S$ (edges of OPT_{i-1} do not touch edges from S)and their total decrease is at most $\bar{w}_{i-1}(S_i \setminus S)$. ∎

2.2 Local Search

We consider a natural local search algorithm. A *t-move* is defined as follows: Take a center s_i^a of an approximate solution, remove its star and place the center at some vertex of V_1, say v_1. If v_1 is a center of the approximate solution, remove its star, and move the center that was placed at v_1 to some other vertex, say v_2. Repeat this process up to t times, until a center that was placed at $v_{t'}$ is placed at a vertex, $v_{t'+1}$ ($t' \leq t$), that either was not a center in the approximate solution, or whose center we ignore without replacing it. Note that we can make a circle, i.e. vertex $v_{t'+1}$ can be the vertex we started with. Thus we obtain a new set of centers, and their respective sizes of stars. To decide which leaves belong to which center, we solve a (max) transportation problem with supplies at the center vertices, each with supply equal to the number of leaves in its star, and all non center vertices are demand vertices with unit demand. The weight of the edge (v_i, j) between a center v_i and a leaf j, is modified to be the original weight $w(v_i, j)$ minus the weight of the single edge of the current approximate solution which touches vertex j.

A t-level local search algorithm called Algorithm t-search is defined as follows: Start with some feasible solution, and check whether its weight can be increased by a t-move. Repeat until no further improvements are possible. The proof of the next theorem will be given in the full version of the paper:

Theorem 2.2. *Algorithm* t-search *is a* $t/(1+2t)$*-approximation algorithm.*

An approximation factor arbitrarily close to 1/2 can be achieved by increasing t. The local search algorithm can be carried out in polynomial time while maintaining the bound up to any desired level of proximity by scaling the weights (see [2] for a detailed description).

3 MTPPSV

The problem can be reformulated as a MAXIMUM CAPACITATED STAR PACKING problem in bipartite graphs, by duplicating each vertex $i \in V_2$ b_i times, and then applying the algorithms from the previous section. However, this approach yields only a pseudo-polynomial time algorithms.

The local search algorithm can be modified to accommodate the demands of vertices in V_2, too. The step which needs to be modified is the one in which we calculate a new assignment of leaf vertices to centers, which was done using a transportation problem with modified weights. Here the weights must be modified in a more complex way. If we wish to "send" x_{ij} units from supply vertex $i \in V_1$ to demand vertex $j \in V_2$ we calculate the weight as x_{ij} original weights

$w(i,j)x_{ij}$ minus the cheapest edges touching j whose total shipment is x_{ij}. (In the earlier case, the $x_{ij} = 1$ so we subtracted only the cost of the cheapest edge touching vertex j.) This yields a maximum transportation problem with piecewise linear concave costs. See the textbook [1] which describes a polynomial time algorithm for equivalent the minimum cost piecewise convex problem.

The algorithm GR can be also implemented in polynomial time by the similar observation. Instead of duplicating demand vertices, on each step the algorithm directly finds the star of maximum weight. The star can have multiple edges between vertices, we define the flow x_{ij} from vertex i to j to be equal to the number of edges between vertices i and j. The profit from using x_{ij} edges is defined in the same way, it is a $w_{ij}x_{ij}$ minus a total cost of x_{ij} cheapest edges touching demand vertex j. Note that originally we assume that there are b_j edges of zero profit touching j. Therefore, on each step we need to solve a maximum transportation problem with piecewise linear concave costs and just one supply vertex. Actually, in this case the algorithm can be implemented directly without using [1].

4 Maximum Capacitated Star Packing

4.1 Local Search

Let S_i^a be the stars of an approximate solution, and S_i^o the stars of an optimal solution, such that the size of S_i^a is equal to the size of S_i^o. Denote by s_i^a (s_i^o) the center of star S_i^a (S_i^o).

A *move* is defined as follows: Take a center s_i^a of an approximate solution, remove its star and place the center at some vertex of the graph (possibly the same vertex), say v_1. If v_1 was a leaf of another star, we remove that edge. If v_1 is a center of the approximate solution, remove its star. Replace S_i^a by a star centered at v_1 of size c_i. The edges of the new star are selected greedily, using modified weights of edges (v_1, v_j) which are equal to the original weight minus the weight of edges selected by other stars of the approximate solution that touch v_j. Stars that had leaves removed from them by this process get the appropriate number of new leaves arbitrarily. An *improving move* is one in which the weight of the approximate solution improves. The algorithm starts with an arbitrary solution and performs improving moves as long as this is possible.

Theorem 4.1. Local-search *returns a* $\frac{1}{3}$*-approximation.*

Proof: Define $touch(S_i^o)$ to be the weight of all edges of an approximate solution touching vertices of S_i^o. At the end of the algorithm, no move is an improving move and thus moving the center of the approximate solution to its location in the optimal solution is non improving. Hence,

$$w(S_i^a) \geq w(S_i^o) - touch(S_i^o).$$

Let $apx = \sum_i w(S_i^a)$ be the value of the approximate solution while $opt = \sum_i w(S_i^o)$ is the optimal value. We now sum the above inequality over all centers

$i = 1, ..., p$. $\sum_i touch(S_i^o) \leq 2apx$, since each edge of the approximate solution touches at most two stars of the optimal solutions, as the stars are vertex disjoint. The summation therefore yields $apx \geq opt - 2apx$, or $apx \geq 1/3opt$. ∎

4.2 Metric Case

We now assume that the weights w_e satisfy the triangle inequality. A *greedy maximum matching* of size m is obtained by scanning the edges in non-increasing order of their weights and selecting edges as long as they are vertex-disjoint and their number does not exceed m.

Lemma 4.2. *Let M be a greedy maximum matching of size m. Let M' be an arbitrary matching of the same size. The weight of the i-th largest edge in M is greater than or equal to the weight of the $2i-1$ largest edge in M'.*

Assume that p is even. Our algorithm first computes a greedy maximum matching of size $p/2$. The vertices incident to the matching will be the centers of approximate star packing. The algorithm takes edges of the greedy matching one by one by decreasing of their values and assigns two unassigned star centers of stars of the biggest cardinality to the ends of the edge considered on that step. There are two ways to do it, and the algorithm chooses each with probability 1/2. After that the algorithm arbitrarily assigns leaves to centers. The algorithm for odd p works similarly. The algorithm can be derandomized by the standard method of conditional probabilities.

Theorem 4.3. *Algorithm Metric (Figure 2) is a $\frac{1}{2}$-approximation algorithm.*

Proof: Let apx be the weight of the solution returned by *Metric*. Let g_i be the length of the i-th largest edge in the greedy matching M computed by *Metric*. Let o_i be the length of the largest edge in the i-th star in a given optimal solution of total weight opt.

Suppose first that p is even. Then by the triangle inequality and the randomized way by which the ends of the greedy matching are assigned to centers of stars, the expected weight of each edge selected to the stars S_{2i-1} and S_{2i} is at least $g_i/2$. Therefore,

$$apx \geq \frac{1}{2} \sum_{i=1}^{p/2} (c_{2i-1} + c_{2i}) g_i.$$

On the other hand,

$$opt \leq \sum_{i=1}^{p/2} (c_{2i-1} o_{2i-1} + c_{2i} o_{2i}) \leq \sum_{i=1}^{p/2} (c_{2i-1} + c_{2i}) g_i,$$

where the second inequality follows from Lemma 4.2. The theorem follows from the above two relations.

Suppose now that p is odd. We repeat the same proof but also use the fact that (by the same lemma) $w(e_m) \geq o_p$. ∎

Metric
begin
Greedily compute a matching $M = (e_1, ..., e_m)$ *where* e_i *has ends* a_i *and* b_i, *and* $m = \lceil \frac{p}{2} \rceil$.
if p *is even:*
Arbitrarily choose from $V \setminus \{a_1, ..., a_m, b_1, ..., b_m\}$ *disjoint subsets* $V_1, ..., V_p$ *of sizes* $c_1, ..., c_p$ *respectively.*
for $i = 1, ..., \frac{p}{2}$
$(r_{2i-1}, r_{2i}) := (a_i, b_i)$ *or* $(r_{2i-1}, r_{2i}) := (b_i, a_i)$, *each with probability 0.5.*
elseif p *is odd:*
Arbitrarily choose from $V \setminus \{a_1, ..., a_m, b_1, ..., b_m\}$ *subsets* $V_1, ..., V_p$ *of sizes* $c_1, ..., c_{p-1}, c_p - 1$.
for $i = 1, ..., \frac{p-1}{2}$
$(r_{2i-1}, r_{2i}) := (a_i, b_i)$ *or* $(r_{2i-1}, r_{2i}) := (b_i, a_i)$, *each with probability 0.5.*
$r_p := a_m$ *or* $r_p := b_m$, *each with probability 0.5.*
$V_p := V_p \cup \{a_m, b_m\} \setminus r_p$.
end if
$S_i :=$ *the star with center* r_i *and leaves* V_i.
return $S_1, ..., S_p$.
end *Metric*

Fig. 2. Algorithm for the metric case

References

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

[2] E.M. Arkin and R. Hassin "On Local Search for Weighted k-set Packing", *Mathematics of Operations Research*, **23** (1998), 640-648.

[3] E.M. Arkin, R. Hassin, and M. Sviridenko "Approximating the Maximum Quadratic Assignment Problem" , *Information Processing Letters* **77** (2001), 13-16.

[4] M. Hujter, B. Klinz and G. Woeginger, A note on the complexity of the transportation problem with a permutable demand vector, Math. Methods Oper. Res. 50 (1999), no. 1, 9–16.

[5] S. Meusel, Minimizing the placement-time on printed circuit boards, Ph.D. Thesis, TU Bergakademie Freiberg, Fakultat fur Mathematik und Informatik, (1998).

[6] S. Meusel and R. Burkard, A transportation problem with a permuted demand vector, Math. Methods Oper. Res. **50** (1999), no. 1, 1–7.

[7] B. Steinbrecher, Ein Transportproblem mit permutiertem Bedarfsvektor, Master's thesis, TU Bergakademie Freiberg, Fakultat fur Mathematik und Informatik, (1997).

[8] L. A. Wolsey, "Maximizing real-valued submodular functions: primal and dual heuristics for location problems", *Mathematics of Operations Research* **7**, (1982) 410-425.

All-Norm Approximation Algorithms

Yossi Azar[1], Leah Epstein[2], Yossi Richter[3], and Gerhard J. Woeginger[4]

[1] School of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel. azar@tau.ac.il ‡
[2] School of Computer and Media Sciences, The Interdisciplinary Center, P.O.B 167, 46150 Herzliya, Israel. lea@idc.ac.il §
[3] School of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel. yo@tau.ac.il ¶
[4] Department of Mathematics, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands, and Institut für Mathematik, Technische Universität Graz, Steyrergasse 30, A-8010 Graz, Austria. g.j.woeginger@math.utwente.nl ‖

Abstract. A major drawback in optimization problems and in particular in scheduling problems is that for every measure there may be a different optimal solution. In many cases the various measures are different ℓ_p norms. We address this problem by introducing the concept of an *All-norm ρ-approximation algorithm*, which supplies one solution that guarantees ρ-approximation to all ℓ_p norms simultaneously. Specifically, we consider the problem of scheduling in the restricted assignment model, where there are m machines and n jobs, each is associated with a subset of the machines and should be assigned to one of them. Previous work considered approximation algorithms for each norm separately. Lenstra *et al.* [12] showed a 2-approximation algorithm for the problem with respect to the ℓ_∞ norm. For any fixed ℓ_p norm the previously known approximation algorithm has a performance of $\theta(p)$. We provide an all-norm 2-approximation polynomial algorithm for the restricted assignment problem. On the other hand, we show that for any given ℓ_p norm ($p > 1$) there is no PTAS unless P=NP by showing an APX-hardness result. We also show for any given ℓ_p norm a FPTAS for any fixed number of machines.

1 Introduction

1.1 Problem Definition

A major drawback in optimization problems and in particular in scheduling problems is that for every measure there may be a different optimal solution. Usually, different algorithms are used for diverse measures, each supplying its

‡ Research supported in part by the Israeli Ministry of industry and trade and by the Israel Science Foundation.
§ Research supported in part by the Israel Science Foundation (grant no. 250/01).
¶ Research supported in part by the Israeli Ministry of industry and trade.
‖ Supported by the START program Y43-MAT of the Austrian Ministry of Science.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 288–297, 2002.

own solution. Therefore, one may ask what is the "correct" solution for a given scheduling problem. In many cases there is no right answer to this question. We show that in some cases one can provide an appropriate answer, especially when the measures are different ℓ_p norms. Specifically, we address the optimization problem of scheduling in the restricted assignment model. We have m parallel machines and n independent jobs, where job j is associated with a weight w_j and a subset $M(j) \subseteq \{1, \ldots, m\}$ of the m parallel machines and should be assigned to one of them. For a given assignment, the load l_i on a machine i is the sum of weights of the jobs assigned to it. We denote by $\boldsymbol{l} = (l_1, \ldots, l_m)$ the machines load vector corresponding to an assignment, and further denote by $\boldsymbol{h}$ the vector $\boldsymbol{l}$ sorted in non-increasing order. We may use the ℓ_p norm ($p \geq 1$) to measure the quality of an assignment, namely the cost of an assignment is the ℓ_p norm of its corresponding load vector. The ℓ_p norm of a vector $\boldsymbol{l}$, denoted $\|\boldsymbol{l}\|_p$, is defined by: $\|\boldsymbol{l}\|_p = (\sum_{i=1}^{m} l_i^p)^{1/p}$.

Most research done so far in the various scheduling models considered the makespan (ℓ_∞) measure. In some applications other norms may be suitable such as the ℓ_2 norm. Consider for example a case where the weight of a job corresponds to its machine disk access frequency. Then each job may see a delay that is proportional to the load on the machine it is assigned to. Thus the *average* delay is proportional to the sum of squares of the machines loads (namely the ℓ_2 norm of the corresponding machine load vector) whereas the *maximum* delay is proportional to the maximum load.

Simple examples illustrate that for the general restricted assignment problem, an optimal solution for one norm is not necessarily optimal in another norm (and in fact may be very far from being optimal). Given that, one may ask what is the "correct" solution to a scheduling problem. When a solution optimal in all norms exists we would naturally define it as the correct solution and try to obtain it. For the special case of restricted assignment with unit jobs only, Alon *et al.* [1] showed that a *strongly-optimal* assignment that is optimal in all norms exists, and can be found in polynomial time. However, this is not the case in general.

1.2 Our Results

All-norm approximation: In light of the above discussion, we introduce the concept of an *All-norm ρ-approximation algorithm*, which supplies one solution guaranteeing simultaneously ρ-approximation with respect to the optimal solutions for all norms. Note that an approximated solution with respect to one norm may not guarantee any constant approximation ratio for any other norm. This does not contradict the fact that there may be a different solution approximating the two norms simultaneously. Simple examples illustrate that we can not hope for an all-norm $(1+\epsilon)$-approximation for arbitrary ϵ for this problem (the example in [1] illustrates that ϵ must be larger than 0.003 even for two norms), hence the best we can hope for (independent of the computational power) is an all-norm ρ-approximation, when ρ is constant. Moreover, from the computational point of view, we can not expect to achieve an all-norm approximation polynomial algorithm with ratio better than 3/2 since Lenstra *et al.* [12] proved a 3/2 lower bound on the approximation ratio of any polynomial algorithm for

the makespan alone (assuming $P \neq NP$). Lenstra *et al.* [12] and Shmoys and Tardos [16] presented a 2-approximation algorithm for the makespan, however their algorithm does not guarantee any constant approximation ratio to optimal solutions for any other norms (it is easy to come up with a concrete example to support that). Our main result is an all-norm 2-approximation polynomial algorithm for the restricted assignment model. Our algorithm returns a feasible solution which is at most 2 times the optimal solution for all ℓ_p norms ($p \geq 1$) simultaneously. In contrast, note that for the related machines model and hence for the more general model of unrelated machines, in general there is no assignment obtaining constant approximation ratio for all norms simultaneously (this can be shown by a simple example even when considering only the ℓ_1 and ℓ_∞ norms).

Kleinberg *et al.* [11] and Goel *et al.* [6] considered the problem of fairest bandwidth allocation, where the goal is to maximize the bandwidth allocated to users, in contrast to minimizing the machines loads. In [6] α-balanced assignments are defined, which are similar to our concept of all-norm approximation. However, the algorithm suggested there works only for unit jobs and is $O(\log m)$-competitive. In contrast, our algorithm works for arbitrary size jobs and guarantees constant approximation. We note that the idea of approximating more than one measure appears in [17,2] where bicriteria approximation for the makespan and the average completion time is provided.

Approximation for any given norm: Recall that for the ℓ_∞ case Lenstra *et al.* [12] presented a 2-approximation algorithm (presented for the more general model of unrelated machines, where each job has an associated m-vector specifying its weight on each machine). For any given ℓ_p norm the only previous approximation algorithm for restricted assignment, presented by Awerbuch *et al.* [3], has a performance of $\theta(p)$ (this algorithm was presented as an on-line algorithm for the unrelated machines model). Note that not only our all-norm 2-approximation algorithm provides 2-approximation to all norms simultaneously, it also improves the previous best approximation algorithm for each fixed ℓ_p norm separately.

Non-approximability for any given norm: Clearly, one may hope to get for any given ℓ_p norm a better approximation ratio (smaller than 2), or even a Polynomial Time Approximation Scheme (PTAS). However, we show that for any given ℓ_p norm ($p > 1$) the problem of scheduling in the restricted assignment model is APX-hard, thus there is no PTAS for the problem unless $P = NP$. Note that for $p = 1$ any assignment is optimal.

Approximation scheme: For any given ℓ_p norm it is impossible to get a PTAS for an arbitrary number of machines. Therefore, the only possible approximation scheme for a given norm is for a fixed number of machines. We present for any given norm a Fully Polynomial Time Approximation Scheme (FPTAS) for any fixed number of machines. Note that for minimizing the makespan Horowitz and Sahni [9] presented a FPTAS for any fixed number of machines. Lenstra *et al.* [12] suggested a PTAS for the same problem (i.e. minimizing the makespan) with better space complexity.

1.3 Techniques and Related Results

Other related results: Other scheduling models have also been studied. For the identical machines model, where each job has an associated weight and can be assigned to any machine, Hochbaum and Shmoys [8] presented a PTAS for the case of minimizing the makespan. Later, Alon *et al.* [1] showed a PTAS for any ℓ_p norm in the identical machines model. For the related machines model, in which each machine has a speed and the machine load equals the sum of jobs weights assigned to it divided by its speed, Hochbaum and Shmoys [7] presented a PTAS for the case of minimizing the makespan. Epstein and Sgall [5] showed a PTAS for any ℓ_p norm in the same model.

Note that, previous work discussed above showed that PTAS can be achieved for the identical and related machines models when considering the makespan for cost. In contrast, only constant approximation is possible for the restricted assignment and unrelated machines models (see [12]). Our work establishes the same phenomenon for the ℓ_p norm, by proving that only constant approximation can exist for restricted assignment.

Techniques: Our main result, the all-norm 2-approximation algorithm, consists of two phases - finding a strongly-optimal fractional assignment and rounding in to an integral assignment which guarantees 2-approximation to the optimal assignments in all norm simultaneously. The first phase depends on constructing linear programs with exponential number of constraints solved using the ellipsoid algorithm with a supplied oracle. Our algorithm works for the more general model of unrelated machines and finds the lexicographically best (smallest) assignment. Hence, in this sense, it generalizes the algorithm suggested by Megiddo [13,14], which can be used for the restricted assignment model only. Although the second phase can employ the rounding scheme of [16], our rounding technique, based on eliminating cycles in a bipartite graph, is considerably simpler and more suitable for our needs. Our hardness of approximation result is reduced (by a L-Reduction) from a result by Petrank [15] concerning a variant of 3-Dimensional matching.

Paper structure: In Section 2 we present our approximation algorithm. In section 3 we prove the hardness of approximation result for the problem, and in section 4 we show for any given ℓ_p norm a FPTAS for any fixed number of machines. The last two sections are omitted due to lack of space, and can be found in the full version (see [4]).

2 All-Norm Approximation Algorithm

We use the notion of a *strongly-optimal assignment* defined in [1] throughout this paper. We repeat the definition in short :

Definition 1. *Given an assignment H denote by S_k the total load on the k most loaded machines. We say that an assignment is* strongly-optimal *if for any other assignment H' and for all $1 \leq k \leq m$ we have $S_k \leq S'_k$.*

A strongly-optimal assignment is optimal in any norm. In the case of unit jobs a strongly-optimal integral assignment exists (and can be found in polynomial

time), however this is not the case in general (see [1]). It turns out there always exists a strongly-optimal *fractional* assignment in the general case. Our algorithm works in two stages: in the first stage we find a strongly-optimal fractional assignment and in the second stage we round this fractional assignment to an integral assignment which guarantees 2-approximation with respect to the optimal solutions for all ℓ_p norms.

2.1 Finding a Strongly-Optimal Fractional Assignment

Lemma 1. *For every instance in the restricted assignment model there exists a fractional assignment that is strongly-optimal. In particular, every fractional assignment which induces the lexicographically smallest load vector is a strongly-optimal fractional assignment.*

Proof. We restrict ourselves only to rational weights. The lexicographically smallest load vector induced by a fractional assignment (when considering the machines load vector sorted in non-increasing order) is uniquely defined and consists of rational weights (since it is a solution of a set of rational linear equations). Denote such an assignment by H. Assume by contradiction that H is not strongly-optimal, thus there exist a fractional assignment H' and an integer k, $1 \leq k \leq m$, such that $S_k > S'_k$ (we may assume that H' also consists of rational weights by means of limit). We may scale all the weights such that each assigned fraction in H and H' is integral. We may then translate the scaled instance to a new instance with unit jobs only, by viewing a job with associated weight w_j as w_j unit jobs. Clearly, the lexicographically smallest assignment for the new instance is the scaled H and it is also the strongly-optimal assignment (see [1]). However, the scaled H' contradicts this fact.

Note that although [1] provides an algorithm to find the strongly-optimal assignment in the unit jobs case which is polynomial in the number of jobs, we can not use it since it is not clear how to choose the units appropriately. Even if such units could be found, translating our original jobs to unit jobs would not necessarily result in a polynomial number of jobs and therefore the algorithm would not be polynomial.

The first stage of our algorithm consists of finding this strongly-optimal assignment. We present a more general algorithm. Our algorithm works for the more general model of unrelated machines and finds the lexicographically smallest fractional assignment (when considering the machines load vector $\boldsymbol{h}$ sorted in non-increasing order). In particular, according to lemma 1, for the restricted assignment model the lexicographically smallest fractional assignment is the strongly-optimal fractional assignment. In this sense, our algorithm generalizes the algorithm suggested by Megiddo [13,14], which can be used for the restricted assignment model only.

Theorem 1. *In the unrelated machines model, the lexicographically smallest fractional assignment can be found in polynomial time.*

Proof. We define the following decision problem in the unrelated machines model: given n jobs, where job j is associated with a weight vector $\boldsymbol{w}_j$, and $k \leq m$ limits: $S_1 \leq S_2 \leq \ldots \leq S_k$ is there an assignment H such that $\sum_{i=1}^{r} l_i \leq S_r$ $(r = 1, \ldots, k)$ where $\boldsymbol{l}$ is the vector of machine loads introduced by H sorted in non-increasing order. We note that the lexicographically smallest prefix vector $\boldsymbol{S} = (S_1, \ldots, S_m)$ induces the lexicographically smallest assignment $\boldsymbol{h}$ by defining $h_i = S_i - S_{i-1}$ $(S_0 = 0)$. Denote by $M(j)$ $(j = 1, \ldots, n)$ the set of machines to which job j can be assigned, i.e $\forall i \in M(j)\ w_{ij} < \infty$. For the case of $k = 1$ (i.e. deciding the makespan) the decision problem can be translated to the following linear program:

$$\begin{aligned}
&\sum_{i=1}^{m} x_{ij} = 1 && \text{for } j = 1, \ldots, n \\
&\sum_{j=1}^{n} x_{ij} w_{ij} \leq S_1 && \text{for } i = 1, \ldots, m \\
&x_{ij} \geq 0 && \text{for } j = 1, \ldots, n\ ,\ i = 1, \ldots, m \\
&x_{ij} = 0 && \text{for } j = 1, \ldots, n\ ,\ i \notin M(j)\ ,
\end{aligned}$$

where x_{ij} denotes the relative fraction of job j placed on machine i. Since we can not identify the machines according to their loads order, the general case is represented by a linear program with number of constraints exponential in m, as follows:

$$\begin{aligned}
&\sum_{i=1}^{m} x_{ij} = 1 && \text{for} && j = 1, \ldots, n \\
&\sum_{j=1}^{n} x_{i_1 j} w_{i_1 j} + \ldots + \sum_{j=1}^{n} x_{i_t j} w_{i_t j} \leq S_t\ \forall 1 \leq t \leq k\ \forall 1 \leq i_1 < \ldots < i_t \leq m \\
&x_{ij} \geq 0 && \text{for} && j = 1, \ldots, n\ ,\ i = 1, \ldots, m \\
&x_{ij} = 0 && \text{for} && j = 1, \ldots, n\ ,\ i \notin M(j)\ .
\end{aligned}$$

We employ the ellipsoid algorithm to solve this linear program in polynomial time (see [10] for details). In order to use the ellipsoid algorithm we should supply a separation oracle running in polynomial time. We next describe the algorithm we use as the oracle for the general linear program:

1. Given the assignment we construct the corresponding machines load vector.
2. We sort the load vector. Denote by $\boldsymbol{h}$ the sorted vector.
3. If there exists r, $1 \leq r \leq k$ such that $\sum_{i=1}^{r} h_i > S_r$ then the algorithm returns 'not feasible' together with the unsatisfied constraint - the one involving the r most loaded machines (whose indices we have).
4. Otherwise the algorithm returns 'feasible'.

Since the sorting operation (step 2) dominates the time complexity of the algorithm, its running time is clearly polynomial. We state the following claim without proof.

Claim. The algorithm returns 'feasible' $\Leftrightarrow$ the given assignment is feasible.

We use an incremental process to find the lexicographically smallest assignment. Our algorithm has m steps where in step i we determine the total load on the i most loaded machines in the assignment, given the total loads on the k most loaded machines ($k = 1, \ldots, i-1$). Each step is done by performing a binary search on the decision problems. Consider the first step for example: we want to establish the load on the most loaded machine. Denote for job j ($j = 1, \ldots, n$) its smallest possible weight by $w_j^{min} = \min_i w_{ij}$. Let $t = \sum_{j=1}^{n} w_j^{min}$. Clearly t is an upper bound on the load of the most loaded machine, and t/m a lower bound. We can perform a binary search on the load of the most loaded machine while starting with $u = t$ (initial upper bound) and $l = t/m$ (initial lower bound). Testing a bound S on the most loaded machine is done by considering the decision problem with the n jobs and limit $S_1 = S$. We can stop the binary search when $u - l < \epsilon$ and set the load on the most loaded machine to the load obtained from the feasible solution to the linear program. Later we show how to choose ϵ such that the value produced by the feasible solution is the exact one since there is at most one possible load value in the range $[l, u]$. Given this ϵ, the number of iterations needed for the binary search to complete is $O(\log(t/\epsilon))$. In the ith step ($i = 1, \ldots, m$) we perform the binary search on the total load of the i most loaded machines given the total loads on the k most loaded machines ($k = 1, \ldots, i-1$). Denote by $L_1, \ldots, L_{i-1}$ the prefix loads we found. We perform the binary search on the total load of the i most loaded machines starting with $u = L_{i-1} + t,\ l = L_{i-1}$. Testing a bound S is done by considering the decision problem with the n jobs and limits $S_1 = L_1, \ldots, S_{i-1} = L_{i-1}, S_i = S$. Again we stop the binary search when $u - l < \epsilon$ and set L_i to the total load on the i most loaded machines produced by the feasible assignment we found for the linear program.

We now determine the value of ϵ. Each feasible solution to the linear problem $\{x_{ij}\}$ can be written as $\left\{\frac{d_{ij}}{d}\right\}$ where d and $\{d_{ij}\}$ are integers smaller than $2^{P(I)}$ for some polynomial P in the size of the input (see [10] for example). If we choose $\epsilon = 2^{-2P(I)}$ then we are guaranteed that there is only one possible load value in the range $[l, u]$ when $u - l < \epsilon$ (see [10]). Thus in each step $i = 1, \ldots, m$ the binary search involves $O(P(I) + log \sum_{j=1}^{n} w_j^{min})$ iterations, polynomial in the size of the input. Hence in polynomial time we find the desired lexicographically smallest assignment.

2.2 Rounding the Strongly-Optimal Fractional Assignment

We now return to the restricted assignment model. As mentioned above, the algorithm presented in theorem 1 finds the strongly-optimal fractional assignment in polynomial time. The second stage of our algorithm consists of rounding the fractional assignment $\{x_{ij}\}$ to an integral assignment for the problem obtaining 2-approximation for every ℓ_p norm measure. We note that although the rounding scheme presented in [16] can be used for this purpose, our rounding technique is considerably simpler and more suitable for our needs.

Theorem 2. *A strongly-optimal fractional assignment can be rounded in polynomial time to an integral assignment which is at most 2 times the optimal solution for all ℓ_p norms at the same time.*

Proof. Given the fractional assignment $\{x_{ij}\}$ we will show how to construct the desired integral assignment $\{\hat{x}_{ij}\}$ in polynomial time. We construct the bipartite graph $G = (U, V, E)$ having $|U| = n$ vertices on one side (representing the jobs) and $|V| = m$ vertices on the other (representing the machines) while $E = \{(i,j)|x_{ij} > 0\}$. At first we would like to eliminate all cycles in G while preserving the same load on all machines. We eliminate the cycles in G in polynomial time by performing the following steps:

1. We define a weight function $W : E \to R^+$ on the edges of G such that $W(i,j) = x_{ij}w_j$, i.e. the actual load of job j that is assigned to machine i.
2. As long as there are cycles in G, find a cycle, and determine the edge with the smallest weight on the cycle (denote this edge by e and its weight by t).
3. Starting from e subtract t and add t from the weights on alternating edges on the cycle, and remove from G the edges with weight 0.

It is clear that this method eliminates the cycles one by one (by discarding the edge with the smallest weight on each cycle) while preserving the original load on all machines. Denote by G the new graph obtained after eliminating the cycles and by $\{x_{ij}\}$ the new strongly-optimal fractional assignment represented by G (which is a forest). In the first rounding phase consider each integral assignment $x_{ij} = 1$, set $\hat{x}_{ij} = 1$ and discard the corresponding edge from the graph. Denote again by G the resulting graph.

In the second rounding phase we assign all the remaining fractional jobs. For this end we construct a matching in G that covers all job nodes by using the same method presented in [12]. We consider each connected component in G, which is a tree, and root that tree in one of the job nodes. Match each job node with any one of its children. Since every node in the tree has at most one father we get a matching and since each job node is not a leaf (each job node has a degree at least 2) the resulting matching covers all job nodes. For each edge (i,j) in the matching set $\hat{x}_{ij} = 1$.

We now prove that the schedule obtained from the assignment $\{\hat{x}_{ij}\}$ guarantees a 2-approximation to the optimal solutions for all ℓ_p norms (for $p \geq 1$). Fix p and denote by OPT the optimal solution for the problem using ℓ_p for cost. Denote by H^{opt} the strongly-optimal fractional schedule obtained after eliminating the cycles and denote by H the schedule returned by the algorithm. Further denote by H_1 the schedule consisting of the jobs assigned in the first rounding phase (right after eliminating the cycles) and by H_2 the schedule consisting of the jobs assigned in the second rounding phase (those assigned by the matching process). We have :

$$\|H_1\|_p \leq \|H^{opt}\|_p \leq \|OPT\|_p \ ,$$

where the first inequality follows from the fact that H_1 is a sub-schedule of H^{opt} and the second inequality results from H^{opt} being a strongly-optimal fractional

schedule thus optimal in any ℓ_p norm compared with any other fractional schedule, and certainly optimal compared with OPT which is an integral schedule. We also know that:

$$\|H_2\|_p \leq \|OPT\|_p \ ,$$

where the inequality results from the fact that H_2 schedules only *one job per machine* thus optimal integral assignment in any ℓ_p norm for the subset of jobs it assigns and certainly has cost smaller than any integral assignment for the whole set of jobs. We can now show :

$$\|H\|_p = \|H_1 + H_2\|_p \leq \|H_1\|_p + \|H_2\|_p \leq \|OPT\|_p + \|OPT\|_p = 2\|OPT\|_p \ ,$$

which concludes the proof that the schedule H we constructed guarantees a 2-approximation to optimal solutions for all ℓ_p norms and can be found in polynomial time.

References

1. N. Alon, Y. Azar, G. Woeginger, and T. Yadid. Approximation schemes for scheduling. In *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms*, pages 493–500, 1997.
2. J. Aslam, A. Rasala, C. Stein, and N. Young. Improved bicriteria existence theorems for scheduling. In *Proc. 10th ACM-SIAM Symp. on Discrete Algorithms*, pages 846–847, 1999.
3. B. Awerbuch, Y. Azar, E. Grove, M. Kao, P. Krishnan, and J. Vitter. Load balancing in the l_p norm. In *Proc. 36th IEEE Symp. on Found. of Comp. Science*, pages 383–391, 1995.
4. Y. Azar, L. Epstein, Y. Richter, and G. J. Woeginger. All-norm approximation algorithms (full version). *http://www.cs.tau.ac.il/∼yo*.
5. L. Epstein and J. Sgall. Approximation schemes for scheduling on uniformly related and identical parallel machines. In *Proc. 7th Annual European Symposium on Algorithms*, pages 151–162, 1999.
6. A. Goel, A. Meyerson, and S. Plotkin. Approximate majorization and fair online load balancing. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms*, pages 384–390, 2001.
7. D. Hochbaum and D. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, 1988.
8. D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *J. of the ACM*, 34(1):144–162, January 1987.
9. E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery*, 23:317–327, 1976.
10. H. Karloff. *Linear Programming*. Birkhäuser, Boston, 1991.
11. J. Kleinberg, Y. Rabani, and E. Tardos. Fairness in routing and load balancing. *J. Comput. System Sci.*, 63(1):2–20, 2001.
12. J.K. Lenstra, D.B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Prog.*, 46:259–271, 1990.
13. N. Megiddo. Optinal flows in networks with multiple sources and sinks. *Mathematical Programming*, 7:97–107, 1974.

14. N. Megiddo. A good algorithm for lexicographically optimal flows in multi-terminal networks. *Bulletin of the American Mathematical Society*, 83(3):407–409, 1977.
15. E. Petrank. The hardness of approximation: Gap location. In *Computational Complexity 4*, pages 133–157, 1994.
16. D. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming A*, 62:461–474, 1993. Also in the proceeding of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993.
17. C. Stein and J. Wein. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *Operations Research Letters*, 21, 1997.

Approximability of Dense Instances of Nearest Codeword Problem

Cristina Bazgan[1], W. Fernandez de la Vega[2], and Marek Karpinski[3]

[1] Université Paris Dauphine, LAMSADE, 75016 Paris, France,
bazgan@lamsade.dauphine.fr
[2] CNRS, URA 410, Université Paris Sud, LRI, 91405 Orsay, France, lalo@lri.fr
[3] Dept. of Computer Science, University of Bonn, 53117 Bonn, Germany
marek@cs.uni-bonn.de

Abstract. We give a polynomial time approximation scheme (PTAS) for dense instances of the Nearest Codeword problem.

1 Introduction

We follow [15] in defining the Nearest Codeword problem as the minimum constraint satisfaction problem for linear equations mod 2 with exactly 3 variables per equation. It is shown in [15] that the restriction imposed on the number of variables per equation (fixing it to be exactly 3) does not reduce approximation hardness of the problem. The problem is, for a given set of linear equations mod 2 to construct an assignment which minimizes the number of unsatisfied equations. We shall use in this paper clearly an equivalent formulation of the problem of minimizing the number of satisfied equations. Adopting the notation of [11] we denote it also as the Min-E3-Lin2 problem. Min-Ek-Lin2 will stand for the k-ary version of the Nearest Codeword problem.

The Nearest Codeword problem arises in a number of coding theoretic, and algorithmic contexts, see, e.g., [1], [15], [8], [7]. It is known to be exceedingly hard to approximate; it is known to be NP-hard to approximate to within a factor $n^{\Omega(1)/\log\log n}$. Only recently the first sublinear approximation ratio algorithm has beeing designed for that problem [5]. In this paper we prove that, somewhat surprisingly, the Nearest Codeword problem on dense instances does have a PTAS. We call an instance of Nearest Codeword problem (Min-E2-Lin2) problem *dense*, if the number of occurrences of each variable in the equations is $\Theta(n^2)$ for n the number of variables. We call an instance of Nearest Codeword (Min-E2-Lin2) *dense in average* if the number of equations is $\Theta(n^3)$. Analogously, we define *density*, and *average density*, for Min-Ek-Lin2 problems.

It is easy to be seen that the results of [2] and [9] on existence of PTASs for average dense maximum constraint satisfaction problems cannot be applied to their average dense minimum analogs (for a survey paper on approximability of some other dense optimization problems see also [13]). This observation

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 298–307, 2002.

can be also strenghten for the dense instances of minimum constraint satisfaction by noting that dense instances of Vertex Cover can be expressed as dense instances of minimum constraint satisfaction problem for 2DNF clauses, i.e. conjunctions of 2 literals, and then applying the result of [6], [14] to the effect that there are no PTAS for the dense Vertex Cover. In [10] it was also proven that the *dense* and *average dense* instances of Min Tsp(1,2) and Longest Path problems do not have polynomial time approximation schemes.

In [2] there were however two dense minimization problems identified as having PTASs, namely dense Bisection, and Min-k Cut. This has lead us to investigate the approximation complexity of dense Nearest Codeword problem. Also recently, PTASs have been designed for dense Min Equivalence and dense Min-kSat problems (cf. [3], [4]). The main result of this paper is a proof of an existence of a PTAS for the dense Nearest Codeword problem.

The approximation schemes developed in this paper for the dense Nearest Codeword problem use some novel density sampler techniques for graphs, and k-uniform hypergraphs, and extend available up to now approximation techniques for attacking dense instances of minimum constraint satisfaction problems.

The Nearest Codeword problem in its bounded arity (=3) form was proven to be approximation hard for its unbounded arity version in [15] (Lemma 37). This results in $n^{\Omega(1)/\log\log n}$ approximation lower bound for the Nearest Codeword problem by [8], [7], where n is the number of variables. It is also easy to show that Nearest Codeword is hard to approximate to within a factor $n^{\Omega(1)/\log\log n}$ on average dense instances.

The paper is organized as follows. In Section 2 we give the necessary definitions and prove the NP-hardness of dense instances of Min-E3-Lin2 in exact setting, and in Section 3 we give a polynomial time approximation scheme for the dense instances of Min-E3-Lin2.

2 Preliminaries

We begin with basic definitions.

Approximability. A minimization problem has a *polynomial time approximation scheme* (a PTAS, in short) if there exists a polynomial time approximation algorithm that gives for each instance x of the problem a solution y of value $m(x, y)$ such that $m(x, y) \leq (1 + \varepsilon)opt(x)$ for every constant $\varepsilon > 0$ where $opt(x)$ is the value of an optimum solution.

Nearest Codeword Problem (Min-E3-Lin2)

Input: A set of m equations in boolean variables $x_1, \ldots, x_n$ where each equation has the form $x_{i_1} \oplus x_{i_2} \oplus x_{i_3} = 0$ or $x_{i_1} \oplus x_{i_2} \oplus x_{i_3} = 1$.
Output: An assignment to the variables that minimizes the number of equations satisfied.

Density. A set of instances of MIN-E3-LIN2 is δ-dense if for each variable x, the total number of occurrences of x is at least δn^2 in each instance. A class of instances of MIN-E3-LIN2 is *dense*, if there is a constant δ such that the class is δ-dense.

Let us show now that dense MIN-E3-LIN2 is NP-hard in exact setting. The reduction is from MIN-E3-LIN2, which is approximation hard for a ratio $n^{\Omega(1)/\log\log n}$ [8], [7], where n is the number of variables. Given an instance I of MIN-E3-LIN2 on a set of n variables $X = \{x_1, \ldots, x_n\}$ with m equations $x_{t_1} \oplus x_{t_2} \oplus x_{t_3} = b$, where $b \in \{0, 1\}$, we construct an instance I' of dense MIN-E3-LIN2 as follows: we extend the set of variables X by two disjoint sets $Y = \{y_1, \ldots, y_n\}$ and $Z = \{z_1, \ldots, z_n\}$. I' contains aside from the equations of I, the equations of the form $x_i \oplus y_j \oplus z_h = 0$ and $x_i \oplus y_j \oplus z_h = 1$ for all $1 \leq i, j, h \leq n$. Note that the system I' is dense. We note also that exactly n^3 of the added equations are satisfied independently of the values of the variables in X, Y and Z. Thus $opt(I') = opt(I) + n^3$, proving the claimed reduction.

3 Dense MIN-E3-LIN2 Has a PTAS

Let the system $\mathcal{S} = \{E_1, ..., E_m\}$ be a δ-dense instance of MIN-E3-LIN2, on a set X of n variables $\{x_1, \ldots, x_n\}$.

We will run two distinct algorithms, algorithm A and algorithm B, and select the solution with the minimum value. Algorithm A gives a good approximate solution for the instances whose minimum value is at least αn^3. Algorithm B gives a good approximate solution for the instances whose minimum value is less than αn^3, where α is a constant depending both on δ and the required accuracy ϵ.

3.1 Algorithm A

Algorithm A depends on formulating the problem as a Smooth Integer Program [2] as follows.

A smooth degree-3 polynomial (with smoothness e) has the form

$$\sum a_{ijh} x_i x_j x_h + \sum b_{ij} x_i x_j + \sum c_i x_i + d$$

where each $|a_{ijh}| \leq e$, $|b_{ij}| \leq en$, $|c_i| \leq en^2$, $|d| \leq en^3$ (cf. [2]).

For each equation $x_i \oplus y_i \oplus z_i = b_i$ in $\mathcal{S}$, we construct the smooth polynomial

$$P_i \equiv (1 - x_i)(1 - y_i)(1 - z_i) + x_i y_i (1 - z_i) + y_i z_i (1 - x_i) + z_i x_i (1 - y_i)$$

if $b_i = 0$, and

$$P_i \equiv x_i (1 - y_i)(1 - z_i) + y_i (1 - x_i)(1 - z_i) + z_i (1 - x_i)(1 - y_i) + x_i y_i z_i$$

if $b_i = 1$. We have then the Smooth Integer Program IP:

$$\begin{cases} \min \sum_{j=1}^{m} P_i \\ \text{s. t. } x_i, y_i, z_i \in \{0, 1\} \ \forall i, \ 1 \leq i \leq n. \end{cases}$$

A result of [2] can be used now to approximate in polynomial time the minimum value of IP with additive error ϵn^3 for every $\epsilon > 0$. This provides an approximation ratio $1 + \epsilon$ whenever the optimum value is $\Omega(n^3)$.

3.2 Algorithm B

The algorithm B is guaranteed to give, as we will show, approximation ratio $1+\epsilon$ for each fixed ϵ, whenever the optimum is at most αn^3 for a fixed α, depending on ϵ and on the density.

Algorithm B

Input: A dense system $\mathcal{S}$ of linear equations in GF[2] over a set X of n variables with exactly 3 variables per equation.

1. Pick two disjoint random samples $S_1, S_2 \subseteq X$ of size $m = \Theta\left(\frac{\log n}{\epsilon^2}\right)$;

2. For each possible assignment $a \in \{0,1\}^{|S_1 \cup S_2|}$ for the variables y in $S_1 \cup S_2$ (y^a will stand for the boolean value of y for the assignement a) do the following:

2.1 For each variable $x \notin S_1 \cup S_2$ do the following:

Let $H^a_{x,0}$ and $H^a_{x,1}$ be the bipartite graphs with common vertex set $V(H^a_{x,0}) = V(H^a_{x,1}) = S_1 \cup S_2$ and edge sets

$$E(H^a_{x,0}) = \{\{y,z\} : \chi_{S_1}(y) \oplus \chi_{S_1}(z) = 1 \wedge x \oplus y \oplus z = b \in \mathcal{S} \wedge y^a \oplus z^a = b\}$$

and

$$E(H^a_{x,1}) = \{\{y,z\} : \chi_{S_1}(y) \oplus \chi_{S_1}(z) = 1 \wedge x \oplus y \oplus z = b \in \mathcal{S} \wedge 1 \oplus y^a \oplus z^a = b\}$$

Let $m^a_0 = |E(H^a_{x,0})|$, $m^a_1 = |E(H^a_{x,1})|$.
If $m^a_0 \geq \frac{2}{3}(m^a_0 + m^a_1)$, then set x to 1.
If $m^a_1 \geq \frac{2}{3}(m^a_0 + m^a_1)$, then set x to 0.

Otherwise, set x to be *undefined.*

2.2 In this stage, we assign values to the variables which are undefined after the completion of stage 2.1. Let D^a be the set of variables assigned in stage 2.1, and let $U^a = S_1 \cup S_2 \cup D^a$. $V^a = X \setminus U^a$ denotes the set of undefined variables. For each undefined variable y, let S_y denote the set of equations which contain y and two variables in U^a. Let k^a_0 (resp. k^a_1) denote the number of equations in S_y satisfied by a and by setting y to 0 (resp. to 1).

If $k^a_0 \leq k^a_1$, then set y to 0. Else, set y to 1.

Let X^a denote the overall assignment produced by the end of this stage.

3. Among all the assignments X^a pick one which satisfies the minimum number of equations of $\mathcal{S}$.

Output that assignment.

4 Proof of the Correctness of Algorithm B When the Value of the Instance Is "Small"

We will use the following graph density sampling lemma. Recall that the density d of a graph $G = (V, E)$ is defined by

$$d = \frac{|E|}{\binom{|V|}{2}}.$$

Lemma 1. *Let d and ϵ be fixed and let the graph $G = (V, E)$ have $|V| = n$ vertices and density d. Let $m = \Theta(1/d\ \epsilon^{-2} \log n)$. Let $X = \{x_1, ..., x_m\}$ and $Y = \{y_1, ..., y_m\}$ be two random disjoint subsets of $V(G)$ with $|X| = |Y| = m$ and let $e(X, Y)$ be the number of edges of G between X to Y. Then, for each sufficiently large n, we have*

$$\Pr[|e(X, Y) - m^2 d| \leq \epsilon m^2 d] = 1 - o(1/n).$$

Proof. We will use the following inequality due to Hoeffding [12]. Let $X_1, ..., X_m$ be independent and identically distributed. Let $\mu = E(X_1)$ and assume that X_1 satisfies $0 \leq X_1 \leq \Delta$. Let $S_m = \sum_{i=1}^{m} X_i$. Then:

$$\Pr(|S_m - \mu m| \geq \epsilon \Delta m) \leq 2 \exp(-2\epsilon^2 m). \tag{1}$$

Clearly

$$E(e(X, Y)) = m^2 d.$$

For each $z \in V \backslash X$, write

$$T_z = |\Gamma(z) \cap X|.$$

Let $T = \sum_{z \in V \backslash X} T_z$. Then, $T = T' + \Delta$ where $\Delta \leq m(m-1)/2$, and T' is the sum of m randomly chosen valencies from the set of valencies of G. Thus using (1),

$$\Pr[|T' - mnd| \leq \epsilon mn + m(m-1)^2/2] \geq 1 - 2 \exp(-O(\epsilon^2 m)).$$

Clearly,

$$\begin{aligned} e(X, Y) &= \sum_{z \in Y} T_z \\ &= \sum_{1 \leq i \leq m} \delta_i \end{aligned}$$

say. Assume now, with negligible error, that the vertices of Y are produced by independent trials. Then, the δ_i are independent random variables with the same distribution as δ_1, defined by

$$\Pr[\delta_1 = k] = \frac{1}{n-m} |\{z \in V(G) | T_z = k\}|,\ 0 \leq k \leq m.$$

Conditionally on θ where $\theta \in mnd(1 \pm \epsilon)$ and $E(\delta_1) = \theta$, and using again (1),

$$\Pr[|e(X,Y) - \frac{m\theta}{n}| \leq \epsilon m^2] \geq 1 - 2\exp(-2\epsilon^2 m)$$

or

$$\Pr[|e(X,Y) - \frac{m\theta}{n}| \leq \epsilon m^2 d] \geq 1 - 2\exp(-2\epsilon^2 d^2 m).$$

The conditioning event has probability at least $1 - 2\exp(-2\epsilon^2 m^2 d)$. We have thus, without any conditioning,

$$\begin{aligned}\Pr[|e(X,Y) - \frac{m\theta}{n}| \leq \epsilon m^2 d] &\geq 1 - 2\exp(-2\epsilon^2 d^2 m) - 2\exp(-2\epsilon^2 m^2 d) \\ &\geq 1 - 3\exp(-2\epsilon^2 d^2 m).\end{aligned}$$

This completes the proof.

We now return to our proof of correctness. We assume, as we can, that a is the restriction to $S_1 \cup S_2$ of an optimal assignment a^*. For each $y \in X$, we let y^{a^*} denote the value of y in a^*. Let $x \in X \setminus (S_1 \cup S_2)$.

Let $G_{x,0}$ and $G_{x,1}$ be the graphs with common vertex set $V(G_{x,0}) = V(G_{x,1}) = X$ and edge sets

$$E(G_{x,0}) = \{\{y,z\} : x \oplus y \oplus z = b \in \mathcal{S} \wedge y^{a^*} \oplus z^{a^*} = b\}$$

and

$$E(G_{x,1}) = \{\{y,z\} : x \oplus y \oplus z = b \in \mathcal{S} \wedge 1 \oplus y^{a^*} \oplus z^{a^*} = b\}$$

Let $n_0^{a^*} = |E(G_{x,0})|, n_1^{a^*} = |E(G_{x,1})|$, $n^{a^*} = n_0^{a^*} + n_1^{a^*}$. Also, let $m^a = m_0^a + m_1^a$.

Lemma 2. *(i) Assume that x is such that we have*

$$n_0^{a^*} \geq \frac{3(n_0^{a^*} + n_1^{a^*})}{4}.$$

Then, with probability $1 - o(1/n)$, x is assigned (correctly) to 1 in step 2.1 of algorithm B.

(ii) Assume that x is such that we have

$$n_1^{a^*} \geq \frac{3(n_0^{a^*} + n_1^{a^*})}{4}.$$

Then, with probability $1 - o(1/n)$, x is assigned (correctly) to 0 in step 2.1 of algorithm B.

(iii) With probability $1 - o(1)$, each variable $y \in D^a$ is assigned to its correct value y^{a^*} by the algorithm B.

Proof. We first prove (iii). Suppose that y is assigned to 1 in stage 2.1. The case where y is assigned to 0 is similar. We have to prove that $n_0^{a^*} \geq n_1^{a^*}$ with probability $1 - o(1/n)$ since if in an optimum solution $x_i = 1$ then $n_0^{a^*} \geq n_1^{a^*}$. Thus, Lemma 1 applied to the graph $G_{x,0}$ with $d = \frac{2n_0^{a^*}}{n(n-1)}$ and the samples S_1 and S_2 gives

$$\Pr\left(m_0^a \leq \frac{8 \cdot 2n_0^{a^*} m^2}{7n(n-1)}\right) = 1 - o(1/n),$$

and so,

$$\Pr\left(n_0^{a^*} \geq \frac{7m_0^a n(n-1)}{2 \cdot 8m^2}\right) = 1 - o(1/n).$$

Also, Lemma 1 applied to the union of the graphs $G_{x,0}$ and $G_{x,1}$ with $d = \frac{2n^{a^*}}{n(n-1)}$ and the samples S_1 and S_2 gives

$$\Pr\left(m^a \geq \frac{8 \cdot 2n^{a^*} m^2}{9n(n-1)}\right) = 1 - o(1/n),$$

and so,

$$\Pr\left(n^{a^*} \leq \frac{9m^a n(n-1)}{2 \cdot 8m^2}\right) = 1 - o(1/n).$$

Since y takes value 1 in stage 2.1 and $m_0^a \geq 2/3m^a$,

$$\Pr\left(\frac{n_0^{a^*}}{n^{a^*}} \geq \frac{7 \cdot 2}{9 \cdot 3}\right) = 1 - o(1/n),$$

and so ,

$$\Pr\left(\frac{n_0^{a^*}}{n^{a^*}} \geq \frac{1}{2}\right) = 1 - o(1/n).$$

Assertion (iii) follows.

Now we prove (i). The proof of (ii) is completely similar to that of (i). Lemma 1 applied to the graph $G_{x,0}$ with $d = \frac{2n_0^{a^*}}{n(n-1)}$ and the samples S_1 and S_2 gives

$$\Pr\left(m_0^a \geq (1-\epsilon)\frac{2m^2}{n(n-1)} n_0^{a*}\right) = 1 - o(1/n).$$

Let $m^a = m_0^a + m_1^a$. We apply now Lemma 1 to the union of the graphs $G_{x,0}$ and $G_{x,1}$. This gives

$$\Pr\left(m^a \leq (1+\epsilon)\frac{2m^2}{n(n-1)} n^{a^*}\right) = 1 - o(1/n).$$

Substraction gives

$$\Pr\left(m_0^a - \frac{2m^a}{3} \geq \frac{2m^2}{n(n-1)}\left((1-\epsilon)n_o^{a^*} - (1+\epsilon)\frac{2(n_0^{a^*} + n_1^{a^*})}{3}\right)\right) = 1 - o(1/n).$$

Using the inequality $n_0^{a^*} + n_1^{a^*} \leq \frac{4n_0^{a^*}}{3}$, we obtain

$$\Pr\left(m_0^a - \frac{2m^a}{3} \geq \frac{2m^2}{n(n-1)}\frac{1-20\epsilon}{9}n_o^{a^*}\right) = 1 - o(1/n),$$

and fixing $\epsilon = 1/20$,

$$\Pr\left(m_0^a - \frac{2m^a}{3} \geq 0\right) = 1 - o(1/n),$$

concluding the proof.

Lemma 3. *With probability $1 - o(1)$, the number of variables undefined after the completion of stage 2.1 satisfies*

$$|V^a| \leq \frac{4 \text{ opt}}{\delta n^2}.$$

Proof. Assume that x is undefined. We have thus simultaneously $n_0^{a^*} < \frac{3}{4}(n_0^{a^*} + n_1^{a^*})$ and $n_1^{a^*} < \frac{3}{4}(n_0^{a^*} + n_1^{a^*})$ and so $n_1^{a^*} > \frac{1}{4}(n_0^{a^*} + n_1^{a^*})$ and $n_0^{a^*} > \frac{1}{4}(n_0^{a^*} + n_1^{a^*})$. Since x appears in at least δn^2 equations, $n_0^{a^*} + n_1^{a^*} \geq \delta n^2$. Thus,

$$opt \geq \min\{n_0^{a^*}, n_1^{a^*}\} \cdot |V^a| \geq \frac{\delta n^2}{4}|V^a|.$$

The assertion of the lemma follows.

We can now complete the correctness proof. Let val denote the value of the solution given by our algorithm and let opt be the value of an optimum solution.

Theorem 1. *Let ϵ be fixed. If opt $\leq \alpha n^3$ where α is sufficiently small, then we have that val $\leq (1 + \epsilon)$opt.*

Proof. Let us write

$$\text{val} = \text{val}_1 + \text{val}_2 + \text{val}_3 + \text{val}_4$$

where:

- val_1 is the number of satisfied equations with all variables in U^a
- val_2 is the number of satisfied equations with all variables in V^a
- val_3 is the number of satisfied equations with two variables in U^a and one in V^a
- val_4 is the number of satisfied equations with one variable in U^a and two in V^a.

With an obvious intended meaning, we write also

$$\text{opt} = \text{opt}_1 + \text{opt}_2 + \text{opt}_3 + \text{opt}_4$$

We have clearly $\mathrm{val}_1 = \mathrm{opt}_1$ and $\mathrm{val}_3 \leq \mathrm{opt}_3$. Thus,

$$\begin{aligned}\mathrm{val} &\leq \mathrm{opt} + \mathrm{val}_2 - \mathrm{opt}_2 + \mathrm{val}_4 - \mathrm{opt}_4 \\ &\leq \mathrm{opt} + \mathrm{val}_2 + \mathrm{val}_4 \\ &\leq \mathrm{opt} + \frac{|V^a|^3}{6} + n\frac{|V^a|^2}{2},\end{aligned}$$

and, using Lemma 3,

$$\begin{aligned}\mathrm{val} &\leq \mathrm{opt} + \frac{4^3\mathrm{opt}^3}{6\delta^3 n^6} + n\frac{4^2\mathrm{opt}^2}{2\delta^2 n^4} \\ &\leq \mathrm{opt}\left(1 + \frac{32\mathrm{opt}^2}{3\delta^3 n^6} + \frac{8\mathrm{opt}}{\delta^2 n^3}\right).\end{aligned}$$

Since $\mathrm{opt} \leq \alpha n^3$ then,

$$\begin{aligned}\mathrm{val} &\leq \mathrm{opt}\left(1 + \frac{32\alpha^2}{3\delta^3} + \frac{8\alpha}{\delta^2}\right) \\ &\leq \mathrm{opt}(1+\epsilon)\end{aligned}$$

for $\alpha \leq \frac{\delta^2\epsilon}{9}$ and sufficiently small ϵ.

5 Extensions to Dense Min-Ek-Lin2

We are able to extend our result to arbitrary k-ary versions of the problem, i.e. to dense Min-Ek-Lin2 for arbitrary k. This requires a bit more subtle construction, and the design of a density sampler for $(k-1)$-uniform hypergraphs. This extension appear in the final version of the paper [4].

Acknowledgments. The authors thank Madhu Sudan and Luca Trevisan for stimulating remarks and discussions.

References

1. S. Arora, L. Babai, J. Stern and Z. Sweedyk, *The Hardness of Approximate Optima in Lattice, Codes, and Systems of Linear Equations*, Proc. of 34th IEEE FOCS, 1993, 724–733.
2. S. Arora, D. Karger and M. Karpinski, *Polynomial Time Approximation Schemes for Dense Instances of NP-hard Problems*, Proc. of 27th ACM STOC, 1995, 284–293; the full paper appeared in Journal of Computer and System Sciences 58 (1999), 193–210.
3. C. Bazgan and W. Fernandez de la Vega, *A Polynomial Time Approximation Scheme for Dense* Min 2Sat, Proc. Fundamentals of Computation Theory, LNCS 1684, Springer, 1999, 91–99.

4. C. Bazgan, W. Fernandez de la Vega and M. Karpinski, *Polynomial Time Approximation Schemes for Dense Instances of Minimum Constraint Satisfaction*, ECCC Technical Report TR01-034, 2001.
5. P. Berman and M. Karpinski, *Approximating Minimum Unsatisfiability of Linear Equations*, Proc. 13th ACM-SIAM SODA, 2002, 514–516.
6. A.E.F. Clementi and L. Trevisan, *Improved Non-Approximability Results for Vertex Cover with Density Constraints*, Proc. of 2nd Conference on Computing and Combinatorics, 1996, Springer, 1996, 333–342.
7. I. Dinur, G. Kindler, R. Raz and S. Safra, *An Improved Lower Bound for Approximating CVP*, 2000, submitted.
8. I. Dinur, G. Kindler and S. Safra, *Approximating CVP to Within Almost Polynomial Factors is NP-Hard*, Proc. of 39th IEEE FOCS, 1998, 99–109.
9. W. Fernandez de la Vega, *Max-Cut Has a Randomized Approximation Scheme in Dense Graphs*, Random Structures and Algorithms, 8(3) (1996), 187–198.
10. W. Fernandez de la Vega and M. Karpinski, *On Approximation Hardness of Dense TSP and Other Path Problem*, Information Processing Letters 70, 1999, 53–55.
11. J. Hastad, *Some Optimal Inapproximability Results*, Proc. of 29th ACM STOC, 1997, 1–10.
12. W. Hoeffding, *Probability Inequalities for Sums of Bounded Random Variables*, Journal of the American Statistical Association, 58(301), 1964, 13–30.
13. M. Karpinski, *Polynomial Time Approximation Schemes for Some Dense Instances of NP-Hard Optimization Problems*, Randomization and Approximation Techniques in Computer Science, LNCS 1269, Springer, 1997, 1–14.
14. M. Karpinski and A. Zelikovsky, *Approximating Dense Cases of Covering Problems*, ECCC Technical Report TR 97-004, 1997, appeared also in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 40, 1998, 169–178.
15. S. Khanna, M. Sudan and L. Trevisan, *Constraint Satisfaction: The Approximability of Minimization Problems*, Proc. of 12th IEEE Computational Complexity, 1997, 282–296.

Call Control with k Rejections*

R. Sai Anand**, Thomas Erlebach, Alexander Hall**, and Stamatis Stefanakos

Computer Engineering and Networks Laboratory (TIK),
ETH Zürich, CH-8092 Zürich, Switzerland
{anand|erlebach|hall|stefanak}@tik.ee.ethz.ch

Abstract. Given a set of connection requests (calls) in a communication network, the call control problem is to accept a subset of the requests and route them along paths in the network such that no edge capacity is violated, with the goal of rejecting as few requests as possible. We investigate the complexity of parameterized versions of this problem, where the number of rejected requests is taken as the parameter. For the variant with pre-determined paths, the problem is fixed-parameter tractable in arbitrary graphs if the link capacities are bounded by a constant, but W[2]-hard if the link capacities are arbitrary. If the paths are not pre-determined, no FPT algorithm can exist even in series-parallel graphs unless $\mathcal{P} = \mathcal{NP}$. Our main results are new FPT algorithms for call control in tree networks with arbitrary edge capacities and in trees of rings with unit edge capacities in the case that the paths are not pre-determined.

1 Introduction

Given a set of connection requests in a communication network, call admission control is the problem of determining which of the requests can be accepted without exceeding the capacity of the network. The goal is to maximize the number of accepted requests or, equivalently, to minimize the number of rejected requests. In [2], Blum et al. argue that, when considering approximation algorithms, it is meaningful to consider the number of rejected requests as optimization criterion, because the number of rejected requests is expected to be small in practice due to overprovisioning of network resources.

In this paper, we consider call admission control from the viewpoint of parameterized complexity [7]. Roughly speaking, the approach of parameterized complexity is to capture a part of the input or output of an $\mathcal{NP}$-hard problem by a *parameter*, usually denoted by k. Then one tries to find a *fixed-parameter tractable* (FPT) algorithm for the problem, i.e., an algorithm with running-time $O(f(k) \cdot \mathrm{poly}(n))$, where $f(k)$ is an arbitrary function of the parameter and $\mathrm{poly}(n)$ is a polynomial function of the input size n. If such an algorithm exists, this means that there is a good hope of solving large instances of the problem efficiently provided that the value of the parameter k is small. Thus, the so-called *combinatorial explosion* that is typical for $\mathcal{NP}$-hard problems can be restricted to the parameter.

* Research partially supported by the Swiss National Science Foundation under Contract No. 21-63563.00 (Project AAPCN).

** Supported by the joint Berlin/Zurich graduate program Combinatorics, Geometry, and Computation (CGC), financed by ETH Zurich and the German Science Foundation (DFG).

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 308–317, 2002.

Motivated by the arguments given above, we consider a parameterized version of the call admission control problem and take the number of rejected requests as the parameter. We investigate different variants of the problem with respect to the network topology, the link capacities, and the existence of pre-determined routes for the requests.

Preliminaries. In the undirected version of the problem, the communication network is modeled as an undirected graph $G = (V, E)$ with edge capacities $c : E \to \mathbb{N}$. Connection requests are represented by pairs of nodes. Accepting a request (u, v) requires reserving one unit of bandwidth on all edges along a path from u to v. A set of paths is *feasible* if no edge $e \in E$ is contained in more than $c(e)$ paths. For a given set P of paths in a graph with edge capacities, an edge e is called *violated* (by P) if the number of paths in P that contain e is greater than $c(e)$. We say that a path p in P *contains a maximal set of violated edges* if there is no other path q in P such that the set of violated edges in p is a strict subset of the set of violated edges in q.

The basic call admission control problem can now be defined as follows.

CallControl: *Given a set R of requests in an undirected graph $G = (V, E)$ and a capacity function $c : E \to \mathbb{N}$, compute a subset $A \subseteq R$ and assign a path to each request in A such that the resulting set of paths is feasible. The goal is to minimize the number $|R \setminus A|$ of rejected paths.*

CallControl is $\mathcal{NP}$-hard even in undirected trees with edge capacities 1 or 2 [10] and in trees of rings with unit edge capacities [8]. Note that in the case of unit edge capacities, all accepted requests must be routed along edge-disjoint paths.

We introduce the following parameterized version of the problem:

CallControl-k: *Given a set R of requests in an undirected graph $G = (V, E)$, a capacity function $c : E \to \mathbb{N}$, and an integer $k \geq 0$, compute a subset $A \subseteq R$ and assign a path to each request in A such that the resulting set of paths is feasible and at most k requests are rejected, or decide that no such subset exists.*

There are several interesting variations of the call control problem. First, it is sometimes the case that the assignment of paths to requests cannot be determined by the algorithm, but is pre-determined by other constraints (such as routing protocols). In this case, an instance of the problem includes, for every request $r = (u, v)$, an undirected path p_r from u to v. If r is accepted, it must be routed along p_r. We denote this variant of CallControl by CallControlPre and its parameterized version by CallControlPre-k.

Sometimes it is meaningful to model the communication network as a *directed* graph $G = (V, E)$. In this case, we assume that a request (u, v) asks for a *directed* path from u to v in G. A special case of directed graphs are the *bidirected* graphs, i.e., graphs that can be obtained from an undirected graph by replacing each undirected edge with two directed edges of opposite directions. Therefore, we use terms like "CallControl in bidirected trees of rings" to specify a concrete variant of the problem.

We always use $|I|$ to denote the size of a given instance of a call control problem. An algorithm for a parameterized call control problem is an *FPT algorithm* if its running-time is bounded by $O(f(k) \cdot \mathrm{poly}(|I|))$ for some function f.

Specific network topologies that we consider are chains (graphs consisting of a single path), rings (graphs consisting of a single cycle with at least three nodes), trees (connected, acyclic graphs) and trees of rings, as well as their bidirected versions. An

undirected graph is a *tree of rings* if it can be obtained by starting with a ring and then, repeatedly, adding a disjoint ring to the graph and then identifying one node of the new ring with an arbitrary node of the existing graph.

A standard technique in the tool-box of parameterized complexity is the method of *bounded search trees* [7]. This technique tackles a problem by searching for a solution in a search tree whose size (number of nodes) is bounded by a function of the parameter only, for example, 2^k. If the work at each node of the search tree is polynomial in the size of the instance, an FPT algorithm is obtained.

This technique lends itself nicely to the parameterized call control problem: If we try to find a solution that rejects at most k requests, we identify a small set R_{rej} of *rejection candidates*. This set must have the property that, if a solution rejecting at most k requests exists at all, then there exists a solution rejecting at most k requests that rejects at least one request in R_{rej}. Thus we can branch for each request $r \in R_{\text{rej}}$ and check recursively whether the set $R \setminus \{r\}$ admits a solution rejecting at most $k-1$ requests. If the resulting search tree is explored up to depth k and no solution is found, we know that no solution rejecting at most k requests can exist.

We will apply the technique of bounded search trees in order to obtain FPT algorithms for parameterized call control problems. The interesting part of each FPT algorithm will be how a set R_{rej} of rejection candidates can be identified and how its size can be proved to be bounded by a constant or by a function of k.

Our Results. In Sect. 2, we give FPT results and hardness results of parameterized call control problems that follow easily from existing results: CALLCONTROLPRE-k is FPT in general graphs provided that the edge capacities are bounded by a constant. If the edge capacities can be arbitrary, CALLCONTROLPRE-k contains HITTINGSET as a special case and is thus W[2]-hard. CALLCONTROL-k is $\mathcal{NP}$-hard even for $k = 0$ and unit edge capacities, implying that there cannot be an FPT algorithm unless $\mathcal{P} = \mathcal{NP}$. These results apply to undirected and bidirected graphs. Our main results are given in Sections 3 and 4. In Sect. 3, we devise an FPT algorithm for CALLCONTROL-k in trees with arbitrary edge capacities. In Sect. 4, we present an FPT algorithm for CALLCONTROL-k in trees of rings with unit edge capacities. Finally, we draw our conclusions in Sect. 5.

Previous Work on Call Control Algorithms. We are not aware of any previous work on parameterized versions of call control problems. Previous work on call admission control has focused on on-line algorithms (that receive the connection requests over time and must accept or reject each request without knowledge about the future) and approximation algorithms. In most of this work, the number of accepted requests has been used as the objective function. A survey of known results on on-line call control algorithms can be found in the book by Borodin and El-Yaniv [4, Chapter 13].

However, call control problems are not only interesting in the on-line scenario. A number of researchers have studied off-line approximation algorithms for the maximum edge-disjoint paths problem (MEDP), i.e., CALLCONTROL with unit edge capacities and with the number of accepted requests as the objective function. The problem is polynomial for chains, rings, and undirected trees. Constant-factor approximation algorithms have been found for bidirected trees [9], trees of rings [8], and a class of graphs including two-dimensional meshes [13]. For general directed graphs with m edges, it is known that no approximation algorithm can achieve ratio $O(m^{\frac{1}{2}-\varepsilon})$ for any $\varepsilon > 0$

unless $\mathcal{P} = \mathcal{NP}$ [11]. Approximation algorithms with ratio $O(\sqrt{m})$ have been found for MEDP [12] and also for its generalization to arbitrary edge capacities, bandwidth requirements and profit values associated with the requests, the unsplittable flow problem [1] (the ratio increases by a logarithmic factor if the largest bandwidth requirement can exceed the smallest edge capacity). We point out that, unlike the unsplittable flow problem, all requests have the same bandwidth requirement in our definition of the call control problem.

On-line algorithms and approximation algorithms for CallControlPre with the number of rejected requests as objective function were studied by Blum et al. [2]. They observed that, in the case of unit edge capacities, this problem is a special case of Vertex-Cover. Concerning on-line algorithms, they gave a 2-competitive algorithm for chains with arbitrary capacities, a $(c+1)$-competitive algorithm for arbitrary networks with capacities bounded by c, and an $O(\sqrt{m})$-competitive algorithm for arbitrary networks with m edges and arbitrary edge capacities. Their algorithms are allowed to preempt (reject) requests that have been accepted earlier. Furthermore, they presented an off-line $O(\log m)$-approximation algorithm for arbitrary graphs and arbitrary edge capacities.

2 General Networks

First, let us consider the problem CallControlPre-k for networks with unit edge capacities, i.e., $c(e) = 1$ for all $e \in E$. A set of accepted paths is feasible if and only if the paths are pairwise edge-disjoint. The *conflict graph* of a given set P of paths is the graph $H = (P, E')$ with a vertex for each path in P and an edge between two vertices if the corresponding paths share an edge. Then, any feasible subset $A \subseteq P$ is an independent set in H, and its complement $P \setminus A$ is a vertex cover in H, i.e., a subset of the vertices such that each edge has at least one endpoint in the subset.

Therefore, checking whether there exists a feasible solution that rejects at most k paths is equivalent to determining whether H contains a vertex cover of size at most k. The vertex cover problem is well known to be FPT [7]; the best known algorithm so far has running time $O(kn + 1.271^k k^2)$ for deciding whether a graph on n nodes has a vertex cover of size at most k [6]. Thus, CallControlPre-k is FPT for arbitrary graphs with unit edge capacities.

Now assume that the edge capacities are bounded by a constant c. If a set of paths violates some edge e, then we can obtain a set P_{rej} of rejection candidates by taking an arbitrary set of $c(e)+1$ paths containing e. Thus we obtain a search tree of depth k and branching degree at most $c+1$. The size of this tree is $O((c+1)^k)$. The task to be carried out at each node of the search tree is determining whether there exists a violated edge e and, if so, picking $c(e)+1$ paths through e to obtain the set P_{rej}. This can easily be done in polynomial time. Hence, there is an algorithm with running-time $O((c+1)^k \cdot \text{poly}(|I|))$ for CallControlPre-k in arbitrary graphs provided that all edge capacities are bounded by the constant c. This discussion leads to the following proposition.

Proposition 1. CallControlPre-k *is fixed-parameter tractable for arbitrary directed or undirected graphs if the edge capacities are bounded by a constant.*

Proposition 1 leaves open the cases where the edge capacities can be arbitrarily large and/or the paths are to be determined by the algorithm. We show that these cases are unlikely to be FPT for arbitrary graphs. Our hardness results apply even to series-parallel graphs, a very restricted subclass of planar graphs with treewidth at most two [3].

Proposition 2. *If the edge capacities can be arbitrarily large,* CallControlPre-k *is* $W[2]$*-hard even for series-parallel graphs.*

Hardness for $W[t]$ for some $t \geq 1$ is considered strong evidence that a problem is not FPT [7]. The proof of Proposition 2, which is omitted due to lack of space, shows that HittingSet-k is a special case of CallControlPre-k. An instance of HittingSet-k consists of a family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of a ground set $U = \{1, 2, \ldots, n\}$ and a parameter k. The problem is to determine whether there is a subset $T \subseteq U$, $|T| \leq k$, such that T "hits" all sets in $\mathcal{S}$, i.e., $T \cap S_i \neq \emptyset$ for all $1 \leq i \leq m$. HittingSet-k is W[2]-complete [7].

If a problem is FPT, this implies that the problem is polynomial for each fixed value of the parameter k. Therefore, the following proposition shows that CallControl-k is very unlikely to be FPT even for series-parallel graphs.

Proposition 3. CallControl-k *is* $\mathcal{NP}$*-hard for* $k = 0$ *in series-parallel graphs with unit edge capacities.*

Proof. For $k = 0$, the problem CallControl-k with unit edge capacities reduces to checking whether *all* requests can be accepted and routed along edge-disjoint paths. This edge-disjoint paths problem has been proved to be $\mathcal{NP}$-hard for series-parallel graphs by Nishizeki, Vygen and Zhou [15]. □

In order to get FPT results not covered by Proposition 1, we must allow arbitrary capacities or arbitrary (not pre-determined) paths. In view of Propositions 2 and 3, however, we have to restrict the class of graphs that we allow as network topologies. Therefore, we consider CallControl-k in tree networks with arbitrary edge capacities and CallControl-k in trees of rings with unit edge capacities.

3 Trees with Arbitrary Capacities

In this section, we develop FPT algorithms for trees with arbitrary edge capacities. First, we discuss the algorithm for undirected trees in detail. Then we explain how the result can be adapted to bidirected trees. Note that CallControl and CallControlPre are equivalent in trees, because paths are uniquely determined by their endpoints.

We remark that CallControl can be solved optimally in polynomial time for chain networks (using techniques of [5]) and for undirected trees with unit edge capacities, but is $\mathcal{NP}$-hard for trees already if edge capacities in $\{1, 2\}$ are allowed [10].

The Undirected Case. Let an instance of CallControl-k in trees be given by an undirected tree $T = (V, E)$ with edge capacities $c : E \to \mathbb{N}$, a set P of paths in the tree, and an integer parameter $k \geq 0$. Consider the tree to be rooted at an arbitrary node. If $k = 0$, the problem reduces to checking whether the set P is feasible, which can be done

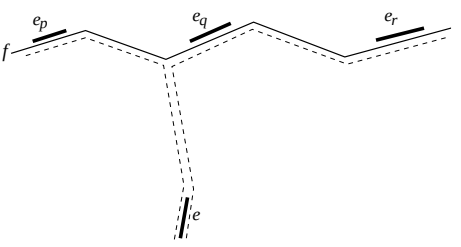


Fig. 1. Proof of Lemma 1.

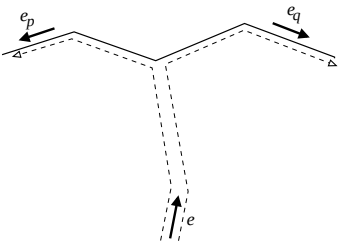


Fig. 2. The bidirected case.

efficiently. So we assume from now on that $k > 0$. If there is no violated edge, answer YES and output P as a solution. Otherwise, let e be a violated edge such that there is no other violated edge in the subtree below e. In what follows, we refer to such an edge as a *bottom-most violated edge*. The algorithm determines a set P_e of paths containing edge e such that $|P_e|$ is small enough (more precisely, $|P_e| \leq 2k$) to be taken as the set of rejection candidates for the bounded search tree technique. Then the algorithm can branch for each $p \in P_e$ and check recursively whether there exists a solution for $P \setminus \{p\}$ that rejects at most $k-1$ paths. At depth k of the search tree, k paths have been rejected, and we will have either found a feasible solution or no solution with k or less rejections exists.

Now we show how the algorithm determines a set P_e of rejection candidates satisfying $|P_e| \leq 2k$. Since e is a violated edge, at least one of the paths in P that contain e must be rejected. Moreover, it is easy to see that there exists a feasible solution that rejects a path through e which contains a maximal set of violated edges; given any feasible solution, we can construct one of the same cardinality and with the desired property by replacing a non-maximal path with a maximal one. Therefore, the algorithm needs to consider only paths that contain a maximal set of violated edges as rejection candidates.

Since we consider only one rejection at a time, we can restrict the set of rejection candidates even further by considering only one representative from each set of paths that contain the same set of violated edges. So let P_e be a set of paths that contain e satisfying the properties defined above (i.e., each path containing a maximal set of violated edges, and no two paths containing the same set of violated edges).

For each path $p \in P_e$, let e_p be the violated edge on p that is farthest from e, and let $E_e = \{e_p \mid p \in P_e\}$ be the set of all such edges e_p. (If e is the only violated edge on a path $p \in P_e$, we let $e_p = e$. This can happen only if $|P_e| = 1$.) Note that $|E_e| = |P_e|$.

Lemma 1. *No path in P can contain three edges in E_e.*

Proof. Assume to the contrary that there is a path f in P that contains three edges in E_e, say, e_p, e_q, and e_r. Without loss of generality, assume that e_q is between e_p and e_r on f, as shown in Fig. 1. Note that f cannot contain e, because otherwise the path q would not contain a maximal set of violated edges.

All three paths p, q, and r must meet f at the same node, for otherwise we would have a cycle. Assume that they meet f at a node between e_p and e_q. Then e_q is contained in r. Since e_q is the farthest violated edge in q and e is a bottom-most violated edge,

path q does not contain a maximal set of violated edges, a contradiction to the choice of P_e. The cases where the three paths p, q, and r meet f in a different node lead to a contradiction as well. □

Lemma 2. *If there exists a solution to the given instance of* CallControl-k *that rejects at most k paths, then there can be at most $2k$ paths in P_e.*

Proof. For every edge in E_e, any feasible solution must reject at least one path containing that edge. Since each path in P can contain at most two edges in E_e by Lemma 1, a feasible solution must reject at least $|E_e|/2 = |P_e|/2$ paths. Therefore, for a feasible solution with at most k rejections to exist, there can be at most $2k$ paths in P_e. □

The depth of the search tree is at most k. In a node of the search tree where i paths are already rejected, the algorithm needs to consider at most $2(k - i)$ branches; by Lemma 2, if $|P_e| > 2(k - i)$, there cannot be a feasible solution in the subtree below the current node. Thus the size of the search tree is bounded by $O(2^k k!)$. Finding a bottom-most violated edge e and determining the set of paths P_e can obviously be done in polynomial time. Thus, the algorithm solves the problem CallControl-k in time $O(2^k \cdot k! \cdot \mathrm{poly}(|I|))$, and we obtain the following theorem.

Theorem 1. *There is an FPT algorithm for* CallControl-k *in undirected tree networks with arbitrary edge capacities.*

The Bidirected Case. The FPT algorithm for call control in undirected trees can easily be adapted to bidirected trees, yielding even a better running-time. The algorithm proceeds as in the undirected case by picking a bottom-most violated edge e (which is now a directed edge) and determining a set P_e of rejection candidates that contain e and a maximal set of violated edges. The set E_e is defined as before. Since the paths and edges are now directed, it is easy to see that no path in P can contain two edges in E_e (see Fig. 2). Therefore, if after rejecting i paths, there are more than $k - i$ paths in P_e, there does not exist a feasible solution in the subtree of the current node of the search tree. Thus the size of the search tree can now be bounded by $O(k!)$ and the total running time is $O(k! \cdot \mathrm{poly}(|I|))$.

Theorem 2. *There is an FPT algorithm for* CallControl-k *in bidirected tree networks with arbitrary edge capacities.*

4 Trees of Rings with Unit Capacities

In this section, we present the first FPT results for an $\mathcal{NP}$-hard call control problem where the paths for the requests must be determined by the algorithm. We restrict the network topology to be a tree of rings, and we require that all edges have capacity 1. We obtain FPT algorithms for undirected trees of rings and bidirected trees of rings.

The Undirected Case. Let an instance of CallControl-k in trees of rings with unit edge capacities be given by an undirected tree of rings $T = (V, E)$, a set S of connection requests, and an integer parameter $k \geq 0$. The algorithm must determine if there exists

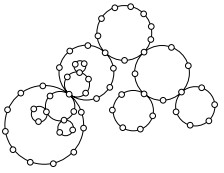


Fig. 3. A tree of rings.

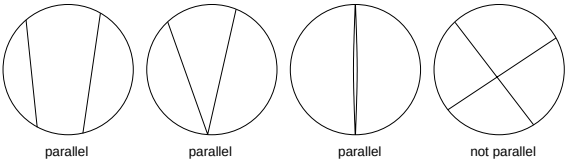


Fig. 4. Chords that are parallel or not parallel.

a subset $S' \subseteq S$ such that $|S \setminus S'| \leq k$ and the requests in S' can be routed along edge-disjoint paths in T. Again, we employ the technique of bounded search trees.

First, let us mention some simple facts about paths in trees of rings (see Fig. 3 for an example of a tree of rings). For any request (u, v), all undirected paths from u to v contain edges of the same rings. For each ring that a path from u to v passes through (i.e., contains an edge of that ring), the node at which the path enters the ring (or begins) and the node at which the path leaves the ring (or terminates) is uniquely determined by the endpoints of the path. Thus, a request (u, v) in a tree of rings can be viewed as a combination of *subrequests* in all rings that a path from u to v passes through. So, a set of requests can be routed along edge-disjoint paths if and only if all subrequests of the requests can be routed along edge-disjoint paths in the individual rings of the tree of rings. Hence, before we tackle the problem in trees of rings, we need to investigate conditions for a set of requests in a ring being routable along edge-disjoint paths.

Let R be a ring. Imagine R drawn as a circle in the plane, with its nodes distributed at equal distance along the circle. A (sub)request (u, v) between two nodes in R can be represented as a straight line segment joining u and v. We call these line segments *chords* and use the terms *chord* and *request* interchangeably if no confusion can arise. Two chords are said to be *parallel* if (1) they do not intersect, or (2) they intersect at a node in R, or (3) they are identical (see Fig. 4). If two chords are parallel then we can assign edge-disjoint paths to the corresponding requests and these paths are uniquely determined.

A *cut* in a ring R is a pair of edges in the ring. A request *crosses* a cut if each of the two possible paths connecting the endpoints of the request contains exactly one of the edges in the cut.

Lemma 3. *Given a ring R and a set S of requests in R, the requests in S can be routed along edge-disjoint paths if and only if (i) the chords of the requests in S are pairwise parallel and (ii) no cut is crossed by three requests.*

Proof (sketched). If $|S| = 1$, the lemma is trivially true. Assume $|S| \geq 2$. The "only if" part is obvious. To prove the "if" part, assume that S satisfies (i) and (ii). Consider any two requests (u, v) and (w, x) in S. By (i), they are parallel and can thus be assigned edge-disjoint paths p_1 and p_2 in a unique way. If there is a third request (y, z) in S, one can show that y and z must lie in the same chain of $R \setminus (p_1 \cup p_2)$ and so there is a path p_3 from y to z that is disjoint from p_1 and p_2. This argument can be repeated for all remaining requests, giving a construction of disjoint paths for all requests in S. □

With the result of Lemma 3 at our disposal, we are ready to obtain the FPT algorithm for CallControl-k in trees of rings with unit edge capacities. If $k = 0$, the algorithm checks if the conditions of Lemma 3 hold for each ring of the tree of rings. If this is the case, an edge-disjoint routing can be computed efficiently (the proof of Lemma 3 is constructive) and the algorithm answers YES. Otherwise, there is no edge-disjoint routing and the algorithm answers NO.

Now assume that $k > 0$. If the condition of Lemma 3 holds for the subrequests in each ring of the tree of rings, the answer is YES and an edge-disjoint routing for all requests is obtained. Otherwise, there are either two subrequests in a ring that are not parallel, so that at least one of the two must be rejected, or there are three subrequests crossing a cut in some ring, so that at least one of the three must be rejected. In the former case, we get a set of two rejection candidates, in the latter case, we have three rejection candidates. For each request r in the set of rejection candidates, we check recursively whether there exists a solution rejecting at most $k-1$ requests from $S \setminus \{r\}$. The degree of any node in the search tree is at most 3. Since the depth of the search tree is bounded by k, the size of the search tree is $O(3^k)$. As the conditions of Lemma 3 can be checked easily in polynomial time at each node of the search tree, we obtain an FPT algorithm with running-time $O(3^k \cdot \mathrm{poly}(|I|))$.

Theorem 3. *There is an FPT algorithm for* CallControl-k *in undirected trees of rings with unit edge capacities.*

The above discussion shows also that CallControl-k in undirected trees of rings with unit edge capacities can be seen as an instance of the problem HittingSet-k in which each set has cardinality at most 3, i.e., of the 3-HittingSet-k problem: the ground set U consists of the requests S, and the family $\mathcal{S}$ of subsets of U consists of all sets of two requests whose subrequests in some ring are not parallel and all sets of three requests whose subrequests cross a cut in some ring. For 3-HittingSet-k, an FPT algorithm with running-time $O(2.311^k + n)$, where n is the size of the input, is given in [14]. Thus, by transforming a given instance of CallControl-k into an instance of 3-HittingSet-k, we obtain an FPT algorithm for CallControl-k in undirected trees of rings with unit edge capacities that runs in time $O(2.311^k + \mathrm{poly}(|I|))$.

The Bidirected Case. We turn to bidirected trees of rings. Each accepted request (u, v) must now be assigned a *directed* path from u to v. As in the undirected case, a set of requests is feasible if and only if all subrequests are feasible in the individual rings. Consider a set S of (sub)requests in a ring. We proceed by doing a case analysis. In each case, we either find that all requests can be routed along edge-disjoint paths, or we are able to identify (in polynomial time) a set S_{rej} of requests such that at least one request in S_{rej} must be rejected in any feasible solution. The cardinality of S_{rej} is at most 5. The details of the case analysis are omitted due to lack of space. We can perform this case analysis for the subrequests of all requests in each individual ring of the tree of rings. Thus, we can apply the bounded search tree technique and get an FPT algorithm for CallControl-k in bidirected trees of rings with unit edge capacities. The size of the search tree can be bounded by $O(5^k)$, so the running-time is $O(5^k \cdot \mathrm{poly}(|I|))$.

Theorem 4. *There is an FPT algorithm for* CallControl-k *in bidirected trees of rings with unit edge capacities.*

5 Conclusion

We have considered parameterized versions of call admission control problems. Since the number of rejected requests can often be expected to be small in practice, we have taken this number as the parameter. For arbitrary networks, the problem was shown to be fixed-parameter tractable if the paths are pre-determined and the edge capacities are bounded by a constant. If either of these restrictions is removed, it was shown that even for series-parallel graphs, the existence of FPT algorithms is unlikely. Hence, we studied trees and trees of rings as network topologies. We gave FPT algorithms for trees with arbitrary capacities and for trees of rings with unit edge capacities if the paths are not pre-determined. Both algorithms can be adapted to the bidirected case.

References

1. Y. Azar and O. Regev. Strongly polynomial algorithms for the unsplittable flow problem. In *Proceedings of the 8th Integer Programming and Combinatorial Optimization Conference (IPCO)*, LNCS 2081, pages 15–29, 2001.
2. A. Blum, A. Kalai, and J. Kleinberg. Admission control to minimize rejections. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures (WADS 2001)*, LNCS 2125, pages 155–164, 2001.
3. H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998.
4. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
5. M. C. Carlisle and E. L. Lloyd. On the k-coloring of intervals. *Discrete Applied Mathematics*, 59:225–235, 1995.
6. J. Chen, I. A. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. In *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'99)*, LNCS 1665, pages 313–324, 1999.
7. R. Downey and M. Fellows. *Parameterized Complexity*. Springer-Verlag, New York, 1997.
8. T. Erlebach. Approximation algorithms and complexity results for path problems in trees of rings. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001)*, LNCS 2136, pages 351–362, 2001.
9. T. Erlebach and K. Jansen. The maximum edge-disjoint paths problem in bidirected trees. *SIAM Journal on Discrete Mathematics*, 14(3):326–355, 2001.
10. N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.
11. V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd, and M. Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 19–28, 1999.
12. J. Kleinberg. *Approximation algorithms for disjoint paths problems*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1996.
13. J. Kleinberg and É. Tardos. Disjoint paths in densely embedded graphs. In *Proc. of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 52–61, 1995.
14. R. Niedermeier and P. Rossmanith. An efficient fixed parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms*, 2(1), 2001.
15. T. Nishizeki, J. Vygen, and X. Zhou. The edge-disjoint paths problem is NP-complete for series-parallel graphs. *Discrete Applied Mathematics*, 115:177–186, 2001.

On Network Design Problems: Fixed Cost Flows and the Covering Steiner Problem

Guy Even[1], Guy Kortsarz[2], and Wolfgang Slany[3]

[1] Dept. of Electrical Engineering-Systems, Tel-Aviv University, Tel-Aviv 69978, Israel. guy@eng.tau.ac.il

[2] Computer Sciences Department, Rutgers University - Camden. guyk@camden.rutgers.edu

[3] Computer Science Dept., Technische Universität Wien, A-1040 Wien, Austria. wsi@dbai.tuwien.ac.at

Abstract. Network design problems, such as generalizations of the Steiner Tree Problem, can be cast as edge-cost-flow problems (a.k.a. fixed-charge flows). We prove a hardness result for the Minimum Edge Cost Flow Problem (MECF). Using the one-round two-prover scenario, we prove that MECF in directed graphs does not admit a $2^{\log^{1-\varepsilon} n}$-ratio approximation, for every constant $\varepsilon > 0$, unless $NP \subseteq DTIME(n^{\text{polylog}n})$. A restricted version of MECF, called Infinite Capacity MECF (ICF), is defined as follows: (i) all edges have infinite capacity, (ii) there are multiple sources and sinks, where flow can be delivered from every source to every sink, (iii) each source and sink has a supply amount and demand amount, respectively, and (iv) the required total flow is given as part of the input. The goal is to find a minimum edge-cost flow that meets the required total flow while obeying the demands of the sinks and the supplies of the sources. We prove that directed ICF generalizes the Covering Steiner Problem. We also show that the undirected version of ICF generalizes several network design problems, such as: Steiner Tree Problem, k-MST, Point-to-point Connection Problem, and the generalized Steiner Tree Problem. An $O(\log x)$-approximation algorithm for undirected ICF is presented, where x denotes the required total flow. We also present a bi-criteria approximation algorithm for directed ICF. The algorithm for directed ICF finds a flow that delivers half the required flow at a cost that is at most $O(n^{\varepsilon}/\varepsilon^5)$ times bigger than the cost of an optimal flow. The running time of the algorithm for directed ICF is $O(x^{2/\varepsilon} \cdot n^{1+1/\varepsilon})$. Finally, randomized approximation algorithms for the Covering Steiner Problem in directed and undirected graphs are presented. The algorithms are based on a randomized reduction to a problem called $\frac{1}{2}$-Group Steiner. This reduction can be derandomized to yield a deterministic reduction. In directed graphs, the reduction leads to a first non-trivial approximation algorithm for the Covering Steiner Problem. In undirected graphs, the resulting ratio matches the best ratio known [KRS01], via a much simpler algorithm.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 318–327, 2002.

1 Results

We improve the hardness result of Krumke *et al.* [KNS+98] for approximating the Minimum Edge Cost Flow Problem (MECF) (see [GJ79, ND32]) using a reduction from one-round two-prover protocols (see [AL96]). We show that MECF with uniform edge-prices does not admit a $2^{\log^{1-\epsilon} n}$-ratio approximation for any constant $\epsilon > 0$ unless $NP \subseteq DTIME(n^{\text{polylog}n})$. This hardness holds even if only two edge capacity values are allowed, namely, $c(e) \in \{1, poly(n)\}$, for every e.

We present a bi-criteria approximation algorithm for directed ICF showing that the results for the directed Steiner problem can be essentially generalized to this much more general problem. First, we present an algorithm that finds a flow with half the required total flow, the cost of which is $O(n^{\varepsilon}/\varepsilon^5)$ times the cost of an optimal flow. The running time of the algorithm is $O(x^{2/\varepsilon} \cdot n^{1+1/\varepsilon})$, where x denotes the requires total flow. (Scaling is applied when the total flow x is non-polynomial.) This algorithm is a greedy algorithm and it generalizes the ideas in the algorithms of Charikar *et al.* [CCC+99] and Kortsarz & Peleg [KP99]. A bi-criteria algorithm for directed ICF is presented that increases the flow amount to $(1-\varepsilon') \cdot x$ with a multiplicative overhead both in the running time and in the approximation ratio that is exponential in $1/\varepsilon'$. For a constant ε', this implies more total flow with the same asymptotic running time and approximation ratio.

We present an $O(\log x)$-approximation algorithm for undirected ICF, where x is the required total flow. Our algorithm is based on a modification of the approximation algorithm of Blum *et al.* [BRV99] for the node-weighted k-Steiner Tree Problem. An interesting open question is whether undirected ICF admits an $O(1)$-ratio approximation algorithm. Such an approximation algorithm would give a uniform $O(1)$-approximation both for k-ST, and for the generalized Steiner problem (just to give two examples).

We present the first non-trivial approximation algorithm for the Directed Covering Steiner Problem. The randomized algorithm has an approximation ratio of $O(\frac{n^{\varepsilon}}{\varepsilon^2} \cdot \log n)$ and a running time $\tilde{O}(n^{3/\varepsilon})$. Derandomization, using 2-universal hash functions, is possible at a quadratic increase in the running time. Our randomized algorithm is based on a randomized reduction to a problem called $\frac{1}{2}$-Group Steiner. In the $\frac{1}{2}$-Group Steiner Problem the objective is to find a min-cost subgraph that contains a vertex of g_i for at least half the groups. We solve the $\frac{1}{2}$-Group Steiner Problem in the directed case using a reduction to the directed Steiner Tree Problem approximated by [CCC+99].

The reduction from the Covering Steiner Problem to the $\frac{1}{2}$-Group Steiner Problem is used to obtain a simple algorithm for the Undirected Covering Steiner Problem. Our approximation ratio is the same as [KRS01] but our algorithm is much simpler.

2 Problem Definitions

A network is a 4-tuple $N = (V, E, c, p)$ where (V, E) is a graph, $c(e)$ are edge capacities, and $p(e)$ are edge prices. Given a source s and the sink t, an st-flow

is a function defined over the edges that satisfies capacity constraints, for every edge, and conservation constraints, for every vertex, except the source and the sink. The net flow that enters the sink t is called the *total flow*, and is denoted by $|f|$.

The *support* of a flow f is the set of edges that deliver positive flow, namely, the set $\{e \in E : f(e) > 0\}$. We denote the support of f by $\chi(f)$. The price of a subset of edges $F \subseteq E$ is the sum of the prices of edges in F. We denote the price of F by $p(F)$. The price of a flow f in the fixed cost model is $p(\chi(f))$.

The following problem is NP-Complete [GJ79, ND32].

The Minimum Edge-Cost Flow Problem (MECF).

Instance:

- A network $N = (V, E, c, p)$ consisting of a (directed or undirected) graph (V, E), edge capacities $c(e)$, and edge prices $p(e)$.
- A budget P.

Question: Is there a maximum st-flow f in N such that $p(\chi(f)) \leq P$?

Instead of considering a single source and sink, one may consider a situation where there is a set of sources $S \subseteq V$ and a set of sinks $T \subseteq V$. The set of candidate flow paths consists of the set of paths from a source $s \in S$ to a sink $t \in T$. This version is reducible to an st-flow problem.

A *supply amount* $c(s)$ of a source $s \in S$ is an upper bound on the net flow deliverable by s. A *demand amount* $c(t)$ of a sink $t \in T$ is an upper bound on the net flow absorbed by t.

The Infinite Capacities version of MECF is defined as follows.

The Infinite Capacities Minimum Edge-Cost Flow Problem (ICF).

Instance:

- A network $N = (V, E, p)$ consisting of a (directed or undirected) graph (V, E), and edge prices $p(e)$. Every edge has an infinite capacity.
- A set of sources $S \subseteq V$ and a set of sinks $T \subseteq V$. Each source $s \in S$ has a positive integral supply amount $c(s)$. Each sink $t \in T$ has a positive integral demand amount $c(t)$.
- A required integral total flow x and a budget P.

Question: Does there exist a flow f such that $|f| \geq x$ and $p(\chi(f)) \leq P$?

Observe that, by a simple reduction, the in-degree of sources and the out-degree of sinks can be zeroed. This can be achieved by adding dummy nodes that act as sources and sinks.

We refer to a flow f with total flow x as an x-flow. We denote by $f^*(N, x)$ an optimal x-flow for an ICF instance (N, x). Namely, $f^*(N, x)$ is an x-flow with minimum support cost among the set of x-flows in N. We denote the cost of an optimal x-flow in N by $p^*(N, x)$, namely, $p^*(N, x) = p(\chi(f^*(N, x)))$. We reformulate ICF as a search problem of finding an optimal x-flow (hence an ICF instance is a pair (N, x) and the budget P is not part of the input).

The Unit ICF Problem (U-ICF). The unit demands and supplies version of ICF (U-ICF) is defined for ICF instances in which $c(s) = 1$, for every source $s \in S$, and $c(t) = 1$, for every sink $t \in T$.

3 Preliminaries

Reduced network N_f. Consider an ICF instance (N, x) and an integral flow f in N. Suppose that $|f| < x$. Let N_f be the network obtained from N by the following changes:

1. The supply of every source is decreased by the amount of flow it supplies in f. Similarly, the demand of every sink is decreased by the amount of flow it receives in f.
2. The price of every edge in $\chi(f)$ is set to zero.

Observe that the reduced network N_f does not have reverse edges as defined in residual networks when computing a max-flow. Indeed, there the capacity of a reverse edge is finite (since the capacity of a reverse edge equals the amount of flow along the corresponding edge), but finite capacities are not allowed in ICF.

Bi-criteria approximation algorithm. An algorithm A is an (α, β) bi-criteria approximation for ICF if, given (N, x), it computes a flow f that satisfies the following two conditions:

$$|f| \geq \alpha \cdot x$$
$$p(\chi(f)) \leq \beta \cdot p^*(N, x).$$

Junction trees. A directed graph is an *arborescence* rooted at r if its underlying graph is a tree, the in-degree of the root r is zero, and there is a directed path from the root r to all the other vertices. A *reverse arborescence* rooted at r is a directed graph such that the directed graph obtained by reversing the directions of all the arcs is an arborescence.

Consider a U-ICF instance (N, x) with a set of sources S and a set of sinks T. A *junction tree* rooted at r is an edge induced subgraph JT of N such that: (i) JT is the union of an arborescence G_1 and a reverse arborescence G_2 both rooted at r. (ii) The leaves of G_1 are sinks. (iii) The leaves of G_2 are sources. (iv) G_1 and G_2 have an equal number of leaves.

Observe that the total flow that can be shipped using the edges of a junction tree JT equals the number of sources in JT.

The problem of finding a low cost junction tree is defined as follows:

The Minimum Cost Junction Tree Problem (Min-JT).

Instance:

- A network $N = (V, E, p)$ consisting of a directed graph (V, E), and edge prices $p(e)$.
- A set S of sources and a set T of sinks.
- An integer x.

Goal: Find a min-cost junction tree JT with x sources (and sinks).

We denote the cost of an optimal junction tree with x sources by $JT^*(N, x)$.

In [CCC+99] the k-Directed Steiner Problem (k-DS) is defined. In this problem the goal is to find a min-cost arborescence rooted at r that spans at least k terminals from a given set of terminals. The best known approximation algorithm for k-DS is given in [CCC+99] where the following theorem is proved (see also [Z97] for earlier results).

Theorem 1. *[CCC+99]* [1] *For any $\varepsilon > 0$ there exists a $k^{\varepsilon}/\varepsilon^3$-ratio, $O(k^{2/\varepsilon} \cdot n^{1/\varepsilon})$-time approximation algorithm for k-DS.*

Min-JT generalizes k-DS, and therefore, it is not easier to approximate than Set-Cover. An approximation algorithm for Min-JT is obtained as follows. Guess the root r of the junction tree. Apply a k-DS approximation algorithm on the graph with sinks as terminals and a k-DS approximation algorithm on the reversed graph with sources as terminals. We summarize this approximation algorithm in the following corollary.

Corollary 1. *For any $\varepsilon > 0$ there exists an $x^{\varepsilon}/\varepsilon^3$-ratio, $O(x^{2/\varepsilon} \cdot n^{1+1/\varepsilon})$-time approximation algorithm for Min-JT.*

The forest lemma. The following lemma shows that we may restrict flows in ICF to forests.

Lemma 1. *For every ICF instance (N, x), there exists an optimal x-flow f^* such that the underlying graph of the graph induced by the edges of $\chi(f^*)$ is a forest.*

The following assumption is based on Lemma 1.

Assumption 2 *The underlying graph of the support of an optimal x-flow is a tree.*

We may assume that the underlying graph is a tree rather than a forest by the following modification. Add a new node v with zero in-degree and add zero cost edges from v to all the other nodes. A subset of these edges can be used to glue the forest into a tree without increasing the cost of the support. Although these glue edges do not deliver flow, we may regard them as part of the support since their cost is zero.

A decomposition lemma. Assumption 2 allows us to restrict the search to trees. A subtree $\mathcal{T}'$ of a tree $\mathcal{T}$ is said to *have an articulation vertex r*, if every path from a node in $\mathcal{T}'$ to a node in $\mathcal{T} - \mathcal{T}'$ traverses the node r. The following decomposition lemma is used for recursing when optimal x-flows avoid junction trees.

Lemma 2. *[BRV99,KP99] Let $\mathcal{T}$ be a tree with edge costs $p(e)$. Let $p(\mathcal{T})$ denote the sum of the edge costs of edges in $\mathcal{T}$. Let S be a subset of vertices in $\mathcal{T}$, and $k \leq |S|$. There exists a sub-tree $\mathcal{T}' \subseteq \mathcal{T}$ that has an articulation vertex such that*

$$|\mathcal{T}' \cap S| \in [k, 3k], \quad \text{and}$$
$$\frac{p(\mathcal{T}')}{|S \cap \mathcal{T}'|} \leq \frac{p(\mathcal{T})}{|S|}.$$

[1] This ratio is larger by an $1/\epsilon$ factor than the one claimed in [CCC+99]. The difference is due to an error in [Z97]

4 An Approximation Algorithm for U-ICF: Directed Networks

In this section we present a bi-criteria approximation algorithm for U-ICF that achieves an $(\frac{1}{2}, O\left(\frac{x^{\varepsilon}}{\varepsilon^5}\right))$-approximation ratio. The algorithm only finds an $x/2$-flow due to monotonicity (see full paper). The running time of the algorithm is $O\left(x^{2/\varepsilon} \cdot n^{1+1/\varepsilon}\right)$ which n times the running time of the Charikar *et al.* [CCC+99] algorithm. Our algorithm and its analysis are closely related and generalize the ideas from the [KP99] and [CCC+99] algorithms. In the full paper we show how to use this algorithm to approximate ICF with only a negligible loss in the ratio.

Notation. Fix $1/3 > \varepsilon > 0$. We denote the approximation ratio for the Directed k-Steiner Trees by $\tau(k)$, namely, $\tau(k) = k^{\varepsilon}/\varepsilon^3$. The *density* of a flow f is the ratio $\frac{p(\chi(f))}{|f|}$. We denote the density of a flow f by $\gamma(f)$.

The algorithm. The approximation algorithm Find-Flow for U-ICF is listed in Figure 1. Algorithm $Find - Flow(N, d, t)$ finds a d-flow f in N. The algorithm computes a flow by computing a sequence of augmenting flows. The parameter t is a threshold parameter used for a stopping condition of the recursion. Note that ε is a parameter that effects the approximation ratio and the running time. Algorithm Find-Flow invokes two procedures. The first procedure is *Min-W-Matching*(N, d) that finds a d-flow that is d-approximate using a min-weight matching, as described in the full paper. The second procedure, *Find-Aug-Flow*(N', i), computes an augmenting flow a_i in N' such $|a_i| \in [\frac{1-\varepsilon}{6} \cdot i, \frac{1}{2} \cdot i]$.

Approximation ratio. We prove the approximation ratio of Algorithm $Find - Flow$ by showing that the procedure *Find-Aug-Flow*(N', i) finds a flow the density of which competes with the density of an optimal (i/ε)-flow in the reduced network. Since the density is compared with an optimal flow in a reduced network, the analysis relies on monotonicity. In the full paper we prove the following claim:

Claim. Let $c = \frac{3}{\ln 2}$ and $\varepsilon < 1/3$. Algorithm $Find - Flow(N, x/2, \tau(x))$ finds an $x/2$-flow f in N that satisfies

$$\frac{p(f)}{p(f_x^*)} \leq \tau(x) + \frac{6}{c \cdot \varepsilon^5} \cdot x^{c \cdot \varepsilon} = O\left(\frac{x^{c \cdot \varepsilon}}{\varepsilon^5}\right).$$

Time complexity. The following claim shows that the asymptotic running time of Algorithm *Find-Aug-Flow* is n times bigger than that of the approximation algorithm for the directed Steiner Problem.

Claim. The running time of algorithm *Find-Aug-Flow*(N, i) is $O(i^{2/\varepsilon} \cdot n^{1+1/\varepsilon})$, where n denotes the number of vertices in the network N.

The following claim summarizes the running time of Algorithm $Find - Flow$.

Claim. The running time of algorithm $Find - Flow(N, d, \tau(d))$ is $O(d^{2/\varepsilon} \cdot n^{1+1/\varepsilon})$, where n denotes the number of vertices in the network N.

Algorithm $Find - Flow(N, d, t)$

1. **If** $d \leq t$ **then return** $(Min\text{-}W\text{-}Matching(N, d))$.
2. **else**
   a) $a \leftarrow Find\text{-}Aug\text{-}Flow(N, \varepsilon \cdot d)$.
   b) **Return** $(a \cup Find - Flow(N_a, d - |a|, t))$.

Algorithm $Find\text{-}Aug\text{-}Flow(N', i)$

1. Remark: $\tau(x) = x^{\varepsilon}/\varepsilon^3$.
2. **If** $i \leq \tau(i/\varepsilon)$ **then return** $(Min\text{-}W\text{-}Matching(N', i))$.
3. **else**
   a) **For** $j = 0$ **to** $\log_{1+\varepsilon} \frac{3}{1-\varepsilon}$ **do**
      i. $i(j) = \frac{1-\varepsilon}{6} \cdot (1+\varepsilon)^j \cdot i$.
      ii. $f_j \leftarrow Find\text{-}Aug\text{-}Flow(N', i(j))$.
   b) $f_{JT} \leftarrow JT(N', i/6)$.
   c) Let f be a flow with minimum density in $\{\{f_j\}_j, f_{JT}\}$.
   d) **Return** (f).

Fig. 1. Algorithm Find-Flow.

5 Improved Bi-criteria Approximation for U-ICF

In this section we present a bi-criteria algorithm for U-ICF that finds a $(1 - \varepsilon) \cdot x$-flow that is $f(\varepsilon) \cdot \rho$-approximate given a $(1/2, \rho)$-bi-criteria approximation algorithm for U-ICF. Note that although $f(\varepsilon)$ is exponential in $1/\varepsilon$, it is constant if ε is. The running time required to increase the flow from an $x/2$-flow to an $(1 - \varepsilon) \cdot x$-flow is also exponential in $1/\varepsilon$, which is again constant if ε is. The improved algorithm will be discussed in the full paper.

6 An Approximation Algorithm for ICF: Undirected Networks

In this section we present an $O(\log x)$-approximation algorithm for ICF when the network is undirected. We avoid the reduction to U-ICF to get a slightly better approximation ratio. The algorithm and its analysis will be presented in the full paper.

7 The Covering Steiner Problem

In this section we present a randomized approximation algorithm for the Covering Steiner Problem both in the undirected and the directed cases. The algorithm is based on a randomized reduction of the Covering Steiner Problem to the $\frac{1}{2}$-Group Steiner Problem.

The following theorem is proved in this section.

Theorem 3. *The undirected Covering Steiner Problem has an approximation algorithm that finds an* $O(\log n \cdot \log\log n \cdot \log(\max_i |g_i|) \cdot \log(\sum_i d_i))$ *ratio solution.*

The directed Covering Steiner Problem has a randomized approximation algorithm that in time $\widetilde{O}(n^{3/\varepsilon})$ *finds a cover that is* $O(\log n \cdot n^{\varepsilon} \cdot \frac{1}{\varepsilon^2})$*-approximate with high probability.*

In the undirected case, this result matches the approximation-ratio of [KRS01] and its main value is a significant simplification in comparison to [KRS01].

The $\frac{1}{2}$-Group Steiner Problem. In the $\frac{1}{2}$-Group Steiner Problem, the input consists of a graph $G = (V, E)$ with edge prices $p(e)$ and q (disjoint) subsets $\{g_i\}_i$ of vertices. A tree $\mathcal{T}$ in G *covers* a set g_i if $\mathcal{T} \cap g_i$ is not empty. The goal in the $\frac{1}{2}$-Group Steiner Problem is to find a min-cost tree that covers at least half the groups.

In the directed case, the $\frac{1}{2}$-Group Steiner Problem is reducible to the k-Directed Steiner Problem. See the reduction of the Group Steiner Tree Problem in [CCC+99]. This implies an approximation as in Theorem 1.

In the undirected case, the $\frac{1}{2}$-Group Steiner Problem can be approximated modifying the approximation algorithm for the Group Steiner Problem of Garg *et al.* [GKR98]. The modified algorithm has 3 stages:

First, a fractional relaxation is defined. The fractional relaxation is a min-cost single-source multi-sink flow problem. We guess the root r of an optimal solution. This root serves as the source. For every group g_i, we define a commodity i and a flow f_i from r to g_i (namely, the sinks are the nodes of g_i). The total flow $|f_i|$ is at most 1. The sum of total flows $\sum_i |f_i|$ is at least $q/2$. The max-flow $f(e)$ along an edge e equals $\max_i f_i(e)$. The objective is to minimize the sum $\sum_e p(e) \cdot f(e)$.

Second, following [GKR98], a rounding procedure for the above fractional relaxation is applied when the graph is a tree. If the graph is a tree, then the union of the supports $\bigcup_i \chi(f_i)$ induces a tree. Moreover, the directions of the flows induce an arborescence. Let $pre(e)$ denote the predecessor edge of the edge e. The rounding first picks an edge e with probability $f(e)/f(pre(e))$, and then keeps e provided that all the edges along the path from the root r to e are picked.

Observe that, for at least $q/4$ groups, $|f_i| \geq 1/4$. Otherwise the total flow is less than $q/2$. In [GKR98], it is proved that if $|f_i|$ is constant, then the rounding procedure covers g_i with probability $\Omega(\frac{1}{\log |g_i|})$. Moreover, the expected cost of the picked edges is the fractional optimum. It follows that $O(\log \max_i |g_i|)$ iterations suffice to cover half the groups, and an $O(\log \max_i |g_i|)$ expected approximation ratio follows.

Finally, the graph is approximated by a tree using the tree-metric technique of Bartal [B98]. The approximation by a tree increases the approximation ratio by $O(\log n \log\log n)$. We point out that Charikar *et al.* [CCGGP-98] presented a derandomization for an approximate tree metric and the above rounding technique.

It follows that in the undirected case the $\frac{1}{2}$-Group Steiner admits an $O(\log n \cdot \log\log n \cdot \max_i |g_i|)$-approximation. **A randomized reduction.** In this section we present a randomized reduction of the Covering Steiner Problem to the $\frac{1}{2}$-

Group Steiner Problem. This reduction essentially proves that for the group Steiner and covering Steiner are equivalent with respect to approximation.

Given a Covering Steiner instance, consider an optimal tree $\mathcal{T}$. For every group g_i, the tree $\mathcal{T}$ includes at least d_i vertices in g_i. We randomly partition g_i into d_i bins. The jth bin in g_i is denoted by g_i^j. A bin g_i^j is *empty* if $\mathcal{T} \cap g_i^j$ is empty. The probability that a bin is empty is less than $1/e$ (as this corresponds to throwing d_i balls to d_i bins). Thus, the expected number of empty bins is less than $\sum_i d_i/e$. We consider this randomized reduction to be successful if the number of empty bins is less than $\sum_i d_i/2$. The Markov Inequality implies that the reduction is successful with a constant probability. We may decrease the probability of a failure to a polynomial fraction by repeating the reduction $\Theta(\log n)$ times.

Observe that if the randomized reduction is successful, then the reduction yields a $\frac{1}{2}$-Group Steiner instance with a solution, the cost of which is at most the cost of $\mathcal{T}$.

The $\frac{1}{2}$-Group Steiner approximation algorithm finds a tree that covers half of the total demands. After $O(\log(\sum_i d_i)) = O(\log n)$ iterations, all the groups are covered. The claimed ratio follows.

Derandomization. The randomized reduction can be derandomized using 2-universal hash functions [MR95]. The derandomization incurs a quadratic increase in the running time.

8 A Hardness Result for MECF

It is natural to ask if a strong lower bound on the approximability of ICF holds. We leave this question open. However, we try to make a first step toward this direction, by giving a strong lower bound for MECF. In addition, the study of MECF is interesting by its own right, for the many important applications of this problem.

Theorem 4. *The Minimum Edge-Cost Flow Problem with uniform edge-prices does not admit a* $2^{\log^{1-\epsilon} n}$*-ratio approximation for any constant* $\epsilon > 0$ *unless* $NP \subseteq DTIME(n^{\text{polylog}n})$ *. This hardness holds even if only two edge capacity values are allowed, namely,* $c(e) \in \{1, poly(n)\}$*, for every* e*.*

Observe that: (i) Non-uniform polynomial edge prices are easily reducible to uniform edge prices. The reduction simply replaces every edge (u, v) by a path of length $p(u, v)$. (ii) If edge capacities are uniform, then MECF becomes a min-cost flow problem, and hence, polynomial. The proof of this theorem is given in the full paper.

References

[AL96] S. Arora and C. Lund, "Hardness of Approximations", In *Approximation Algorithms for NP-hard Problems*, Dorit Hochbaum, Ed., PWS Publishing , 1996.

[B98] Y. Bartal, "On approximating arbitrary metrics by tree metrics", STOC, 1998.

[BRV99] A. Blum and R. Ravi and S. Vempala A constant-factor approximation algorithm for the k-MST Problem. *JCSS*, 58:101–108, 1999.

[CCC+99] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha and M. Li. "Approximation Algorithms for directed Steiner Problems", J. of Algs., 33, p. 73-91, 1999.

[CCGGP-98] M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin "Approximating a finite metric by small number of trees", FOCS, 1998.

[DK99] Y. Dodis and S. Khanna, "Designing Networks with bounded pairwise distance", *STOC*, 750–759, 1999.

[GJ79] M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. *W.H. Freeman and Company*, 1979.

[GKR98] N. Garg and G. Konjevod and R. Ravi, A polylogarithmic approximation algorithm for the group Steiner tree problem. *SODA'98*, pages 253–259, 1998.

[GKR+99] V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd and M. Yannakakis. Near-Optimal Hardness Results and Approximation Algorithms for Edge-Disjoint Paths and related Problems. *STOC 99*.

[KR00] G. Konjevod and R. Ravi, An Approximation Algorithm for the Covering Steiner Problem. *SODA 2000*, 338–334, 2000.

[KRS01] G. Konjevod, R. Ravi, and Aravind Srinivasan, Approximation Algorithms for the Covering Steiner Problem. manuscript, 2001.

[KP99] G. Kortsarz and D. Peleg. "Approximating the Weight of Shallow Steiner Trees", Discrete Applied Math, vol 93, pages 265-285, 1999.

[KNS+98] S.O. Krumke, H. Noltemeier, S. Schwarz, H.-C. Wirth and R. Ravi. Flow Improvement and Network Flows with Fixed Costs. *OR-98*, Zürich, 1998.

[MR95] R. Motwani and P. Raghavan, Randomized Algorithms, Cambridge Univ. Press, 1995.

[Z97] A. Zelikovsky. A series of approximation algorithms for the Acyclic directed Steiner Tree Problem. *Algorithmica*, 18:99–110, 1997.

Packet Bundling*

Jens S. Frederiksen and Kim S. Larsen

Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark
{svalle,kslarsen}@imada.sdu.dk

Abstract. When messages, which are to be sent point-to-point in a network, become available at irregular intervals, a decision must be made each time a new message becomes available as to whether it should be sent immediately or if it is better to wait for more messages and send them all together. Because of physical properties of the networks, a certain minimum amount of time must elapse in between the transmission of two packets. Thus, whereas waiting delays the transmission of the current data, sending immediately may delay the transmission of the next data to become available even more.
We consider deterministic and randomized algorithms for this on-line problem, and characterize these by tight results under a new quality measure. It is interesting to note that our results are quite different from earlier work on the problem where the physical properties of the networks were emphasized less.

1 Introduction

We consider point-to-point transmission of data in a network. Transmission of data is in the form of packets, which contain some header information, such as the identification of the receiver and sender, followed by the actual data. For obvious reasons, data is also referred to as messages.

When data to be sent becomes available a little at a time at irregular intervals, the question arises on the sending side whether to send a given piece of data immediately or whether to wait for the next data to become available, such that they can be sent together. Sending the data together is referred to as *Packet Bundling*.

The reason why this is at all an issue is because of physical properties of the networks which imply that after one packet has been sent, a certain minimum amount of time must elapse before the next packet may be sent. Thus, whereas waiting for more data will certainly delay the transmission of the current data, sending immediately may delay the transmission of the next data to become available even more. In addition to reducing the overall transmission delay when bundling messages, we also reduce the bandwidth requirement of the sender,

* Supported in part by the Danish Natural Science Research Council (SNF) and in part by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 328–337, 2002.

since overhead due to packet headers and network gap is reduced. The problem of making these decisions is referred to as the *Packet Bundling Problem*.

A very similar problem, the Dynamic TCP Acknowledgment Problem, was introduced in [4]. Since it usually does not make sense to delay the transmission of large amounts of data, the focus for packet bundling is small messages, and acknowledgments to the receipt of packets are examples of such. The problem was studied as an on-line problem [2] with the cost function being the number of packets sent plus the sum of the latencies for each message. The latency of a message is the time from when the data is available until it is sent. This is known as the flow-time cost measure. An exact characterization of the optimal algorithms for the deterministic and randomized case can be found in [4] and [6], respectively.

The problem we consider is different from the Dynamic TCP Acknowledgment Problem in two ways: we build on top of an accepted model for distributed computing and, related to this decision, we also choose a different cost function.

To our knowledge, the currently most widely accepted model of computation for distributed computing is now the so-called LogP model [3], which is tractable from a theoretical point of view, but also realistic enough that good theoretical algorithms are likely to be good in practice as well; on many different platforms. The ingredients in the model are the latency L, the processor overhead in sending messages o, the gap imposed by the network between messages g, and the number of processors P. We refer to the original paper for a complete treatment.

Comparing the theoretical work of [4,6] with the model assumptions of [3], the most noticeable difference is the lack of the gap or overhead parameter in [4,6]. In their work, they allow packets to be sent arbitrarily small distances apart. This significantly influences the results. With decreasing intervals between messages, the latency of each packet contributes to the cost function. Thus, a good algorithm would send frequently; very likely faster than possible in a real world scenario.

We base our theoretical work on the LogP model, which means we respect the physical gap and overhead. At times when messages become available separated by very small time intervals, the cost function of [4,6] would impose an unreasonable large penalty, if the decision is to delay transmission, which leads us to consider another cost function: the sum of time intervals when at least one unsent message is available. In our opinion, this measure is just as natural and it has the significant advantage that good and bad algorithms can be distinguished. Using the flow-time cost function in the model where the physical gap or overhead is respected does not lead to interesting results. In fact, it has been shown [5] that any reasonable deterministic or uniform randomized algorithm has a competitive ratio of exactly two, where an algorithm is called reasonable if it does not postpone the transmission of a message by more than the sum of the gap and overhead values.

We analyze natural families of deterministic and randomized algorithms for the problem and find the best among these.

2 Packet Bundling

Referring to the description of the LogP model from the introduction, since we are considering the situation from the perspective of a single processor, there is no reason to distinguish between gap and overhead, since the crucial value is the maximum of the two. In the remainder of the paper, we normalize with respect to this maximum, and assume that it is one (the important fact is that it is different from zero).

Consider of the problem of a person A wishing to send small messages to another person B. All messages have to be sent using the same, single messenger, who uses one time unit for each delivery, i.e., when the messenger has left with one or more messages (a packet), no messages can be sent for the next one time unit.

When A decides to send a message, she can either send it immediately (assuming that the messenger is in), or wait some time (probably less than one) to see whether other messages have to be sent, so that these messages can be sent together.

The formal definition of the problem is as follows:

Definition 1. *In the* Packet Bundling Problem *one is given a sequence* $\sigma = a_1, \ldots, a_n$ *of message arrival times and is asked to give a sequence of packet times* $p_1, \ldots, p_m$ *at which packets of messages are sent. All messages are considered to be small, so that an unlimited number of messages can fit in a packet. The set of messages sent in packet* p_i *is denoted* $\tilde{p}_i$.

The packets should respect the following restrictions:

- *All packet times should be at least one unit apart, i.e.,*

$$\forall i < m\colon\ p_{i+1} - p_i \geq 1.$$

- *All messages should be sent no earlier than their arrival time, i.e.,*

$$\forall i \leq n\ \exists j \leq m\colon\ a_i \in \tilde{p}_j \wedge a_i \leq p_j.$$

If a packet is sent at time p_i*, then the set of messages contained in the packet are considered delivered at time* $p_i + 1$.

The cost function measures the total time elapsed while there are messages which have arrived, but have not been delivered:

$$\sum_{i=1}^{m} \left((p_i + 1) - \max\{(p_{i-1} + 1), \min_{a_j \in \tilde{p}_i} a_j\} \right)$$

where we define $p_0 = -\infty$*. For convenience, we identify a message with its arrival time, justifying the notation* $a_j \in \tilde{p}_i$*, assuming that these are distinct; if not,* $\tilde{p}_i$ *can be thought of as a multiset.*

We consider on-line algorithms. Thus, the algorithm is given the arrival times, a_i, one at a time, and has to decide the packet time at which the message

is sent before the arrival time of the next message (if any) is revealed. If the next message arrives before the previously chosen packet time, the algorithm is allowed to reconsider its choice. For any algorithm, ALG, and input sequence, σ, we let $ALG(\sigma)$ denote the value of the cost function when ALG is run on σ.

The performance of deterministic algorithms is measured in comparison with the optimal off-line algorithm, OPT, using the standard competitive ratio. OPT knows the entire input sequence, when it decides when to send each packet, and can hence achieve a lower cost.

An algorithm ALG is (strictly) c-competitive for a constant c, if for all input sequences, σ, the following holds: $ALG(\sigma) \leq c \cdot OPT(\sigma)$. The infimum of all such values c is called the competitive ratio of ALG.

The performance of randomized algorithms is measured likewise, though using the expected competitive ratio, $E[ALG(\sigma)]$, instead.

3 Deterministic Algorithms

In this section, we consider deterministic on-line algorithms for the problem. Specifically, we consider the following family of algorithms:

> A_k: When a message arrives, it is sent together with all messages (if any) arriving after this one at the earliest possible time after k time units.

Without loss of generality, we assume that if A_k decides to let the messenger leave at a certain point in time, and one or more messages arrive exactly when he is about to leave, then he leaves without these new messages. If this is a problem in a proof, then all messages arriving when the messenger leaves can be considered to arrive $\epsilon \in o(1)$ time units later. Because of the infimum which is taken in the definition of the competitive ratio, this will generally not alter the result.

Theorem 1. *The competitive ratio of A_k is:*

$$\mathcal{R}(A_k) = \begin{cases} 1 + \frac{1}{1+k} & , \text{ if } 0 \leq k < \hat{\varphi} \\ 1 + k & , \text{ if } \hat{\varphi} \leq k \end{cases}$$

where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ and $\hat{\varphi} = \varphi - 1$. The best ratio φ is achieved by $A_{\hat{\varphi}}$.

For a fixed algorithm, A_k, any input sequence for our problem can be divided into phases as follows: Each phase starts with the arrival of the first message after the previous phase has ended, and ends at the earliest possible time when there are no messages to deliver and the messenger is in. In the special case when the messenger returns at the exact same time a new message arrives (and no other messages are due for delivery), the phase ends, and the new message starts the next phase.

Lemma 1. *For a worst-case sequence for A_k, we can assume that the messenger carries only one message at a time.*

Proof. The short proof [5] is omitted here due to space restrictions. □

Lemma 2. *There exists a worst-case sequence for A_k where, if any messages arrive when the messenger is out, they arrive exactly at the point in time where the messenger leaves or returns.*

Proof. We transform a worst-case sequence into a sequence with the desired property which is still worst-case by repeatedly dealing with one message at a time. The detailed proof [5] is omitted here due to space restrictions. □

Proof (Proof of Theorem 1). Let us first consider the case when $k \leq 1$:

By Lemmas 1 and 2, a worst-case sequence can be assumed to consist of phases of the form $\sigma_1 = 0$ or $\sigma_n = 0, k, k+1, k+2, \ldots, k+(n-2)$, where n is the number of messages. We separate each phase from the next by more than two time units. By definition of A_k, this means that messages from different phases cannot interfere, so relative costs can be calculated separately for each phase.

A_k's cost is $A_k(\sigma_n) = k + n$, whereas OPT's cost is

$$OPT(\sigma_n) = \begin{cases} k+n-1 & , \text{ if } n > 1 \\ 1 & , \text{ if } n = 1 \end{cases}$$

For $n = 1$, this gives a competitive ratio of $\frac{A_k(\sigma_1)}{OPT(\sigma_1)} = k + 1$, and for $n > 1$, it gives a competitive ratio of $\frac{A_k(\sigma_n)}{OPT(\sigma_n)} = \frac{k+n}{k+n-1}$, which is maximized for $n = 2$, where $\frac{A_k(\sigma_2)}{OPT(\sigma_2)} = \frac{k+2}{k+1} = 1 + \frac{1}{1+k}$.

Comparing the two cases, we find that σ_2 is the worst possible for $k \leq \hat{\varphi}$, whereas σ_1 is worst for $\hat{\varphi} \leq k \leq 1$.

The case when $1 < k$ is similar [5], but omitted here due to space restrictions. □

No deterministic algorithm has a competitive ratio better than $A_{\hat{\varphi}}$:

Theorem 2. *Let ALG be any deterministic algorithm for the* Packet Bundling Problem. *Then* $\mathcal{R}(ALG) \geq \mathcal{R}(A_{\hat{\varphi}}) = \varphi$.

Proof. We show how to construct an input sequence for ALG, where it has a competitive ratio larger than or equal to φ.

The input will be given in a number of phases, each consisting of either one or two messages. Between each phase there is a time interval large enough so that neither ALG nor $A_{\hat{\varphi}}$ at the end of the interval has any messages to deliver, nor are they at the moment delivering any messages.

Let us first consider phase σ_i, and let the first message in this phase arrive at time a_{i_1}. Let k_i be the length of the time interval ALG waits before it sends the message. Referring to Theorem 1, we know for A_{k_i} whether a worst-case phase for A_{k_i} has one or two messages. If it has two, another message is set to arrive at time $a_{i_2} = a_{i_1} + k_i$, if not, the phase ends.

Referring again to Theorem 1, the following holds for phase σ_i:

$$ALG(\sigma_i) \geq A_{k_i}(\sigma_i) \geq A_{\hat{\varphi}}(\sigma_i) \geq \varphi\, OPT(\sigma_i)$$

For $k_i \leq \hat{\varphi}$, a phase consists of two messages. Whereas A_{k_i} sends the second message immediately after having sent the first, we cannot be sure that ALG does too. Thus, the costs of A_{k_i} and ALG are not necessarily the same (giving inequality instead of equality between the first two terms).

Thus, for the entire input sequence, we have $ALG(\sigma) \geq \varphi\, OPT(\sigma)$. □

4 Randomized Algorithms

We now consider a family of randomized algorithms solving the same problem. Our deterministic algorithm family A_k chose a specific k and sent a message at the earliest possible time after k time units. $RAND_\Delta$ chooses the interval it waits at random.

$RAND_\Delta$: When a new message arrives and no other messages are waiting, this message and later messages (if any) are sent after a period of time chosen uniformly between 0 and Δ, or at the first possible time hereafter.

We only consider algorithms with $\Delta \leq 1$, since any algorithm, $RAND_\Delta$, with $\Delta > 1$ easily can be seen to have a competitive ratio larger than $RAND_1$.

One could also consider other families of randomized algorithms. Instead of using a uniform distribution, we could have used an exponential distribution with parameter Δ varying from zero to infinity (as in [6]), or a cut-off exponential distribution described by the density function: $f(\delta) = \frac{1}{\Delta} e^{-\frac{\delta}{\Delta}} / \left(1 - e^{-1}\right)$ for $\delta \in [0, \Delta]$, and zero otherwise. By careful examination both of these are for any Δ easily shown to have a worse competitive ratio than the best member of the $RAND_\Delta$-family.

Without loss of generality, we will as with A_k assume that messages arriving exactly at the randomly chosen time δ when the messenger leaves will not be delivered immediately. For $\Delta > 0$, this does not make any difference to the expected competitive ratio as δ is chosen uniformly at random in the range $[0, \Delta]$. For $\Delta = 0$, $RAND_0$ and A_0 are identical, and we can as for A_0 consider all messages arriving when the messenger leaves, as arriving $o(1)$ time later without any difference.

The competitive ratio for $RAND_\Delta$ is given by the following theorem.

Theorem 3. *The expected competitive ratio of* $RAND_\Delta$ *is*

$$\overline{\mathcal{R}}(RAND_\Delta) = \begin{cases} \frac{1}{2} + \frac{3}{2(\Delta+1)} & , \text{ if } 0 \leq \Delta \leq \frac{1}{2} \\ \frac{6\Delta^2+4\Delta+1}{4\Delta(1+\Delta)} & , \text{ if } \frac{1}{2} \leq \Delta \leq \sqrt[3]{\frac{1}{2}} \\ \frac{\Delta}{2} + 1 & , \text{ if } \sqrt[3]{\frac{1}{2}} \leq \Delta \leq 1 \end{cases}$$

The best ratio $\frac{\sqrt[3]{\frac{1}{2}}}{2} + 1 \approx 1.397$ *is achieved by* $RAND_{\sqrt[3]{\frac{1}{2}}}$.

As for A_k, the theorem is shown by constructing a worst-case input sequence. For a fixed $RAND_\Delta$, any input sequence is divided into phases almost as previously: Each phase starts with the arrival of the first message after the previous phase has ended, and ends exactly when, regardless of the random choices made, there are definitely neither any messages waiting to be sent nor is the messenger out. In the event that a new message arrives at the exact same time as the messenger returns, and where no random choices would have made the messenger arrive later, the phase ends, and the new message starts the next phase. Due to linearity of expectation, it is enough to consider a worst case phase.

Lemma 3. *Messages in a worst-case phase for $RAND_\Delta$ are not further than one apart, i.e., $\forall i : a_{i+1} - a_i \leq 1$.*

Proof. Let a_i be the first message such that $a_{i+1} > a_i + 1$. As all messages before a_i are at most one apart, OPT can send the messages $a_1, \ldots, a_i$ at time a_i, so that it does not incur any cost between time $a_i + 1$ and a_{i+1}. This means that the arrival of message a_{i+1} (together with all other messages after message a_{i+1}) can be shifted to any later time without increasing the cost of OPT.

Since message a_i leaves with the messenger at time $a_i + 1$ at the latest, message a_{i+1} arrives either when the messenger is out or when the messenger has returned (and then no other messages will be waiting at that time). The cost of $RAND_\Delta$ is maximal if the randomly chosen waiting time after a_{i+1} is not shared by time where the messenger is out, i.e., the cost is maximal if message a_{i+1} arrives after the messenger returns, and thereby if the message is not in the same phase, but in the next. □

Lemma 4. *For a worst case phase with messages $\sigma = (a_1 = 0), \ldots, a_m$, the expected competitive ratio of $RAND_\Delta$ is at most:*

$$\overline{\mathcal{R}}(RAND_\Delta) = \frac{E[RAND_\Delta(\sigma)]}{OPT(\sigma)} \leq \frac{a_m + 2}{a_m + 1} = 1 + \frac{1}{a_m + 1}$$

Proof. Follows directly from Lemma 3. □

The following lemma shows that a worst case phase with $\sigma = (a_1 = 0), \ldots, a_m$ and $a_m \leq 1$ can be assumed to contain at most two messages:

Lemma 5. *For any phase with messages $\sigma = (a_1 = 0), \ldots, a_m$ and $a_m \leq 1$, the phase obtained by looking only at the first and the last message of σ, $\sigma' = a_1, a_m$, has an expected competitive ratio which is no better, i.e.,*

$$\frac{E[RAND_\Delta(\sigma)]}{OPT(\sigma)} \leq \frac{E[RAND_\Delta(\sigma')]}{OPT(\sigma')}$$

Proof. As $a_m \leq 1$, we have $OPT(\sigma) = OPT(\sigma') = a_m + 1$. For $RAND_\Delta$, we consider two cases. Assume that the messenger leaves with the first message (and other messages if any) at time δ. If $a_m < \delta$, then the cost of σ is the same as for σ'. If $\delta \leq a_m$, then message a_m is not sent until the messenger leaves the next time. This point of time is determined by δ, Δ, and the first message, a_i, with $\delta \leq a_i$. If we leave out the messages before message a_m (but after message a_1), then on average the messenger does not leave earlier the second time. □

Furthermore, as the next lemma shows, if $a_m \leq 1$, then in addition to assuming that $m \leq 2$, we can assume that $a_m \in \{0, \Delta\}$. Note that for $\Delta > 0$, $a_2 = 0$, i.e., $\sigma = 0, 0$ has the same expected competitive ratio as $\sigma = 0$. For $\Delta = 0$, the sequence $\sigma = 0, 0$ is the same as $\sigma = 0, \Delta$.

Lemma 6. *A worst case input sequence for $RAND_\Delta$ with two messages, $\sigma = (a_1 = 0), a_2$, where $a_2 \leq 1$, must have $a_2 \in \{0, \Delta\}$. This gives the following lower bounds, of which at least one is an upper bound for all input sequences $\sigma = a_1, \ldots, a_m$, where $a_m - a_1 \leq 1$:*

$\sigma = 0$ has an expected competitive ratio of

$$1 + \frac{\Delta}{2}.$$

$\sigma = 0, \Delta$ has an expected competitive ratio of c, where

$$c = \begin{cases} \frac{1}{2} + \frac{3}{2(\Delta+1)} & , \text{ if } \Delta \leq \frac{1}{2} \\ \frac{6\Delta^2+4\Delta+1}{4\Delta(\Delta+1)} & , \text{ if } \Delta > \frac{1}{2} \end{cases}$$

Proof. For $\sigma = 0$, the expected competitive ratio is $\frac{E[RAND_\Delta(\sigma)]}{OPT(\sigma)} = 1 + \frac{\Delta}{2}$.

For $\sigma = 0, a_2$, we prove the result by case analysis.

Let δ be the (now fixed) randomly chosen time at which $RAND_\Delta$ sends the first message (and the next, if $a_2 < \delta$). The expected cost of $RAND_\Delta$ is:

$$k(a_2, \delta) = \delta + 1 + \begin{cases} 0 & , \text{ if } a_2 \leq \delta \\ 1 & , \text{ if } \delta < a_2 \leq \delta + 1 - \Delta \\ 1 + \frac{(a_2+\Delta)-(\delta+1)}{2} & , \text{ if } \delta + 1 - \Delta < a_2 \leq 1 \end{cases}$$

This can be rephrased to a form which is more convenient in the proof:

$$k(a_2, \delta) = \delta + 1 + \begin{cases} 1 + \frac{(a_2+\Delta)-(\delta+1)}{2} & , \text{ if } 0 \leq \delta < a_2 + \Delta - 1 \\ 1 & , \text{ if } a_2 + \Delta - 1 \leq \delta < a_2 \\ 0 & , \text{ if } a_2 \leq \delta \leq \Delta \end{cases} \quad (1)$$

The expected cost of $RAND_\Delta$ is $E[RAND_\Delta(\sigma)] = \frac{1}{\Delta}\int_0^\Delta k(a_2, \delta)d\delta$, whereas $OPT(\sigma) = a_2 + 1$, giving an expected competitive ratio of $c(a_2) = \int_0^\Delta \frac{k(a_2,\delta)}{\Delta(a_2+1)}d\delta$.

Let us first consider the case when $\Delta \leq \frac{1}{2}$. If $0 \leq a_2 \leq \Delta$, then the first case of $k(a_2, \delta)$ in equation (1) becomes empty, since $a_2 + \Delta - 1 \leq 0$. Thus, the expected competitive ratio is

$$c(a_2) = \frac{\int_0^{a_2}(\delta+2)d\delta + \int_{a_2}^\Delta(\delta+1)d\delta}{\Delta(1+a_2)} = \frac{\frac{\Delta^2}{2} + \Delta + a_2}{\Delta(1+a_2)} = \frac{1}{\Delta} + \frac{\frac{\Delta}{2} + 1 - \frac{1}{\Delta}}{1+a_2}$$

Since $\Delta \leq \frac{1}{2}$, this is easily seen to be maximal for a_2 as small as possible, i.e., $a_2 = 0$, and $c(0) = \frac{\Delta}{2} + 1$. Let us then consider the case when $\Delta < a_2 \leq 1 - \Delta$:

$$c(a_2) = \frac{\int_0^\Delta(\delta+2)d\delta}{\Delta(1+a_2)} = \frac{\frac{\Delta}{2} + 2}{1+a_2}$$

This is again maximal for a_2 as small as possible, i.e., it is at most $c(\Delta) = \frac{1}{2} + \frac{3}{2(1+\Delta)}$. The last case is when $1 - \Delta \leq a_2 \leq 1$:

$$c(a_2) = \frac{\int_0^{\Delta}(\delta+2)d\delta + \int_0^{a_2+\Delta-1} \frac{a_2+\Delta-1-\delta}{2} d\delta}{\Delta(1+a_2)} = \frac{\frac{\Delta}{2} + 2 + \frac{(a_2+\Delta-1)^2}{4\Delta}}{1+a_2}$$

This is maximal for a_2 as large as possible. We have $c(1) = \frac{3}{8}\Delta + 1$, which is, however, smaller than the results found previously.

The case when $\Delta > \frac{1}{2}$ is similar [5], but omitted due to space restrictions. □

Lemma 7. *Let $\sigma = (a_1 = 0), \ldots, a_m$ be any worst-case phase for $RAND_\Delta$ with $\Delta > \frac{1}{2}$ and $1 < a_m \leq 1 + \Delta$, then*

$$\frac{E[RAND_\Delta(\sigma)]}{OPT(\sigma)} \leq \frac{\Delta^2 + 2\Delta + a_m - 1}{\Delta(a_m + 1)}$$

Proof. As before, let δ be the now fixed randomly chosen time at which $RAND_\Delta$ sends message a_1 (and all other messages arriving no later than time δ, if any). The worst-case cost of $RAND_\Delta$ can be described as follows:

$$w(\delta) \leq \begin{cases} a_m + 2 & , \text{ if } \delta + 1 < a_m \\ a_m + \Delta + 1 & , \text{ if } a_m < \delta + 1 \end{cases}$$

This gives an expected worst-case cost for $RAND_\Delta$ of at most $\frac{1}{\Delta}\int_0^{\Delta} w(\delta)d\delta = \frac{\Delta^2+2\Delta+a_m-1}{\Delta}$, whereas $OPT(\sigma) = a_m + 1$, since σ is a worst-case phase. □

Proof (Proof of Theorem 3). Lemma 6 states lower bounds for $RAND_\Delta$ which are the best possible for phases $\sigma = (a_1 = 0), \ldots, a_m$ with $a_m \leq 1$. By Lemma 4, if $a_m > 1$, a worst case input sequence has a competitive ratio less than $\frac{3}{2}$.

For $\Delta \leq \frac{1}{2}$, Lemma 6 is enough, since the input sequence $0, \Delta$ has an expected competitive ratio of $\frac{1}{2} + \frac{3}{2(\Delta+1)} \geq \frac{3}{2}$.

For $\Delta > \frac{1}{2}$, by Lemma 4, any input sequence with $a_m > 1 + \Delta$ has an expected competitive ratio of at most $\frac{(1+\Delta)+2}{(1+\Delta)+1}$. Lemma 7 gives a similar bound on the expected competitive ratio for $a_m \in (1, 1 + \Delta]$. It is easily shown that

$$\max\left\{\frac{\Delta+3}{\Delta+2}, \frac{\Delta^2+2\Delta+a_m-1}{\Delta(a_m+1)}\right\} \leq \max\left\{1 + \frac{\Delta}{2}, \frac{6\Delta^2+4\Delta+1}{4\Delta(\Delta+1)}\right\}.$$

So, either 0 or $0, \Delta$ is a worst case sequence, and Lemma 6 gives the result. □

5 Concluding Remarks

We have considered a new cost function instead of the cost function which is almost a standard in theoretical analysis of this type of problems, namely flow-time. With the new cost function, algorithms can be distinguished effectively, whereas using flow-time, this is not possible while respecting the LogP model

assumptions. The behavior of the optimal off-line algorithm can be a little peculiar, however. If we consider sequences where n messages arrive less than one unit apart, nothing in our cost function encourages the optimal off-line algorithm to send any messages until the nth message has arrived.

While the behavior of an off-line optimal algorithm is secondary to the ability of the total set-up to distinguish between good and bad on-line algorithms, our results are robust enough that the behavior of OPT could be altered. Assume that we change the cost function such that when a message has been waiting for one time unit (or equivalently, has not been delivered two units after it became available), a strictly higher penalty is imposed. This will encourage a different behavior, where messages are sent earlier. However, OPT can still send all messages with the same cost. It will send at time t_n immediately after the nth message has arrived (as before), but it could also send at all times in the set $\{t_n - i | i \in \mathbb{N}, t_n - i \geq 0\}$.

The cost of all reasonable on-line algorithms as defined in the introduction, including the cost of A_k for $k \leq 1$ and $RAND_\Delta$ for $\Delta \leq 1$, will also be unchanged since none of them will wait more than one time unit before sending. Thus, our results hold for this more general type of cost function.

Finally, our algorithms can in principle be built into any operating system, though the ease with which this can be done depends on the exact design of the operating system in question, in particular on the availability of an extra timer to support interrupts from our algorithm. We have concrete plans to try out one of our algorithms by building it into the new Occam Operating System [1].

Acknowledgments. We thank Brian Vinter for drawing our attention to the Packet Bundling Problem and for initial discussions regarding the cost function.

References

1. Fred Barnes, Brian Vinter, and Peter H. Welch. The Occam Operating System project. Work in progress.
2. Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.
3. David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.
4. Daniel R. Dooly, Sally A. Goldman, and Stephen D. Scott. TCP dynamic acknowledgment delay: Theory and practice. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 389–398. ACM Press, 1998.
5. Jens S. Frederiksen and Kim S. Larsen. Packet bundling. Technical Report 9, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, 2002.
6. Anna R. Karlin, Claire Kenyon, and Dana Randall. Dynamic TCP acknowledgement and other stories about $e/(e-1)$. In *Proceedings of the 33th Annual ACM Symposium on Theory of Computing*, pages 502–509. ACM Press, 2001.

Algorithms for the Multi-constrained Routing Problem

Anuj Puri[1] and Stavros Tripakis[2]

[1] EECS Department, University of California at Berkeley, CA 94720.
anuj@eecs.berkeley.edu.
[2] VERIMAG, Centre Equation, 2, ave de Vignate, 38610 GIERES, France.
tripakis@imag.fr, www-verimag.imag.fr.

Abstract. We study the problem of routing under multiple constraints. We consider a graph where each edge is labeled with a cost and a delay. We then consider the problem of finding a path from a source vertex to a destination vertex such that the sum of the costs on the path satisfy the cost constraint and the sum of the delays satisfy the delay constraint. We present three different algorithms for solving the problem. These algorithms have varying levels of complexity and solve the problem with varying degrees of accuracy. We present an implementation of these algorithms and discuss their performance on different graphs.

1 Introduction

Finding paths in a graph with respect to multiple criteria is a fundamental problem, with applications in many areas, such as telecommunications (finding routes with respect to various quality-of-service criteria — delay, cost, packet loss, etc), or mission planning (finding inexpensive routes in a terrain, while avoiding unsafe regions).

In this paper, we reconsider the problem of finding paths in a multi-weight graph, satisfying multiple constraints. We simplify our presentation by considering only two-weight graphs (we call the weights *delay* and *cost*), and we discuss extensions to more than two weights in the conclusion. Our objective is to find a path from a given source node to a given destination node, such that the sum of all delays on the path is less than a given D, and the sum of all costs is less than a given C. Solving this problem exactly is well known to be NP-Complete [4,6,5,8]. An alternative version of the problem is to find a path minimizing the delay, while keeping the cost bounded by C. The two versions are equivalent, in the sense that an algorithm to solve one version can be used to solve the other version, without significant increase in complexity.

We provide various algorithms for solving the problem. The first algorithm is based on a Bellman-Ford type of iteration and comes in two versions: (1) exact, with complexity $O(|V||E|min\{C, D\})$, where V and E are the sets of vertices and edges in the graph, and (2) approximative with bounded-error $\epsilon \in [0, 1]$ (a user input) and complexity $O(|V|^2|E|(1 + \frac{1}{\epsilon}))$. Error ϵ means that the path

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 338–347, 2002.

found has cost at most $C \cdot (1+\epsilon)$ and delay at most $D \cdot (1+\epsilon)$. The second algorithm is based on a *Lagrange relaxation* technique and solves iteratively a series of standard shortest-path problems, until no improvement in the path can be achieved. In this case, the error is at most 1 (we provide an example exhibiting this behavior), in practice, however, we obtain paths very close to optimal ones.

Unfortunately, due to space limitations, we cannot discuss related work. Such a discussion can be found in the full version of the paper, available from the home pages of the authors.

2 Problem Formulation

We consider a directed *two-weight* graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. An edge $e \in E$ is $e = (v, w, c, d)$ where the edge goes from v to w, and has delay $\mathsf{delay}(e) = d$ and $\mathsf{cost}(e) = c$. We write this as $v \xrightarrow{(c,d)} w$. When there is no confusion, we may also write the edge as (v, w) and say the edge is labeled with (c, d).

A path is $p = v_1 \xrightarrow{(c_1,d_1)} v_2 \xrightarrow{(c_2,d_2)} v_3 \xrightarrow{(c_3,d_3)} \cdots \xrightarrow{(c_n,d_n)} v_{n+1}$. The cost of a path is $\mathsf{cost}(p) = \sum_{i=1}^{n} c_i$ and its delay is $\mathsf{delay}(p) = \sum_{i=1}^{n} d_i$.

Given a path p and cost constraint $C \geq 1$ and delay constraint $D \geq 1$, we say p is *feasible* provided $\mathsf{cost}(p) \leq C$ and $\mathsf{delay}(p) \leq D$. The problem of *routing under two constraints* is, given $G = (V, E)$, cost constraint C and delay constraint D, a *source* node $s \in V$ and a *destination* node $t \in V$, find a feasible path p from s to t, or decide that no such path exists.

Example. Consider the two-weight graph of Figure 1. Each edge is labeled with (c, d) where c is the cost of the edge and d is the delay of the edge. For example, the edge from vertex 1 to vertex 2 has cost 3 and delay 1. Suppose the source vertex is 1, the destination vertex is 4, the cost constraint is $C = 5$ and the delay constraint is $D = 2$. Then, the path $1 \xrightarrow{(3,1)} 2 \xrightarrow{(2,1)} 4$ is feasible, whereas $1 \xrightarrow{(1,2)} 3 \xrightarrow{(1,2)} 4$ is not (since it violates the delay constraint). The reader can check that if $C = 4$ and $D = 3$, then there is no feasible path.

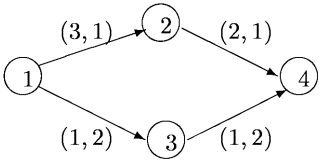


Fig. 1. A Simple Network

Rather than checking to see if a graph has a feasible path, it is sometimes useful to try to minimize the following objective function

$$M(p) = max\{\frac{max\{\mathsf{cost}(p), C\}}{C}, \frac{max\{\mathsf{delay}(p), D\}}{D}\}.$$

Observe that for any path p, $M(p) \geq 1$ and $M(p) = 1$ iff p is feasible. But even if a feasible path does not exist or is hard to find, by trying to minimizing $M(p)$

we can get a path that comes "close" to satisfying the constraints. Formally, we define the error of a path p as

$$\mathsf{error}(p) = \frac{M(p) - M(p^*)}{M(p^*)}$$

where p^* is a path which minimizes M. Notice that $\mathsf{error}(p) \geq 0$ and $\mathsf{error}(p) = 0$ iff p is feasible. Also note that if $\mathsf{cost}(p) \leq C \cdot (1+\epsilon)$ and $\mathsf{delay}(p) \leq D \cdot (1+\epsilon)$ then $\mathsf{error}(p) \leq \epsilon$. Indeed, the two above conditions imply that $M(p) \leq 1+\epsilon$ and, since $M(p^*) \geq 1$, we get $\mathsf{error}(p) \leq \epsilon$.

3 Complexity

The multiple-constraint routing problem is known to be NP-complete. For completeness of the paper, we provide a proof here as well.

Theorem 1. *The routing problem with two constraints is NP-Complete.*

Proof. We will provide a reduction from the knapsack problem. Recall that in the knapsack problem, we are given positive integers $c_1, c_2, \ldots, c_n$, and N, and the objective is to find a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} c_i = N$.

From the knapsack problem, we construct a graph with vertices $\{1, \ldots, n\}$. There are two edges from vertex i to vertex $i+1$: edge $(i, i+1, c_i, 0)$ and edge $(i, i+1, 0, c_i)$. Figure 2 shows the scenario. Our objective is to find a path from vertex 1 to vertex n with cost constraint N and delay constraint $\sum_{i=1}^{n} c_i - N$. It is easy to check that there is a path that satisfies the constraints iff there is a solution to the knapsack problem.

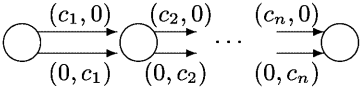


Fig. 2. Graph obtained for the knapsack problem

4 Two Pseudo-Polynomial Algorithms

In this section, we propose two algorithms for the problem of routing under two constraints. Both algorithms are extensions of the standard Bellman-Ford algorithm. The first algorithm is exact and has worst-case complexity $O(|V| \cdot |E| \cdot min\{C, D\})$. The second algorithm is approximative and has worst-case complexity $O(|V|^2 \cdot |E| \cdot (1 + \frac{1}{\epsilon}))$, where $\epsilon \in [0, 1]$ is a parameter set by the user.[1] The second algorithm is approximative in the sense that, it might not yield a feasible path, even if such a path exists. However, the error in a path p returned by the algorithm is bounded: $\mathsf{error}(p) \leq \epsilon$. Notice that the approximative

[1] In fact, the exact algorithm can be obtained simply by setting $\epsilon = 0$ in the approximative algorithm.

algorithm is worth using only when $|V|$ is (much) smaller than $\frac{\epsilon}{1+\epsilon} \cdot min\{C, D\}$. Otherwise, the exact algorithm, being less expensive, is preferable.

We begin by making a few assumptions, without loss of generality. Given a two-weight graph $G = (V, E)$, where $|V| = n$, let $\mathsf{cost}_{max} = max\{c \mid (\_, \_, c, \_) \in E\}$ and $\mathsf{delay}_{max} = max\{d \mid (\_, \_, d, \_) \in E\}$. Then, we can assume that $n \cdot \mathsf{cost}_{max} > C$ and $n \cdot \mathsf{delay}_{max} > D$. Otherwise, any (acyclic) path trivially satisfies one of the constraints, and it remains to find a path satisfying the other constrain too, which can be done using a standard shortest-path algorithm. We will also assume that the greatest common divisor of $\{C, \mathsf{cost}(e) \mid e \in E\}$ is 1, and similarly for the delays (otherwise we could just divide all costs/delays by their greatest common divisor, without affecting the problem).

Our algorithm works as follows. For each vertex w, we compute a set of *cost-delay pairs* F_w. Each $(c, d) \in F_w$ represents a possible path p from w to the destination vertex v, where $\mathsf{cost}(p) = c$ and $\mathsf{delay}(p) = d$. To keep the size of F_w manageable, we eliminate from F_w all elements corresponding to infeasible paths (i.e., all (c, d) such that $c > C$ or $d > D$). Moreover, we eliminate from F_w all redundant elements, that is, all elements with both cost and delay greater from some other element.

Cost-delay sets. A *cost-delay set* for a vertex w is a set $F_w \subseteq \mathsf{N} \times \mathsf{N}$. An element (c, d) of F_w is called *infeasible* if either $c > C$ or $d > D$. An element (c, d) of F_w is called *redundant* if there exists a different $(c', d') \in F_w$ such that $c' \leq c$ and $d' \leq d$. A cost-delay set F is said to be *minimal* if it contains no infeasible or redundant elements. It can be checked that if F is minimal, then $|F| \leq min\{C, D\}$. Also, to every cost-delay set F corresponds a unique greatest minimal subset $F' \subseteq F$. We write $\mathsf{minimal}(F)$ to denote the greatest minimal subset of F. Figure 3 displays the typical structure of a cost-delay set and its minimal. Black and grey bullets are infeasible and redundant elements, respectively.

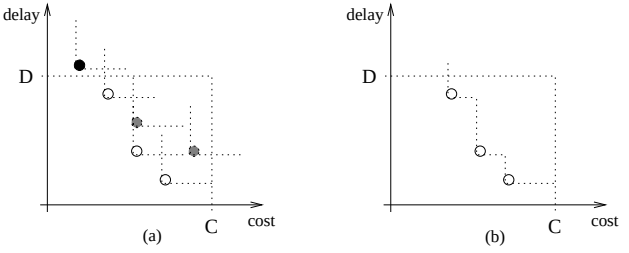


Fig. 3. A cost-delay set (a) and its minimal (b)

Minimal cost-delay sets admit an efficient canonical representation as sorted lists. Consider a minimal set $F = \{(c_1, d_1), (c_2, d_2), ..., (c_n, d_n)\}$ and assume, without loss of generality, that $c_1 \leq c_2 \leq \cdots \leq c_n$. Then, $d_1 \geq d_2 \geq \cdots \geq d_n$ must hold, otherwise there would be at least one redundant element in F.

Consequently, F can be represented as the list $(c_1, d_1)\ (c_2, d_2)\ \cdots\ (c_n, d_n)$, sorted using cost as the "key". This representation is canonical in the sense that two minimal sets F_1, F_2 are equal iff their list representations are identical.

Given minimal (i.e., feasible and non-redundant) F_1, F_2, the union $F_1 \cup F_2$ is always feasible, but not necessarily non-redundant. In order to compute $F = \mathsf{minimal}(F_1 \cup F_2)$ directly from the list representations L_1, L_2 of F_1, F_2, we can use a simple modification of a usual merge-sort algorithm on lists. The latter takes as input L_1, L_2 and produces L, the list representation of F. In order to guarantee the absence of redundant points in L, it compares at each step the heads (c_1, d_1) and (c_2, d_2) of (the remaining parts of) L_1, L_2. If $c_1 \leq c_2$ and $d_1 \leq d_2$ then (c_2, d_2) is redundant and is skipped. If $c_2 \leq c_1$ and $d_2 \leq d_1$ then (c_1, d_1) is skipped. Otherwise, the pair with the smallest c_i is inserted in L and the head pointer move one element ahead in the corresponding list L_i. It is easy to see that this algorithm is correct. The cost of the algorithm is $n_1 + n_2$, where n_i is the length of L_i. Therefore, the worst-case complexity of computing the union of cost-delay sets is $O(min\{C, D\})$.

Translation is defined on a cost-delay set F and a pair $(c, d) \in \mathsf{N}^2$:

$$F + (c, d) \stackrel{\text{def}}{=} \{(c' + c, d' + d) | (c', d') \in F\}$$

If F is minimal, then $F + (c, d)$ is non-redundant, however, it may contain infeasible points. These can be easily eliminated, however, while building the list L' for $min(F + (c, d))$: the list of F is traversed, adding (c, d) to each of its elements, (c_i, d_i); if $c_i + c \leq D$ and $d_i + d \leq D$ then $(c_i + c, d_i + d)$ is inserted at the end of L', otherwise it is infeasible and it is skipped. At the end, L' will be sorted by cost. The complexity of translation is $O(min\{C, D\})$.

4.1 The Exact Algorithm

For the sake of clarity, we first present the exact algorithm (although it is a special case of the approximative algorithm). The algorithm iteratively computes the (minimal) cost-delay sets of all vertices in the graph. Let F_w^j denote the cost-delay set for vertex w at iteration j. Initially, all vertices have empty cost-delay sets, $F_w^0 = \emptyset$, except v, for which $F_v^0 = \{(0, 0)\}$. At each iteration, each vertex updates its cost-delay set with respect to all its successor vertices. Computation stops when no cost-delay set is updated any more.

Let $w_1, ..., w_k$ be the successor vertices of w, that is, $w \stackrel{(c_i, d_i)}{\longrightarrow} w_i$, for $i = 1, ..., k$ (note that $w_1, ..., w_k$ might not be distinct). Then, the cost-delay set of w at iteration $j + 1$ will be:

$$F_w^{j+1} = \mathsf{minimal}\Big(F_w^j \cup \bigcup_{i=1}^{k} \big(F_{w_i}^j + (c_i, d_i)\big)\Big) \tag{1}$$

Proposition 2. *(Termination) The updating of the cost-delay sets will stabilize after at most $|V|$ iterations, that is, for any vertex w, $F_w^{|V|+1} = F_w^{|V|}$. (Correctness) A feasible path from w to v exists iff $F_w^{|V|} \neq \emptyset$. For any $(c,d) \in F_w^{|V|}$, there exists a path p from w to v such that $\mathsf{cost}(p) = c$ and $\mathsf{delay}(p) = d$.*

Worst-case complexity. Proposition 2 implies that the algorithm stops after at most $|V|$ iterations. At each iteration, the cost-delay set of each vertex is updated with respect to all its successor vertices. Thus, there are at most $|E|$ updates at each iteration. Each update involves a translation and a union, both of which have complexity $O(min\{C, D\})$. Therefore, the overall worst-case complexity of the algorithm is $O(|V| \cdot |E| \cdot min\{C, D\})$.

4.2 The Bounded-Error Approximative Algorithm

The approximative algorithm is similar to the one of section 4, with the additional fact that it eliminates elements of cost-delay sets which are "too close" to some other element. More formally, for $(c_1, d_1), (c_2, d_2) \in \mathsf{N}^2$, define:

$$||(c_1, d_1), (c_2, d_2)|| \stackrel{\text{def}}{=} max\{|c_1 - c_2|, |d_1 - d_2|\}$$

Then, a cost-delay set F is said to have *minimal distance* δ iff for all distinct $(c_1, d_1), (c_2, d_2) \in F$, $||(c_1, d_1), (c_2, d_2)|| \geq \delta$.

Given a cost-delay set F and some δ, we want to find a subset $F' \subseteq F$, such that: (1) F' has minimal distance δ, and (2) for all $x \in F - F'$, there exists $y \in F'$ such that $||x, y|| < \delta$. In general, there may be more than one subsets of F satisfying the above conditions. We want to find one of them. We use the following procedure. Assume $L = (x_1, ..., x_n)$ is the list representation of F. The procedure produces L', the list representation for F'. Initially, $L' = (x_1)$. Let y denote the last element of L', at each point during the execution of the procedure. For each $i \geq 2$, if $||x_i, y|| \geq \delta$ then x_i is appended at the end of L' and y is updated to x_i, otherwise, x_i is skipped. It can be shown that the list built that way represents a legal δ-distance subset of F. From now on, we denote this set by $\mathsf{min\_dist}(\delta, F)$.

The approximative algorithm is defined by modifying Equation (1) as follows:

$$B_w^{j+1} = \mathsf{min\_dist}\Big(\delta_\epsilon, \mathsf{minimal}\Big(B_w^j \cup \bigcup_{i=1}^{n} \big(B_{w_i}^j + (c_i, d_i)\big)\Big)\Big) \tag{2}$$

where $\delta_\epsilon = \frac{min\{C,D\} \cdot \epsilon}{|V|}$, and in the definition of $\mathsf{minimal}$, the feasibility region is extended by $(\epsilon \cdot C, \epsilon \cdot D)$, that is, we require that for all $(c,d) \in B_w$, $c \leq (1+\epsilon) \cdot C$ and $d \leq (1+\epsilon) \cdot D$.

Proposition 3. *The approximative algorithm terminates after at most $|V|$ steps, that is, $B_w^{|V|+1} = B_w^{|V|}$. If $B_u = \emptyset$ at the end of the approximative algorithm, then no feasible path from u to v exists. Otherwise, for each $(c,d) \in B_u$, there exists a path p from u to v such that $\mathsf{cost}(p) = c$, $\mathsf{delay}(p) = d$ and $\mathsf{error}(p) \leq \epsilon$.*

Worst-case complexity. The only difference from the exact algorithm is in the size of the cost-delay sets B_w. Since the latter have minimal distance δ_ϵ and are bounded by the feasibility region $((1+\epsilon)\cdot C, (1+\epsilon)\cdot D)$, we have $|B_w| \leq \frac{(1+\epsilon)\cdot min\{C,D\}}{\delta_\epsilon}$. By definition of δ_ϵ, we get $|B_w| \leq \frac{(1+\epsilon)}{\epsilon}|V|$. This yields an overall worst-case complexity of $O(|V|^2 \cdot |E| \cdot (1+\frac{1}{\epsilon}))$.

5 An Iterative Shortest Path Algorithm

In this section, we present an algorithm which is based on a *Lagrange-relaxation* and solves iteratively a sequence of standard shortest path problems. The objective is to find a path minimizing $M(\cdot)$.

For this section, we assume that we have normalized the costs and delays by dividing the costs by C and the delays by D. A path is feasible in the new graph if $\mathsf{cost}(p) \leq 1$ and $\mathsf{delay}(p) \leq 1$. Note that a path is feasible in the new graph iff it was feasible in the original graph. For the new graph, $C = D = 1$, thus, $M(p)$ is not modified by the normalization.

To find a path satisfying two constraints by using the shortest path algorithm, we choose an $0 \leq \alpha \leq 1$ and replace the cost c and the delay d associated with an edge with the weight $\alpha \cdot c + (1-\alpha)\cdot d$. We then use a standard shortest path algorithm, say, Dijkstra's algorithm, to find a path with the smallest weight. We refer to this path as $SP(G,\alpha)$. As the next lemma shows, $p = SP(G,\alpha)$ has an error $\mathsf{error}(p)$ of at most 1 for $\alpha = \frac{1}{2}$.

Lemma 4. *For a graph $G = (V,E)$, $M(p^*) \leq M(p) \leq 2M(p^*)$, where $p = SP(G,\alpha)$ and p^* is a path minimizing M.*

Proof. Recall that for all paths p', $M(p') \geq 1$. If $M(p) = 1$, then clearly $M(p) \leq 2M(p^*)$. So assume $M(p) > 1$. Then $M(p) \leq \mathsf{cost}(p) + \mathsf{delay}(p) \leq \mathsf{cost}(p^*) + \mathsf{delay}(p^*) \leq 1 + 1 \leq M(p^*) + M(p^*) = 2M(p^*)$.

The previous lemma shows that by initializing $\alpha = \frac{1}{2}$, we obtain a path p with $\mathsf{error}(p) \leq 1$. By modifying α appropriately, the algorithm can reduce the error. This is done using *binary search*: suppose we know that the optimal value of α lies in the interval $[l,u]$ (initially $[0,1]$); we find $p = SP(G,\alpha)$ for $\alpha = \frac{l+u}{2}$; if $\mathsf{cost}(p) \leq \mathsf{delay}(p)$, we eliminate the interval $(\frac{l+u}{2}, u]$ from consideration, otherwise, we eliminate $[l, \frac{l+u}{2})$. The algorithm terminates when no new paths are found (this will eventually happen, since the number of paths is finite). The reason that half of the interval can be eliminated follows from the following result.

Proposition 5. *Let $p = SP(G,\alpha)$ and $p' = SP(G,\alpha')$. (1) If $\alpha < \alpha'$ then $\mathsf{cost}(p') \leq \mathsf{cost}(p)$ and $\mathsf{delay}(p') \geq \mathsf{delay}(p)$. (2) If $\alpha > \alpha'$ then $\mathsf{cost}(p') \geq \mathsf{cost}(p)$ and $\mathsf{delay}(p') \leq \mathsf{delay}(p)$.*

Now, suppose the algorithm finds $p = SP(G,\alpha)$ with $\mathsf{cost}(p) > C$. Then, by increasing α, we get a new path with smaller cost. Similarly, if $\mathsf{delay}(p) > D$, we decrease α in order to decrease the delay. What if both $\mathsf{cost}(p) > C$ and

$\mathsf{delay}(p) > D$? Then we know that no feasible path exists. Indeed, if there was a feasible path p^*, then $\mathsf{cost}(p^*) \leq C < \mathsf{cost}(p)$ and $\mathsf{delay}(p^*) \leq D < \mathsf{delay}(p)$, thus, $\alpha \cdot \mathsf{cost}(p^*) + (1-\alpha) \cdot \mathsf{delay}(p^*) < \alpha \cdot \mathsf{cost}(p) + (1-\alpha) \cdot \mathsf{delay}(p)$, which contradicts the fact that p is optimal w.r.t. α.

Example. Notice that there are examples where the error can be arbitrarily close to 1. For instance, consider the example in Figure 4 where the cost constraint is $C = 1$ and the delay constraint is $D = 1$. It is easy to check that for any $0 \leq \alpha \leq 1$, $\mathsf{error}(SP(G, \alpha)) = 1 - \epsilon$.

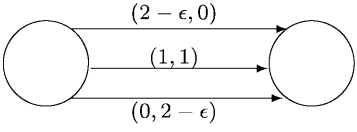


Fig. 4. A graph where the error is $1-\epsilon$.

6 Experimental Results

We have implemented our algorithms in a prototype tool written in C. The tool takes as input a multi-weight graph, a source node, a destination node, a weight to be minimized, and upper bounds on the rest of the weights. The tool is asked to run either the bounded-error approximative algorithm (if so, a step-error must be provided), BEA for short, or the iterative shortest-path algorithm, ISP for short. We next report on experiments using the tool on graphs obtained by translating elevation maps of physical landscapes into multi-weight graphs. [2] Tables 5 and 6 present the results. The notation is as follows: n is the number of nodes, m is the number of edges, t is the CPU time in seconds, c is the "cost" weight of a path found, d_1, d_2 are the two "delay" weights of a path, δ_ϵ is the step-error for BEA, "s/d" stands for "source/destination pair". Three different experiments are shown for BEA (varying δ_ϵ) and two different experiments for ISP (varying the source/destination pair). A example route is shown in Figure 7.

From Tables 5 and 6, the following observations can be made: (1) ISP is two or more orders of magnitude faster than BEA, while at the same time producing paths which are both feasible (w.r.t. d_1) and as good as the paths produced by BEA (w.r.t. c). (2) BEA is sensitive to the step-error parameter, δ_ϵ. Reducing δ_ϵ by one or two orders of magnitude resulted in dramatic increases in CPU time. (3) The algorithms are not very sensitive to changes in source/destination.

From Figure 6, we see that adding one more weight/constraint to the problem dramatically increases the execution time of BEA. Whereas we have been able to execute the algorithm in 2-weight graphs of size up to 225680 vertices, we could only treat 3-weight graphs of relatively small sizes (up to 1700 vertices) in reasonable time.

[2] A map is a two-dimensional array, its i, j-th element giving the altitude of the point with longitude i and latitude j. A landscape of dimension $n_1 \times n_2$ results in a graph with $n_1 \cdot n_2$ vertices and approximately $4 \cdot n_1 \cdot n_2$ edges (central vertices having four successors, "north, south, east, west"). The cost c of an edge is taken to be the difference in elevation between the destination and source vertices. The "delays" d_1 and d_2 (the second delay was used only in 3-weight graphs) are generated randomly according to a Gaussian distribution.

	BEA			ISP	
	$\delta_\epsilon = 10^{-4}$	$\delta_\epsilon = 10^{-5}$	$\delta_\epsilon = 10^{-6}$	1st s/d	2nd s/d
graph 1	$t = 70$	$t = 413$	$t = 1274$	$t = 0.94$	$t = 0.48$
$n = 4641$	$c = 1532$			$c = 1564$	$c = 723$
$m \approx 4 \cdot n$	$d_1 = 2.15\%$			$d_1 = 1.27\%$	$d_1 = 0.14\%$
graph 2	$t = 56$	$t = 1278$	$t = 16740$	$t = 7.16$	$t = 3.48$
$n = 36417$	$c = 4152$			$c = 4220$	$c = 1184$
$m \approx 4 \cdot n$	$d_1 = 0.2\%$	$d_1 = 0.15\%$		$d_1 = 0.04\%$	$d_1 = 0\%$
graph 3	$t = 725$	$t = 3503$	$t = 9971$	$t = 47.96$	$t = 31.56$
$n = 225680$	$c = 9409$			$c = 9411$	$c = 4487$
$m \approx 4 \cdot n$	$d_1 = 0.01\%$	$d_1 = 0\%$		$d_1 = 0\%$	$d_1 = 0\%$

Fig. 5. Experimental results of the bounded-error approximative algorithm (BEA) and the iterative shortest-path algorithm (ISP) on two-weight graphs.

	$\delta_\epsilon = 10^{-4}$	$\delta_\epsilon = 10^{-5}$	$\delta_\epsilon = 10^{-6}$
graph 4	$t = 1.49$	$t = 1.80$	$t = 1.85$
$n = 777$	$c = 720$		
$m \approx 4 \cdot n$	$d_1 = 5.6\%$ $d_2 = 6.43\%$		
graph 5	$t = 4.25$	$t = 153.97$	$t = 6095.42$
$n = 1178$	$c = 994$		
$m \approx 4 \cdot n$	$d_1 = 1.79\%$ $d_2 = 5.39\%$		
graph 6	$t = 1352.54$	$t = 17631.51$	$t = 29274.12$
$n = 1722$	$c = 1024$		
$m \approx 4 \cdot n$	$d_1 = 3.68\%$ $d_2 = 4.66\%$		

Fig. 6. Experimental results of BEA on 3-weight graphs.

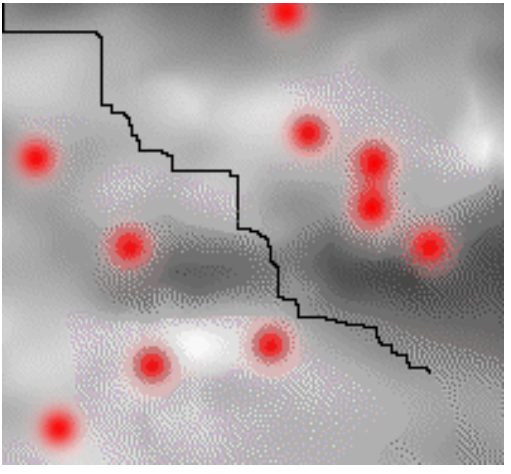


Fig. 7. An output of the bounded-error approximative algorithm on a map translated into a 2-weight graph: the solid black line depicts the path produced by the algorithm; (red) dots are "high-delay" zones; the grey scale background represents the elevation variations of the landscape (white: high, black: low).

7 Conclusion

We have presented three algorithms for solving the routing problem with multiple constraints. These algorithms vary in their complexity and the accuracy of their solutions. Our contributions mainly consist in improvements in the complexity of previously existing algorithms. We have also implemented our algorithms and examined their performance on different graphs of relatively large size.

In the general case of k-weight graphs, each edge is labeled with a k-tuple $(c_1, c_2, \ldots, c_k)$, and there is one upper bound C_i for each of the weights (or, alternatively, one of the weights must be minimized while keeping the others bounded). The Bellman-Ford-type algorithms of Section 4 can be extended in a straight-forward way to the general case: the cost-delay sets now become general *Pareto* sets of dimension k. The complexity of the exact algorithm becomes $O(|V||E| \prod_{i=1}^{k} C_i)$ and the complexity of the bounded-error approximative algorithm becomes $O(|V|^k|E|(1+\frac{1}{\epsilon})^k)$. It is also possible to extend some parts of the algorithm in Section 5. It is possible to obtain a path with error $\epsilon = k - 1$ for a problem with k constraints by solving the shortest path algorithm. But it is not clear how to extend the algorithm which iterates over shortest path problems to the case with more than two constraints.

References

1. S. Chen and K. Nahrstedt. On finding multi-constrained paths. In *International Conference on Communications (ICC'98)*, June 1998.
2. S. Chen and K. Nahrstedt. An overview of quality-of-service routing for the next generation high-speed networks: Problems and solutions. *IEEE Network, Special Issue on Transmission and Distribution of Digital Video*, 1998.
3. H. F. Salama et. al. A distributed algorithm for delay-constrained unicast routing. In *IEEE INFOCOM'97*, Kobe, Japan,, April 1997.
4. M. Garey and D. Johnson. *Computers and Intractability: a guide to the theory of NP-completeness*. Freeman, 1979.
5. R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17(1), February 1992.
6. J. M. Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14:95–116, 1984.
7. A. Orda. Routing with end-to-end qos guarantees in broadband networks. *IEEE/ACM Transactions on Networking*, June 1999.
8. Z. Wang and J. Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7), September 1996.

Computing the Threshold for q-Gram Filters

Juha Kärkkäinen*

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
juha@mpi-sb.mpg.de

Abstract. A popular and much studied class of filters for approximate string matching is based on finding common q-grams, substrings of length q, between the pattern and the text. A variation of the basic idea uses *gapped* q-grams and has been recently shown to provide significant improvements in practice. A major difficulty with gapped q-gram filters is the computation of the so-called *threshold* which defines the filter criterium. We describe the first general method for computing the threshold for q-gram filters. The method is based on a carefully chosen precise statement of the problem which is then transformed into a constrained shortest path problem. In its generic form the method leaves certain parts open but is applicable to a large variety of q-gram filters and may be extensible even to other classes of filters. We also give a full algorithm for a specific subclass. For this subclass, the algorithm has been implemented and used succesfully in an experimental comparison.

1 Introduction

Given a *pattern* string P and a *text* string T, the *approximate string matching problem* is to find all substrings of the text (*matches*) that are within a *distance* k of the pattern P. The most commonly used distance measure is the *Levenshtein distance*, the minimum number of single character insertions, deletions and replacements needed to change one string into the other. A simpler variant is the *Hamming distance*, that does not allow insertions and deletions, i.e., it is the number of nonmatching characters for strings of the same length. The *indexed* version of the problem allows preprocessing the text to build an index while the *online* version does not. Surveys are given in [15,16,18].

Filtering is a way to speed up approximate string matching, particularly in the indexed case but also in the online case. A *filter* is an algorithm that quickly discards large parts of the text based on some *filter criterium*, leaving the remaining part to be checked with a proper (online) approximate string matching algorithm. A filter is *lossless* if it never discards an actual occurrence; we consider only lossless filters. The ability of a filter to reduce the text area is called its (*filtration*) *efficiency*.

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 348–357, 2002.

Many filters are based on *q-grams*, substrings of length q. The q-gram similarity (defined as a distance in [25]) of two strings is the number of q-grams shared by the strings. The q-gram filter is based on the *q-gram lemma*:

Lemma 1 ([12]). *Let P and S be strings with (Levenshtein or Hamming) distance k. Then the q-gram similarity of P and S is at least $t = |P| - q + 1 - kq$.*

The value t in the lemma is called the *threshold* and gives the minimum number of q-grams that an approximate match must share with the pattern, which is used as the filter criterium. The method is well-suited for indexed matching using an index of text q-grams.

Above we did not define precisely how to count the number of shared q-grams. There are, in fact, many alternatives giving different tradeoffs between filtration efficiency, filter speed and index size. Here are some variations:

- If the same q-gram occurs r_P times in P and r_S times in S, it would be correct to count it as $\min\{r_P, r_S\}$, r_P, r_S, or $r_P r_S$ shared q-grams.
- As noted in [11], a q-gram need to be counted only if it occurs at approximately the same position in P and S.
- Count the shared q-grams between the pattern and large text areas, buckets. Buckets with less than t shared q-grams can be discarded as a whole [12,4].

For all these variations of counting, the threshold t defining the filter criterium is the one given by Lemma 1; using a higher threshold would make the filter lossy, using a lower threshold would reduce filtration efficiency. This is a reflection of the fact that they are all *upper bound approximations* of the same *core similarity measure*. We define this core similarity measure in Section 2.

There are many ways to generalize the basic method, including the following:

Gapped q-grams. The q-grams may contain gaps. For example, the gapped 3-grams of *shape* `##-#` of the string `acgtc` are `ac-t` and `cg-c`. In [5,6], it is shown that by using q-grams of a carefully chosen shape, the filtration efficiency can be improved significantly. The added complexity makes online filters slower, but on indexed filters the effect is negligible.

Sampling. A popular way to reduce time requirement and/or index size at the cost of filtration efficiency is to consider only a sample, say every 5th, of the q-grams of the text or the pattern [8,24,14,22,23,21,17,19].

Multiple shapes. As an opposite to sampling, the number of q-grams can be increased by using gapped q-grams of several different shapes [7,20]. This improves filtration efficiency but increases time and/or space requirements.

Approximate q-grams. Another way to improve filtration efficiency at the cost of slower filtering is to allow errors in q-grams [14,8,22,19].

Of course, various combinations of these methods are possible. For example, sampling and approximate q-grams have often been used together [14,8,22, 19]. Gapped q-grams, in particular, offer a lot of possibilities for combination through the use of (possibly multiple) different shapes. The recent results in [5, 6] suggest that the possibilities of gapped q-grams are worth exploring. However, the problem is the difficulty of determining the threshold.

All the filtering methods mentioned above can be formulated using a similarity measure based on counting shared q-grams. A text substring is checked only if the similarity between the pattern and the substring is at least a given *threshold*. Most of the methods mentioned above give a simple equation for the threshold. However, when gapped q-grams are involved, things get more complicated. Even for the simple filter in [5] (Hamming distance, single shape, no sampling or approximate q-grams) no simple equation can be given; the threshold was computed separately for each shape with a dynamic programming algorithm. Pevzner and Waterman [20], too, consider only the Hamming distance and give an equation for a very limited class of regular shapes, and even that is not the optimal value but a lower bound (which guarantees losslessness but at a reduced efficiency). Califano and Rigoutsos [7] use a heuristically chosen threshold supported by probabilistic calculations and experiments; their filter is lossy.

In this paper, we consider the problem of computing the threshold. We give a formal definition of the value of the threshold that captures the essence of the concept and describe an algorithm for computing it. The definition and the algorithm are generic, leaving certain parts open, but being applicable to a large variety of q-gram filters. Filling in the missing pieces for a given class of filters is a non-trivial task, but it is simpler and much more precisely defined than trying to define and compute the threshold from scratch. As a concrete example, we also give the missing pieces for the class of filters considered in [6]. For these filters, the algorithm has been implemented.

The outline of the method (and the paper) is as follows. In section 2, we give a simple, precise statement of the threshold computation problem for *core similarity measures* of a specific form. The threshold problem is then transformed into a *constraint shortest path problem* (defined in Section 4) on a graph that depends on the core similarity (as described in Section 3). When applying the method to a specific filter, two things must be specified. First, a core similarity measure of the specified form must be given (Section 2). The similarity measure used by the filter should be an upper bound approximation of the core similarity. Second, an algorithm for building the above mentioned graph for the chosen core similarity must be given (Section 3). On the other hand, the constrained shortest path algorithm (Section 4) can be used for any filter.

2 Threshold

As in the classic q-gram lemma, we define the threshold of a q-gram filter as a function of the length m of the pattern and the distance limit k. That is, the threshold $t(m, k)$ is the smallest number of matching q-grams between a pattern of length m and a substring of the text that is within distance k of the pattern. The number of matching q-grams is a similarity function for strings. Since we are looking for the minimum similarity, we can assume that there are no "accidentally" matching q-grams, i.e., q-grams match only if they are not affected (too much) by the edit operations. Therefore, the minimum is defined by the worst possible arrangement of the edit operations.

Following [10], we define an *edit transcript* as a string over the alphabet M(atch), R(eplace), I(nsert) and D(elete), describing a sequential character-by-character transformation of one string to another. For two strings P and S, let $\mathcal{T}(P,S)$ denote the set of all transcripts transforming P to S. For example, $\mathcal{T}(\texttt{actg},\texttt{acct})$ contains MMRR, MMIMD, MIMMD, IRMMD, IDIMDID, etc.. For a transcript $\tau \in \mathcal{T}(P,S)$, the source length $slen(\tau)$ of τ is the length of P, i.e., the number of non-insertions in τ. The Levenshtein cost $c_L(\tau)$ is the number of non-matches. The Hamming cost $c_H(\tau)$ is infinite if τ contains insertions or deletions and the same as Levenshtein cost otherwise. The Levenshtein distance and Hamming distance of P and S are $d_L(P,S) = \min_{\tau\in\mathcal{T}(P,S)} c_L(\tau)$ and $d_H(P,S) = \min_{\tau\in\mathcal{T}(P,S)} c_H(\tau)$, respectively.

Here we defined distance measures for strings using cost functions for edit transcripts. Similarly, we define the q-gram similarity measures for strings using *profit functions* for edit transcripts. Then we can define the threshold as follows.

Definition 1. *The* threshold *for a cost function c and a profit function p is*

$$t_p^c(m,k) = \min_{\tau}\{p(\tau) \mid slen(\tau) = m, c(\tau) \le k\}.$$

The following lemma gives the filter criterium.

Lemma 2. *Let c be a cost function and p a profit function for edit transcripts. Define a distance d of two strings P and S as $d(P,S) = \min_{\tau\in\mathcal{T}(P,S)} c(\tau)$ and a similarity s as $s(P,S) = \max_{\tau\in\mathcal{T}(P,S)} p(\tau)$. Now, if $d(P,S) \le k$, then $s(P,S) \ge t_p^c(|P|,k)$.*

The lemma holds for any choice of cost c and profit p. The cost functions leading to the Hamming and Levenshtein distances were defined above. Below, we give examples of profit functions that define q-gram similarity measures.

Let I be a set of integers. The *span* of I is $span(I) = \max I - \min I + 1$, i.e., the size of the minimum contiguous interval containing I. The *position* of I is $\min I$, and the *shape* of I is the set $\{i - \min I \mid i \in I\}$. An integer set Q with position zero is called a *shape*. For any shape Q and integer i, let Q_i denote the set with shape Q and position i, i.e., $Q_i = \{i + j \mid j \in Q\}$. Let $Q_i = \{i_1, i_2, \ldots, i_q\}$, where $i = i_1 < i_2 < \cdots < i_q$, and let $S = s_1 s_2 \ldots s_m$ be a string. For $1 \le i \le m - span(Q) + 1$, the *$Q$-gram* at position i in S, denoted by $S[Q_i]$, is the string $s_{i_1} s_{i_2} \ldots s_{i_q}$. For example, if $S = \texttt{acagagtct}$ and $Q = \{0,2,3,6\}$, then $S[Q_1] = S[Q_3] = \texttt{aagt}$ and $S[Q_2] = \texttt{cgac}$.

A *match alignment* M_τ of a transcript τ is the set of pairs of positions that are matched to each other. For example, $M_{\texttt{MIMRDMR}} = \{(1,1),(2,3),(5,5)\}$. For a set I of integers, let $M_\tau(I)$ be the set to which M_τ maps I, i.e., $M_\tau(I) = \{j \mid i \in I \text{ and } (i,j) \in M_\tau\}$. A *$Q$-hit* in a transcript τ is a pair (i,j) such that $M_\tau(Q_i) = Q_j$. Note that a Q-hit (i,j) in $\tau \in \mathcal{T}(P,S)$ implies $P[Q_i] = S[Q_j]$.

Now we are ready to define our first profit function. The *Q-profit* $p_Q(\tau)$ of a transcript τ is the number of its Q-hits, i.e., $p_Q(\tau) = |\{(i,j) \mid M_\tau(Q_i) = Q_j\}|$. Using p_Q as the profit function defines the *Q-similarity* of two strings P and S as $s_Q(P,S) = \max_{\tau\in\mathcal{T}(P,S)} p_Q(\tau)$. If Q is the contiguous shape $\{0,1,\ldots,q-1\}$,

s_Q is the core similarity measure underlying the classic q-gram filter, and the threshold of Definition 1 agrees with the one in Lemma 1 for both Hamming and Levenshtein distance. For a gapped shape Q, s_Q is the core similarity measure for the Hamming distance filters described in [5].

As a more complicated example, let us define the profit functions for the Levenshtein distance filters in [6]. The filters use a basic shape with only one gap and two other shapes formed from the basic shape by increasing and decreasing the length of the gap by one. For example, with the basic shape `##-#` we would also use the shapes `##--#` and `###`. The filter compares the q-grams of all three shapes in the pattern to the q-grams of the basic shape in the text.

For any $b_1, g, b_2 > 0$, let (b_1, g, b_2) denote the *one-gap shape* $\{0, \ldots, b_1 - 1, b_1 + g, \ldots, b_1 + g + b_2 - 1\}$. For a one-gap shape $Q = (b_1, g, b_2)$, let $Q^{+1} = (b_1, g + 1, b_2)$ and $Q^{-1} = (b_1, g - 1, b_2)$ (or $Q^{-1} = \{0, \ldots, b_1 + b_2 - 1\}$ if $g = 1$). Then, a $Q \pm 1$-*hit* in a transcript τ is a pair (i, j) of integers such that $Q_j \in \{M_\tau(Q_i^{-1}), M_\tau(Q_i), M_\tau(Q_i^{+1})\}$. If $\tau \in \mathcal{T}(P, S)$, then a $Q \pm 1$-hit (i, j) in τ implies $S[Q_j] \in \{P[Q_i^{-1}], P[Q_i], P[Q_i^{+1}]\}$, which is the criterion for a shared q-gram for the filters in [6]. The $Q \pm 1$-*profit* of τ, denoted by $p_{Q\pm1}(\tau)$, is the number of $Q \pm 1$-hits in τ, i.e., $p_{Q\pm1}(\tau) = |\{(i, j) \mid Q_j \in \{M_\tau(Q_i^{-1}), M_\tau(Q_i), M_\tau(Q_i^{+1})\}\}|$. The $Q \pm 1$-*similarity* of two strings P and S is $s_{Q\pm1}(P, S) = \max_{\tau \in \mathcal{T}(P,S)} p_{Q\pm1}(\tau)$.

The similarities s_Q and $s_{Q\pm1}$ are *core similarity measures*. Computing them for given strings is not straightforward. Instead filters use simpler *upper bound approximations* that may give a higher similarity value. Designing good upper bound approximations is nontrivial and beyond the scope of this paper.

3 Profit Automata

In the remainder of the paper, we describe a general technique for computing the threshold according to Definition 1. The technique is based on automata for computing the values $slen(\tau)$, $c(\tau)$, and $p(\tau)$ for any transcript τ. These automata have four transitions out of each state labeled with M, R, I and D. Each transition has an output value, which is the change in the target value caused by the transition. Thus, the target value is the sum of the output labels on the transition path corresponding to τ. Such automata are more generally known as *weighted finite automata* [3] or *string-to-weight transducers* [13]. Fig. 1 shows simple examples.

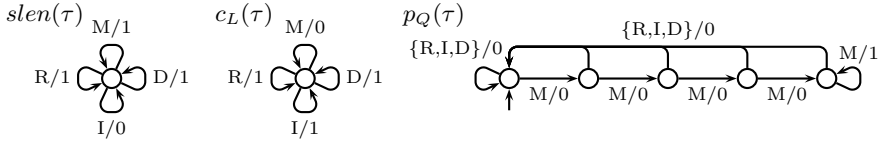


Fig. 1. Automata for computing the source length $slen(\tau)$, Levenshtein cost $c_L(\tau)$ and Q-profit $p_Q(\tau)$, where $Q = \{0, 1, 2, 3, 4\}$, for any transcript τ

For more complicated forms of profits the automata are also more complicated. We show how to build the automaton for the profit $p_{Q\pm1}$ for any one-gap shape $Q = (b_1, g, b_2)$. Automata for other profits can be build using similar ideas.

We start by describing a dynamic programming algorithm for finding the $Q \pm 1$-hits for a given transcript $\tau = \tau_1 \dots \tau_n$. Table 1 defines the computation of an entry $P[i, j]$ in a table $P[0..s, 0..n]$, where $s = b_1 + g + b_2$ is the span of Q. A column j in table P stores the state of the computation after reading the prefix $\tau_1 \cdots \tau_j$ of τ. Each entry represents a set of shape Q overlapping or touching the cut point between τ_j and τ_{j+1}. The value infinite means that the set cannot be a part of a $Q \pm 1$-hit. When the cut point is in the gap, a finite value represents the change in the length of the gap (insertion–deletion difference). An example is shown in Fig. 2. In the last row of the table P, each zero signifies a hit. Thus the profit can be computed as $p_{Q\pm1}(\tau) = \big|\{j \in \{1, \dots, n\} \mid P[s, j] = 0\}\big|$.

Table 1. Rules for computing the values in the dynamic programming table $P[0..s, 0..n]$

	$j = 0$	$\tau_j = \mathrm{M}$	$\tau_j = \mathrm{R}$
$i = 0$	0		
$1 \le i \le b_1$	∞	$P[i-1, j-1]$	∞
$b_1 \le i \le b_1 + g$			$P[i-1, j-1]$
$i = b_1 + g + 1$		0 if $-1 \le P[i-1, j-1] \le 1$ ∞ otherwise	∞
$b_1 + g + 1 < i \le s$		$P[i-1, j-1]$	

	$\tau_j = \mathrm{I}$	$\tau_j = \mathrm{D}$
$i = 0$	0	
$1 \le i < b_1$	∞	
$i = b_1$		$P[i, j-1] - 1$ if $P[i, j-1] + b_1 + g - i \ge 0$
$b_1 < i \le b_1 + g$	$P[i-1, j-1] + 1$	∞ otherwise
$b_1 + g < i \le s$	∞	

A column j in table P depends only on the previous column $j - 1$ and the symbol τ_j. Thus the computation can be done with an automaton, where each state represents a distinct column. The first and last entry can be omitted from the state description: the first because it is always 0, the last because it does not affect the next column. Instead, the last entry determines the output value of the transition: 1 when last entry is 0, 0 when the last entry is ∞. The result is an automaton for computing the profit $p_{Q\pm1}(\tau)$ similar to the automatons in Fig. 1. An example is given in Fig. 3.

The size of the automaton is bounded in the following lemma. The proof is omitted in this extended abstract.

τ_j	τ_{j+1}		M	M	R	D	M	M	I	R	M	M	M	M	D	D	I	M	M	D	D	D	M
	# # − #	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
#	# − #	1	∞	0	0	∞	∞	0	0	∞	∞	0	0	0	∞	∞	∞	0	0	∞	∞	∞	0
# #	− #	2	∞	∞	0	∞	∞	∞	0	∞	∞	∞	0	0	0	−1	−2	∞	∞	0	−1	−2	∞	∞
# # −	#	3	∞	∞	∞	0	−1	∞	∞	+1	∞	∞	∞	0	0	−1	∞	−1	∞	∞	∞	∞	∞	∞
# # − #		4	∞	∞	∞	∞	∞	0	∞	∞	∞	∞	∞	∞	0	∞	∞	∞	0	∞	∞	∞	∞	∞

Fig. 2. The dynamic programming table P for $Q = (2, 1, 1) = \{0, 1, 3\}$

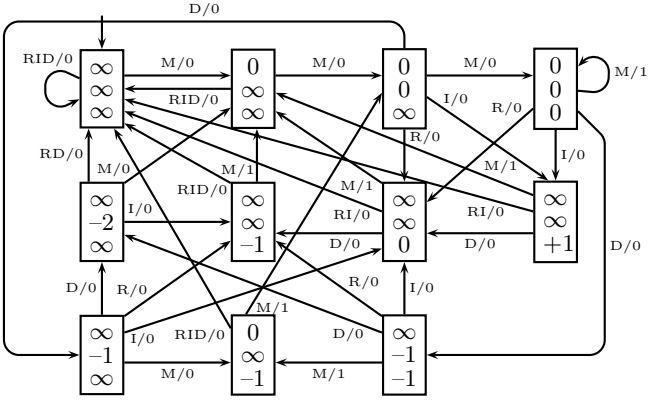


Fig. 3. An automaton for computing the profit $p_{Q\pm1}(\tau)$ for $Q = (2, 1, 1) = \{0, 1, 3\}$

Lemma 3. *The number of states in the profit automaton for the profit $p_{Q\pm1}$, where $Q = (b_1, g, b_2)$, is less than $R = b_1 b_2^2 (g + 2)^{3\lceil g/b_1 \rceil}$. The automaton can be constructed in $\mathcal{O}(R(b_1 + g + b_2))$ time and $\mathcal{O}(Rg/b_1)$ space.*

4 Constrained Shortest Path Problem

We are now ready to describe the threshold computation problem as a constraint shortest path problem. The graph is formed by combining the three automata for computing $slen(\tau)$, $c(\tau)$ and $p(\tau)$. In general, the state set of the combined automaton is the Cartesian product of the state sets of the three automata, but since the automata for source length and Leveshtein or Hamming cost have just one state, the graph is essentially the profit automaton with additional labels.

Thus, we have a directed graph $G = (V, E)$, where each edge $e \in E$ is labeled with three non-negative values: $l(e)$, $c(e)$ and $p(e)$ corresponding to the (source) length, cost, and profit, respectively. The values are additive along paths. One node is designated as the source node s (the initial state). Each node has at most four outward edges.

Now, the treshold computation problem can be stated as follows:

Problem 1. Find the minimum profit $p(\pi)$ of a path π in G starting from the source node (and ending anywhere) that satisfies: (1) $c(\pi) \leq k$ and (2) $l(\pi) = m$.

The constraint (2) can be replaced with $l(\pi) \geq m$ without changing the solution.

This problem is very similar to the constrained shortest path (CSP) problem [26], differing in two ways from the standard form. First, no target node for the path is specified. However, by creating an additional node t as the target and adding an all-zero edge from every other node to t, the problem can be stated in the more usual form. Second, in the standard form all constraints are of type (1), i.e., limited from above. Limited-from-below constraints change the problem in a nontrivial way, e.g., the shortest path may contain cycles. However, many of the basic techniques used in CSP algorithms are still applicable.

The CSP problem is NP-hard, but there are pseudopolynomial algorithms, i.e., algorithms that work in polynomial time when the edge labels are polynomially bounded integers. We give a pseudopolynomial algorithm that belongs to a well-known class of shortest path algorithms called label-setting algorithms, which can be seen as generalizations of Dikstra's algorithm [1, Chapter 4]. For the CSP problem, label-setting algorithms have been given in [2,9]. Our algorithm uses one nonstandard trick to deal with the limited-from-below constraint.

The basic idea of the algorithm is to maintain a collection of paths (initially only the empty path) in a priority queue PQ. The paths are selected from PQ in increasing order of their profit, extended along the four outward egdes, and the extensions are added to the PQ. Paths with cost more than k are removed. Since an extension cannot decrease the profit, the first path of required length selected from PQ is an optimal path.

To prune the set of paths, we use the following concept of domination:

Definition 2. *Let π_1 and π_2 be two paths from s to v. We say that* π_1 dominates π_2 *if $p(\pi_1) \leq p(\pi_2)$, $c(\pi_1) \leq c(\pi_2)$ and $l(\pi_1) \geq l(\pi_2)$. If equality holds in each case, the paths are called* equivalent. *Otherwise, the domination is* strict.

Clearly, if π_1 dominates π_2, any extension of π_1 dominates the corresponding extension of π_2. Thus, no extensions of π_2 need to be considered. The algorithm extends a path only if it is not dominated by an already processed path and is not strictly dominated by any path to be processed later. To check for domination by already processed paths, information about the dominant paths is maintained at nodes. To avoid extending a path that is strictly dominated by a later path, the priority queue order is refined. The profit remains the primary key but secondary keys are used to break ties. The details are left to the full paper. The complexity of the algorithm is given by the following theorem.

Theorem 1. *Given the graph $G = (V, E)$, the algorithm computes the threshold t in $\mathcal{O}(k|V| \min\{m, t\} + kd_{max}t)$ time and $\mathcal{O}(k|V|p_{max})$ space, where m and k are the limits of length and cost, respectively, p_{max} is the largest profit value of an edge, and d_{max} is the length of the longest zero-cost, zero-profit path in G.*

By combining Theorem 1 with Lemma 3 we get the following result.

Theorem 2. *Given a one-gap shape $Q = (b_1, g, b_2)$ and positive integers m and k, the threshold $t^{cL}_{p_{Q\pm1}}(m, k)$ can be computed in $\mathcal{O}(kmb_1b_2^2g^{3\lceil g/b_1\rceil})$ time and $\mathcal{O}((k + g/b_1)b_1b_2^2g^{3\lceil g/b_1\rceil})$ space.*

5 Concluding Remarks

We have described a method for computing the threshold of a q-gram filter that is applicable to a large variety of filters. The practicality of the method is demonstrated by the implementation of the algorithm for the filters based on the $s_{Q\pm1}$ similarity. It has been used in an experimental comparison of q-gram filters, the results of which are described in [6]. The shortest paths part of the implementation can be reused for other classes of filters without modification.

The method is an important step towards designing good q-gram filters. First, the threshold (or at least a good lower bound) is needed by any filter. A general method gives us a large family of filters to choose from. Second, the value of the threshold is an important criterium in comparing filters (particularly in choosing the shapes of q-grams [5,6]). Third, the framework developed here may be helpful in designing filters. In particular, the separation of core similarity measures and their upper bound approximations seems a useful concept.

Acknowledgements. Discussions with Stefan Burkhardt, Mark Ziegelmann and Kurt Melhorn have contributed to this paper. The implementation is partly due to Stefan Burkhardt.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, 1993.
2. Y. P. Aneja, V. Aggarwal, and K. P. K. Nair. Shortest chain subject to side conditions. *Networks*, 13:295–302, 1983.
3. A. I. Buchsbaum, R. Giancarlo, and J. R. Westbrook. On the determinization of weighted finite automata. *SIAM J. Comput.*, 30(5):1502–1531, 2000.
4. S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (QUASAR). In *Proc. 3rd Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 77–83. ACM Press, 1999.
5. S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *LNCS*, pages 73–85. Springer, 2001.
6. S. Burkhardt and J. Kärkkäinen. One-gapped q-gram filters for Levenshtein distance. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching*, LNCS. Springer, 2002. To appear.
7. A. Califano and I. Rigoutsos. FLASH: A fast look-up algorithm for string homology. In *Proc. 1st International Conference on Intelligent Systems for Molecular Biology*, pages 56–64. AAAI Press, 1993.

8. W. I. Chang and T. G. Marr. Approximate string matching and local similarity. In *Proc. 5th Annual Symposium on Combinatorial Pattern Matching*, volume 807 of *LNCS*, pages 259–273. Springer, 1994.
9. L. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis. Time constrained routing and scheduling. In M. O. Ball et al., editors, *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, chapter 2, pages 35–139. North-Holland, 1995.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
11. N. Holsti and E. Sutinen. Approximate string matching using q-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32, 1994.
12. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 16th Symposium on Mathematical Foundations of Computer Science*, volume 520 of *LNCS*, pages 240–248. Springer, 1991.
13. M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.
14. E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.
15. G. Navarro. *Approximate Text Searching.* PhD thesis, Dept. of Computer Science, University of Chile, 1998.
16. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
17. G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`.
18. G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001. Special issue on Managing Text Natively and in DBMSs.
19. G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching*, volume 1848 of *LNCS*, pages 350–363. Springer, 2000.
20. P. A. Pevzner and M. S. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13(1/2):135–154, 1995.
21. F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. 3rd South American Workshop on String Processing*, pages 257–271. Carleton University Press, 1996.
22. E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. 3rd Annual European Symposium on Algorithms*, volume 979 of *LNCS*, pages 327–340. Springer, 1995.
23. E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching*, volume 1075 of *LNCS*, pages 50–63. Springer, 1996.
24. T. Takaoka. Approximate pattern matching with samples. In *Proc. 5th International Symposium on Algorithms and Computation (ISAAC)*, volume 834 of *LNCS*, pages 236–242. Springer, 1994.
25. E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–212, 1992.
26. M. Ziegelmann. *Constrained Shortest Paths and Related Problems.* PhD thesis, Universität des Saarlandes, Germany, 2001.

On the Generality of Phylogenies from Incomplete Directed Characters

Itsik Pe'er[1], Ron Shamir[1], and Roded Sharan[1]

School of Computer Science, Tel-Aviv University, Tel-Aviv, 69978 Israel,
{izik,rshamir,roded}@post.tau.ac.il

Abstract. We study a problem that arises in computational biology, when wishing to reconstruct the phylogeny of a set of species. In Incomplete Directed Perfect Phylogeny (IDP), the characters are binary and directed (i.e., species can only gain characters), and the states of some characters are unknown. The goal is to complete the missing states in a way consistent with a perfect phylogenetic tree. This problem arises in classical phylogenetic studies, when some states are missing or undetermined, and in recent phylogenetic studies based on repeat elements in DNA. The problem was recently shown to be polynomial. As different completions induce different trees, it is desirable to find a *general* solution tree. Such a solution is consistent with the data, and every other consistent solution can be obtained from it by node splitting. Unlike the situation for complete datasets, a general solution may not exist for IDP instances. We provide a polynomial algorithm to find a general solution for an IDP instance, or determine that none exists.

1 Introduction

A *phylogenetic tree* describes the divergence patterns leading from a single ancestor species to its contemporary descendants. The task of *phylogenetic reconstruction* is to infer a phylogenetic tree from information regarding the contemporary species (see, e.g., [2]).

The character-based approach to tree reconstruction describes contemporary species by their attributes or *characters*. Each character takes on one of several possible discrete *states*. The input is represented by a matrix $\mathcal{A}$ where a_{ij} is the state of character j in species i, and the i-th row is the *character vector* of species i. The output sought is a phylogenetic tree along with the suggested character vectors of the internal nodes. This output must satisfy properties specified by the problem variant.

In this paper, we discuss a phylogenetic reconstruction problem called *Incomplete Directed Perfect Phylogeny (IDP)* [6]. It is assumed that each character is binary, where its absence or presence is denoted by 0 or 1, respectively. A character may be gained at most once (across the phylogenetic tree), but may never be lost. The input is a matrix of character vectors, where some character states are missing. The question is whether one can complete the missing states so that the resulting matrix admits a tree satisfying the above assumptions.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 358–367, 2002.

The problem of handling incomplete phylogenetic data arises whenever some of the data is missing or ambiguous. Quite recently, a novel kind of genomic data has given rise to the same problem: Nikaido et al. [5] use inserted repetitive genomic elements, particularly SINEs (Short Interspersed Nuclear Elements), as a source of evolutionary information. SINEs are short DNA sequences that were copied and randomly reinserted into various genomic loci during evolution. The distinct insertion loci are identifiable by the flanking sequences on both sides of the insertion site. These insertions are assumed to be unique events in evolution. Furthermore, a SINE insertion is assumed to be irreversible, i.e., once a SINE sequence has been inserted somewhere along the genome, it is practically impossible for the exact, complete SINE to leave that specific locus. However, the inserted segment along with its flanking sequences may be lost when a large genomic region, which includes them, is deleted. In that case we do not know whether a SINE insertion had occurred in the missing site prior to its deletion. One can model such data by assigning to each locus a character, whose state is '1' if the SINE occurred in that locus, '0' if the locus is present but does not contain the SINE, and '?' if the locus is missing. The resulting reconstruction problem is precisely Incomplete Directed Perfect phylogeny.

Previous studies of related problems implied $\Omega(n^2m)$-time algorithms for IDP [1,3], where n and m denote the number of species and characters, respectively. In a recent work [6] we provided near optimal $O(nm \cdot polylog(n+m))$-time algorithms for the problem.

In this paper we tackle a different aspect of IDP. Often there is more than one tree that is consistent with the data. When the input matrix is complete, there is always a tree $\mathcal{T}^*$ that is *general*, i.e., it is a solution, and every other tree consistent with the data can be obtained from $\mathcal{T}^*$ by node splitting. In other words, $\mathcal{T}^*$ describes all the definite information in the data, and ambiguities (nodes with three or more children) can be resolved by additional information. This is not always the case if the data matrix is incomplete: There may or may not be a general solution tree. In the latter case, any particular solution tree we choose may be ruled out by additional information, while this information is still consistent with an alternative solution tree. It is thus desirable to decide if a general solution exists and to generate such a solution if the answer is positive.

In this study we provide answers to both questions. We prove that an algorithm from [6], which we call Solve_IDP, provides the general solution of a problem instance, if such exists. We also give an algorithm which determines if the solution $\mathcal{T}$ produced by Solve_IDP is indeed general. The complexity of the latter algorithm is $O(nm + kd)$, where k is the number of 1-entries in the input matrix, and d denotes the maximum degree of $\mathcal{T}$.

The paper is organized as follows: In Section 2 we provide some preliminaries and background on the IDP problem. In Section 3 we analyze the generality of the solution produced by Solve_IDP and the complexity of testing generality. For lack of space some proofs are sketched or omitted.

2 Preliminaries

We first specify some terminology and notation. Matrices are denoted by an upper-case letter, while their elements are denoted by the corresponding lower-case letter. Let $G = (V, E)$ be an undirected graph. We denote its set of vertices V also by $V(G)$. A nonempty set $W \subseteq V$ is *connected* in G, if there is a path in G between every pair of vertices in W.

Let $\mathcal{T}$ be a rooted tree over a leaf set S. The *out-degree* of a node x in $\mathcal{T}$ is its number of children, and is denoted by $d(x)$. For a node x in $\mathcal{T}$ we denote the leaf set of the subtree rooted at x by $L(x)$. $L(x)$ is called a *clade* of $\mathcal{T}$. For consistency, we consider $\emptyset$ to be a clade of $\mathcal{T}$ as well, and call it the *empty clade*. $S, \emptyset$ and all singletons are called *trivial clades*.

Observation 1 *(cf. [4]) A collection $\mathcal{C}$ of subsets of a set S is the set of clades of some tree over S if and only if $\mathcal{C}$ contains the trivial clades and for every intersecting pair of its subsets, one contains the other.*

A tree $\mathcal{T}$ is uniquely characterized by its set of clades. The transformation between a branch-node representation of a tree and a list of its clades is straightforward. Thus, we hereafter identify a tree with the set of its clades.

Throughout the paper we denote by $S = \{s_1, \ldots, s_n\}$ the set of all species and by $C = \{c_1, \ldots, c_m\}$ the set of all (binary) characters. For a graph K, we define $S(K) \equiv S \cap V(K)$. Let $\mathcal{B}_{n \times m}$ be a binary matrix whose rows and columns correspond to species and characters, respectively, and $b_{ij} = 1$ if and only if the species s_i has the character c_j. A *phylogenetic tree for* $\mathcal{B}$ is a rooted tree $\mathcal{T}$ with n leaves corresponding to the n species of S, such that each character is associated with a clade of $\mathcal{T}$, and the following properties are satisfied:
(1) If c_j is associated with a clade S' then $s_i \in S'$ if and only if $b_{ij} = 1$.
(2) Every non-trivial clade of $\mathcal{T}$ is associated with at least one character.

A $\{0, 1, ?\}$ matrix is called *incomplete*. For convenience, we also consider binary matrices as incomplete. Let $\mathcal{A}_{n \times m}$ be an incomplete matrix in which $a_{ij} = 1$ if s_i has c_j, $a_{ij} = 0$ if s_i lacks c_j, and $a_{ij} =?$ if it is not known whether s_i has c_j. For two subsets $S' \subseteq S$ and $C' \subseteq C$ we denote by $\mathcal{A}|_{S',C'}$ the submatrix of $\mathcal{A}$ induced on $S' \times C'$. For a character c_j and a state $x \in \{0, 1, ?\}$, the *x-set* of c_j in $\mathcal{A}$ is the set of species $\{s_i \in S : a_{ij} = x\}$. c_j is called a *null character* if its 1-set is empty. We denote $E_x^{\mathcal{A}} = \{(s_i, c_j) : a_{ij} = x\}$, for $x = 0, 1, ?$. (In the sequel, we omit the superscript $\mathcal{A}$ when it is clear from the context.)

A binary matrix $\mathcal{B}$ is called a *completion* of $\mathcal{A}$ if $E_1^{\mathcal{A}} \subseteq E_1^{\mathcal{B}} \subseteq E_1^{\mathcal{A}} \cup E_?^{\mathcal{A}}$. Thus, a completion replaces all the ?-s in $\mathcal{A}$ by zeroes and ones. If $\mathcal{B}$ has a phylogenetic tree $\mathcal{T}$, we say that $\mathcal{T}$ is a *phylogenetic tree for* $\mathcal{A}$ as well. We also say that $\mathcal{T}$ *explains* $\mathcal{A}$ *via* $\mathcal{B}$. An example of these definitions is given in Figure 1.

The problem of Incomplete Directed Perfect Phylogeny is defined as follows:
Incomplete Directed Perfect Phylogeny (IDP):
Instance: An incomplete matrix $\mathcal{A}$.
Goal: Find a phylogenetic tree for $\mathcal{A}$, or determine that no such tree exists.

Let $\mathcal{B}$ be a species-characters binary matrix of order $n \times m$. Construct the bipartite graph $G(\mathcal{B}) = (S, C, E)$ with $E = \{(s_i, c_j) : b_{ij} = 1\}$. An induced path

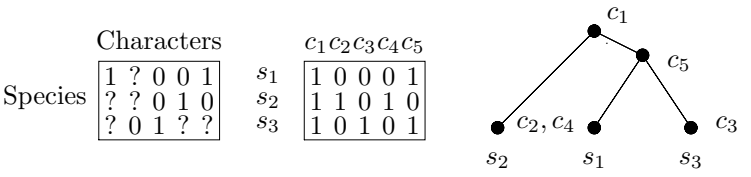


Fig. 1. Left to right: An incomplete matrix $\mathcal{A}$, a completion $\mathcal{B}$ of $\mathcal{A}$, and a phylogenetic tree that explains $\mathcal{A}$ via $\mathcal{B}$. Each character is written to the right of the root of its associated clade. (Example taken from [6].)

of length four in $G(\mathcal{B})$ is called a Σ *subgraph* if it starts (and therefore ends) at a vertex corresponding to a species. A bipartite graph with no induced Σ subgraph is said to be Σ*-free*.

We now recite several characterizations of IDP.

Theorem 2 ([6]). *$\mathcal{B}$ has a phylogenetic tree if and only if $G(\mathcal{B})$ is Σ-free.*

In [6] we used Theorem 2 to reformulate IDP as a graph sandwich problem. Finding a completion of an input matrix $\mathcal{A}$ was shown to be equivalent to finding a Σ-free supergraph of $G(\mathcal{A})$ whose set of edges does not intersect $E_0^{\mathcal{A}}$.

Corollary 1. *Let $\hat{S} \subseteq S$ and $\hat{C} \subseteq C$ be subsets of the species and characters, respectively. If $\mathcal{A}$ has a phylogenetic tree, then so does $\mathcal{A}|_{\hat{S},\hat{C}}$.*

Corollary 2. *Let $\mathcal{A}$ be a matrix explained by a tree $\mathcal{T}$ and let $\hat{S} = L(x)$ be a clade in $\mathcal{T}$. Then $\mathcal{A}|_{\hat{S}\times C}$ is explained by the subtree of $\mathcal{T}$ rooted at x.*

For a subset $S' \subseteq S$ of species, we say that a character c is S'*-universal* in $\mathcal{B}$, if its 1-set contains S' (i.e., every species in S' has that character).

Proposition 1 ([6]). *If $G(\mathcal{B})$ is connected and Σ-free, then there exists a character which is S-universal in $\mathcal{B}$.*

2.1 An Algorithm for IDP

In this section we briefly describe an algorithm for IDP, given in [6]. Let $\mathcal{A}$ be the input matrix. We denote by $B(\mathcal{A})$ the binary matrix of $\mathcal{A}$'s dimension with ?-s replaced by zeros. Define $G(\mathcal{A}) \equiv G(B(\mathcal{A})) = (S, C, E_1^{\mathcal{A}})$. For a nonempty subset $S' \subseteq S$, we say that a character is S'*-semi-universal* in $\mathcal{A}$ if its 0-set does not intersect S'.

The algorithm for solving IDP is described in Figure 2. It outputs the set of non-empty clades of a tree explaining $\mathcal{A}$, or *False* if no such tree exists. The algorithm is recursive and is initially called with Solve_IDP($\mathcal{A}$). It was shown in [6] that Solve_IDP has a deterministic implementation which takes $O(nm + |E_1| \log^2 l)$ time, and a randomized implementation which takes $O(nm + |E_1| \log(l^2/|E_1|) + l(\log l)^3 \log\log l)$ expected time, where $l = n + m$.

Solve_IDP($\mathcal{A}$)**:**

1. **If** $|S| > 1$ then **do**:
   a) Remove all S-semi-universal characters and all null characters from $G(\mathcal{A})$.
   b) If the resulting graph G' is connected then output *False* and **halt**.
   c) Otherwise, let $K_1, \ldots, K_r$ be the connected components of G', and let $\mathcal{A}_1, \ldots, \mathcal{A}_r$ be the corresponding sub-matrices of $\mathcal{A}$.
   d) **For** $i = 1, \ldots, r$ **do**: Solve_IDP($\mathcal{A}_i$).
2. Output S.

Fig. 2. An algorithm for solving IDP [6].

3 Determining the Generality of the Solution

A 'yes' instance of IDP may have several distinct phylogenetic trees as solutions. These trees may be related in the following way: We say that a tree $\mathcal{T}$ *generalizes* a tree $\mathcal{T}'$, and write $\mathcal{T} \subseteq \mathcal{T}'$, if every clade of $\mathcal{T}$ is a clade of $\mathcal{T}'$, i.e., the evolutionary hypothesis expressed by $\mathcal{T}'$ includes all the details of the hypothesis expressed by $\mathcal{T}$, and possibly more. Therefore, $\mathcal{T}'$ represents a more specific hypothesis, and $\mathcal{T}$ represents a more general one. We say that a tree $\mathcal{T}$ is the *general solution* of an instance $\mathcal{A}$, if $\mathcal{T}$ explains $\mathcal{A}$, and generalizes every other tree which explains $\mathcal{A}$. Figure 3 demonstrates the definitions, and also gives an example of an instance which has no general solution.

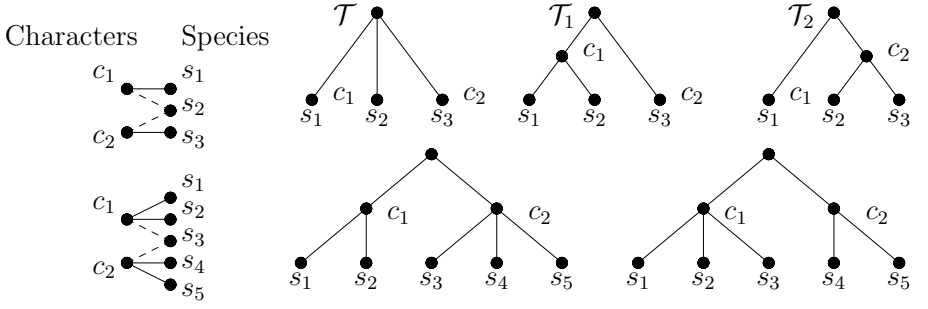


Fig. 3. Top left: An IDP instance which has a general solution. Dashed lines denote $E_?$-edges, while solid lines denote E_1-edges. Top-right: $\mathcal{T}, \mathcal{T}_1$ and $\mathcal{T}_2$ are the possible solutions. $\mathcal{T}$ generalizes $\mathcal{T}_1$ and $\mathcal{T}_2$ (which are obtained by splitting the root node of $\mathcal{T}$), and is the general solution. Bottom left: An IDP instance which has no general solution. Bottom middle and bottom right: Two possible solutions. The only tree which generalizes both solutions is the tree comprised of the trivial clades only, but this tree is not a solution.

3.1 Finding a General Solution

We prove in this section that whenever a general solution exists, Solve_IDP finds it. We use the following notation: Let $\mathcal{A}$ be an incomplete matrix and let $\hat{S} \subseteq S$. We denote by $W_{\mathcal{A}}(\hat{S})$ the set of $\hat{S}$-semi-universal characters in $\mathcal{A}$. Note that if $\mathcal{A}$ is binary, then $W_{\mathcal{A}}(\hat{S})$ is its set of $\hat{S}$-universal characters. We define the operator $\widetilde{\ }$ on incomplete matrices: We denote by $\widetilde{\mathcal{A}}$ the submatrix $\mathcal{A}|_{S,C\setminus W_{\mathcal{A}}(S)}$ of $\mathcal{A}$. In particular, $G(\widetilde{\mathcal{A}})$ is the graph produced from $G(\mathcal{A})$ by removing its set of S-semi-universal vertices.

Lemma 1. *Let $\mathcal{T}$ be the general solution for an instance $\mathcal{A}$ of IDP. Let $S' = L(x)$ be a clade of $\mathcal{T}$, corresponding to some node x. Let $\mathcal{T}'$ be the subtree of $\mathcal{T}$ rooted at x, and let $\mathcal{A}'$ be the instance induced on $S' \cup C$. Then $\mathcal{T}'$ is the general solution for $\mathcal{A}'$.*

Proof. By Corollary 2, $\mathcal{T}'$ explains $\mathcal{A}'$. Suppose that $\mathcal{T}''$ also explains $\mathcal{A}'$ and $\mathcal{T}' \not\subseteq \mathcal{T}''$. Then $\hat{\mathcal{T}} = (\mathcal{T} \setminus \mathcal{T}') \cup \mathcal{T}''$ explains $\mathcal{A}$, and $\mathcal{T} \not\subseteq \hat{\mathcal{T}}$, a contradiction. □

A nonempty clade of a tree is called *maximal* if the only clade that properly contains it is S.

Lemma 2. *Let $\mathcal{T}$ be a phylogenetic tree for a binary matrix $\mathcal{B}$. A non-empty clade S' of $\mathcal{T}$ is maximal if and only if S' is the species set of some connected component of $G(\widetilde{\mathcal{B}})$.*

Proof. Suppose that S' is a maximal clade of $\mathcal{T}$. We claim that S' is contained in some connected component K of $G(\widetilde{\mathcal{B}})$. If $|S'| = 1$ this trivially holds. Otherwise, the character c associated with S' connects all its species, and $c \notin W_{\mathcal{B}}(S)$, proving the claim. Proposition 1 implies that S is disconnected in $G(\widetilde{\mathcal{B}})$ and, therefore, $S' \subseteq S(K) \subset S$. Suppose to the contrary that $S(K)$ properly contains S'. In particular, $|S(K)| > 1$. By Proposition 1, there exists a character c' in $G(\widetilde{\mathcal{B}})$ whose 1-set is $S(K)$. Hence, $S(K)$ must be a clade of $\mathcal{T}$ which is associated with c', contradicting the maximality of S'.

To prove the converse, let S' be the species set of some connected component K of $G(\widetilde{\mathcal{B}})$. We first claim that S' is a clade. If $|S'| = 1$, S' is a trivial clade. Otherwise, by Proposition 1, there exists an S'-universal character c' in $G(\widetilde{\mathcal{B}})$. Since K is a connected component, c' has no neighbors in $S \setminus S'$. Hence, S' must be a clade in $\mathcal{T}$. Suppose to the contrary that S' is not maximal, then it is properly contained in a maximal clade S'', which by the previous direction is the species set of K, a contradiction. □

Theorem 3. *Solve_IDP produces the general solution for every IDP instance that has one.*

Proof. Let $\mathcal{A}$ be an instance of IDP for which there exists a general solution $\mathcal{T}^*$. Let $\mathcal{T}_{alg}$ be the solution tree produced by Solve_IDP. By definition $\mathcal{T}^* \subseteq \mathcal{T}_{alg}$. Suppose to the contrary that $\mathcal{T}^* \neq \mathcal{T}_{alg}$. Let S' be the largest clade in $\mathcal{T}_{alg} \setminus \mathcal{T}^*$

(S' must be non-trivial), and let S'' be the smallest clade in $\mathcal{T}_{alg}$ which properly contains it. Let $\mathcal{A}'$ be the instance induced on $S'' \cup C$. By Corollary 2, $\mathcal{A}'$ is explained by the corresponding subtrees $\mathcal{T}'_{alg}$ of $\mathcal{T}_{alg}$ and $\mathcal{T}'^*$ of $\mathcal{T}^*$. By Lemma 1, $\mathcal{T}'^*$ is the general solution of $\mathcal{A}'$. Due to the recursive nature of Solve_IDP, it produces $\mathcal{T}'_{alg}$ when invoked with input $\mathcal{A}'$. Thus, w.l.o.g., one can assume that $S'' = S$ and S' is a maximal clade of $\mathcal{T}_{alg}$.

Suppose that $\mathcal{T}^*$ explains $\mathcal{A}$ via a completion $\mathcal{B}^*$, and let $G^* = G(\mathcal{B}^*)$. Since S' is a maximal clade, it is reported during a second level call of Solve_IDP$(\cdot)$ (the call at the first level reports the trivial clade S). Hence, it must be the species set of some connected component K in $G(\widetilde{\mathcal{A}})$. Since every S-universal character in G^* is S-semi-universal in $\mathcal{A}$, S' is contained in some connected component K^* of $G(\widetilde{B^*})$. Denote $S^* \equiv S(K^*)$. By Lemma 2, S^* is a maximal clade of $\mathcal{T}^*$. Since $S' \notin \mathcal{T}^*$, we have $S' \neq S^*$, and therefore, $S^* \supset S'$. But $\mathcal{T}^* \subseteq \mathcal{T}_{alg}$, implying that S^* is also a non-trivial clade of $\mathcal{T}_{alg}$, in contradiction to the maximality of S'. □

3.2 Determining the Existence of a General Solution

We give in this section a characterization of IDP instances that admit a general solution. We also provide an algorithm to determine whether the solution tree $\mathcal{T}$ returned by Solve_IDP is general. The complexity of the latter algorithm is shown to be $O(mn + |E_1|d)$, where d is the maximum out-degree of $\mathcal{T}$.

Let $\mathcal{A}$ be a 'yes' instance of IDP. Consider a recursive call Solve_IDP$(\mathcal{A}')$ nested within Solve_IDP$(\mathcal{A})$, where $\mathcal{A}' = \mathcal{A}|_{C',S'}$. Let $K_1, \ldots, K_r$ be the connected components of $G(\widetilde{\mathcal{A}'})$ computed in Step 1c. Observe that $S(K_1), \ldots, S(K_r)$ are clades to be reported by recursive calls launched during Solve_IDP$(\mathcal{A}')$. A set U of characters is said to be (K_i, K_j)-*critical* if: (1) Characters in U are both $S(K_i)$-semi-universal and $S(K_j)$-semi-universal in $\mathcal{A}'$; and (2) removing U from $G(\widetilde{\mathcal{A}'})$ disconnects $S(K_i)$. Note that by definition of U, $U \subseteq W_{\mathcal{A}'}(S(K_i))$, and $a'_{sc} =?$ for all $c \in U, s \in S(K_j)$. A clade $S(K_i)$ is called *optional* with respect to $\mathcal{A}$, if $r \geq 3$ and there exists a (K_i, K_j)-critical set for some index $j \neq i$. Otherwise, we say that $S(K_i)$ is *supported*. In the example of Figure 3 (bottom), $K_1 = \{s_1, s_2, c_1\}, K_2 = \{s_3\}, K_3 = \{s_4, s_5, c_2\}$. The set $U = \{c_1\}$ is (K_1, K_2)-critical, so $S(K_1) = \{s_1, s_2\}$ is optional.

Theorem 4. *Let $\mathcal{T}_{alg}$ be a tree produced by Solve_IDP on instance $\mathcal{A}$. Then $\mathcal{T}_{alg}$ is the general solution for $\mathcal{A}$ if and only if all its clades are supported.*

Proof. ⇒ Suppose to the contrary that $\mathcal{T}_{alg}$ contains an optional clade with respect to $\mathcal{A}$. W.l.o.g., assume it is maximal, i.e., during the recursive call Solve_IDP$(\mathcal{A})$, $G' = G(\widetilde{\mathcal{A}})$ has $r \geq 3$ connected components, $K_1, \ldots, K_r$, and there exists a (K_i, K_j)-critical set U (for some $1 \leq i \neq j \leq r$). Let $\mathcal{A}_i, \mathcal{A}_j$ and $\mathcal{A}_{ij}$ be the sub-instances induced on K_i, K_j and $K_i \cup K_j$, respectively. Consider the tree $\mathcal{T}'$ which is produced by a small modification to the execution of Solve_IDP$(\mathcal{A})$: Instead of recursively invoking Solve_IDP$(\mathcal{A}_i)$ and

Solve_IDP($\mathcal{A}_j$), call Solve_IDP($\mathcal{A}_{ij}$). Then $\mathcal{T}'$ is a phylogenetic tree which explains $\mathcal{A}$, but $\mathcal{T}'$ includes the clade $S(K_i \cup K_j)$. Since $r \geq 3$, $S(K_i \cup K_j)$ is non-trivial and is not a clade of $\mathcal{T}_{alg}$, a contradiction.

$\Leftarrow$ Suppose that $\mathcal{T}_{alg}$ is not the general solution for $\mathcal{A}$, i.e., there exists a solution $\mathcal{T}^*$ of $\mathcal{A}$ such that $\mathcal{T}_{alg} \not\subseteq \mathcal{T}^*$. We shall prove the existence of an optional clade with respect to $\mathcal{A}$. (The reader is referred to the example in Figure 4 for notation and intuition. The example follows the steps of the proof, leading to the identification of an optional clade.) Let $\mathcal{B}^*$ be a completion of $\mathcal{A}$ which is explained by $\mathcal{T}^*$, and denote $G^* = G(\mathcal{B}^*)$. Let $S' \in \mathcal{T}_{alg} \setminus \mathcal{T}^*$ be the largest clade reported by Solve_IDP which is not a clade of $\mathcal{T}^*$. W.l.o.g. (as argued in the proof of Theorem 3), S' is a maximal clade of $\mathcal{T}_{alg}$, and $S' = S(K_1)$, where $K_1, \ldots, K_r$ are the connected components of $G(\widetilde{\mathcal{A}})$.

Let $\{S_i^*\}_{i=1}^t$ be the nested set of clades in $\mathcal{T}^*$ that contain S': $S = S_1^* \supset \cdots \supset S_t^* \supset S'$. For each $i = 1, \ldots, t$, let C_i^* be the set of characters in $\mathcal{B}^*$ whose 1-set is non-empty and is properly contained in S_i^*. Denote $\mathcal{B}_i^* = \mathcal{B}^*|_{S_i^*, C_i^*}$ and let $H_i^* = G(\mathcal{B}_i^*)$, i.e., H_i^* is the subgraph of G^* induced on $S_i^* \cup C_i^*$. Let H_i be the subgraph of $G(\mathcal{A})$ induced on the same vertex set. Since G^* is a supergraph of $G(\mathcal{A})$, each H_i^* is a supergraph of H_i.

Claim. S' is disconnected in H_t^* and, therefore, also in H_t.

Proof. Suppose to the contrary that S' is contained in some connected component K^* of H_t^*. $S(K^*)$ is thus a clade of the (unique) phylogenetic tree for $\mathcal{B}_t^*$ and, therefore, also a clade of $\mathcal{T}^*$. It follows that $S_t^* \supset S(K^*) \supset S'$, where the first containment follows from the fact that H_t^* is disconnected, and the second from the assumption that S' is not a clade of $\mathcal{T}^*$. This contradicts the minimality of S_t^*. □

We now return to the proof of Theorem 4. Recall that S' is connected in $H_1 = G(\widetilde{A})$. Thus, the previous claim implies that $t > 1$. Let K_p be a connected component of $G(\widetilde{\mathcal{A}})$ such that $S(K_p) \subseteq S \setminus S_2^*$. Such a component exists since $G(\widetilde{\mathcal{A}})$ is not connected and S_2^* is the species set of one of its components. Let l be the minimal index such that there exists some connected component K_i of $G(\widetilde{\mathcal{A}})$ for which $S(K_i)$ is disconnected in H_l. l is properly defined as $S(K_1) = S'$ is disconnected in H_t. $l > 1$, since otherwise some K_i is disconnected in H_1 and, therefore, also in its subgraph $G(\widetilde{\mathcal{A}})$, in contradiction to the definition of $K_1, \ldots, K_r$. By minimality of l, $S_l^* \supseteq S(K_i)$. Also, $S_l^* \supseteq S_t^* \supset S' = S(K_1)$, so $S_l^* \neq S(K_i)$. We now claim that there exists some connected component K_j of $G(\widetilde{\mathcal{A}})$, $j \neq i$, such that $S(K_j) \subseteq S_l^*$. Indeed, if $i \neq 1$ then $j = 1$. If $i = 1$ then $l = t$ (by an argument similar to that in the proof of Claim 3.2), and since $S_l^* \setminus S'$ is non-empty, it intersects $S(K_j)$ for some $j \neq i$. By minimality of l, $S(K_j)$ is properly contained in $S_l^* \setminus S'$.

Define $U \equiv W_{G^*}(S_l^*)$. By definition all characters in U are S_l^*-universal in G^*, and are thus both K_i-semi-universal and K_j-semi-universal in $\mathcal{A}$. $S(K_i)$ is disconnected in $H_l = G(\mathcal{A}|_{C_l^*, S_l^*})$. Since K_i is a connected component of $G(\widetilde{\mathcal{A}})$, $S(K_i)$ is disconnected in $G(\mathcal{A}|_{C_l^*, S})$, implying that U is a (K_i, K_j)-critical set.

Also, K_i, K_j and K_p are distinct, implying that $r \geq 3$. Hence, U demonstrates that $S(K_i)$ is optional. □

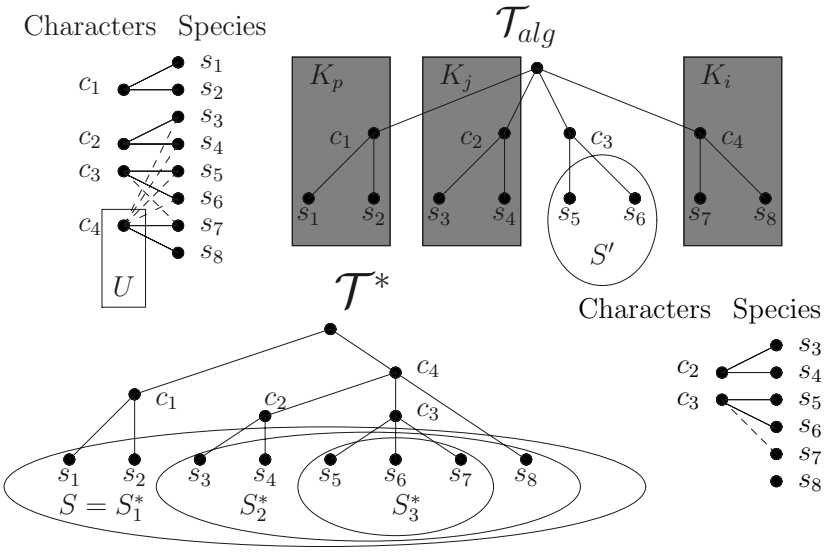


Fig. 4. An example demonstrating the proof of the 'if' part of Theorem 4, using the same notation. Left: A graphical representation of an input instance $\mathcal{A}$. Dashed lines denote $E_?$-edges, while solid lines denote E_1-edges. Top right: The tree $\mathcal{T}_{alg}$ produced by Solve_IDP. Bottom middle: A tree $\mathcal{T}^*$ corresponding to a completion $\mathcal{B}^*$ that uses all the edges in $E_?$. Bottom right: The graphs H_2 (solid edges) and H_2^* (solid and dashed edges). $\mathcal{T}_{alg} \not\subseteq \mathcal{T}^*$, and $S' = \{s_5, s_6\}$. There are $t = 3$ clades of $\mathcal{T}^*$ which contain S': $S_1^* = \{s_1, \ldots, s_8\}$, $S_2^* = \{s_3, \ldots, s_8\}$, and $S_3^* = \{s_5, s_6, s_7\}$. The component $K_p = \{c_1, s_1, s_2\}$ has its species in $S \setminus S_2^*$. Since $W_{\mathcal{A}}(S) = W_{\mathcal{B}^*}(S) = \emptyset$, $H_1 = G(\mathcal{A})$. Since $W_{\mathcal{B}^*}(S_2^*) = \{c_4\}$, the species set of the connected component $K_i = \{s_7, s_8, c_4\}$ is disconnected in H_2, implying that $l = 2$. For a choice of $K_j = \{s_3, s_4, c_2\}$, the set $U = \{c_4\}$ is (K_i, K_j)-critical, demonstrating that S' is optional.

The characterization of Theorem 4 leads to an efficient algorithm for determining whether a solution $\mathcal{T}_{alg}$ produced by Solve_IDP is general.

Theorem 5. *There is an $O(nm+|E_1|d)$-time algorithm to determine if a given solution $\mathcal{T}_{alg}$ is general, where d is the maximum out-degree in $\mathcal{T}_{alg}$.*

Proof. The algorithm simply traverses $\mathcal{T}_{alg}$ bottom-up, searching for optional clades. For each internal node x visited, whose children are $y_1, \ldots, y_{d(x)}$, the algorithm checks whether any of the clades $L(y_1), \ldots, L(y_{d(x)})$ is optional. If an optional clade is found the algorithm outputs *False*. Correctness follows from Theorem 4.

For analyzing the complexity, it suffices to show how to check whether a clade $L(y_i)$ is optional. If $d(x) = 2$, or y_i is a leaf, then certainly $L(y_i)$ is supported.

Otherwise, let U_i be the set of characters whose associated clade (in $\mathcal{T}_{alg}$) is $L(y_i)$. Let U_j^i denote the set of characters in U_i which are $L(y_j)$-semi-universal, for $j \neq i$. The computation of U_j^i for all i and j takes in total $O(nm)$ time, since for each character c and species s we check at most once whether $(s, c) \in E_?^{\mathcal{A}}$, for an input instance $\mathcal{A}$.

It remains to show how to efficiently check whether for some j, U_j^i disconnects $L(y_i)$ in the appropriate subgraph encountered during the execution of Solve_IDP. To this end, we define an auxiliary bipartite graph H^i whose set of vertices is $W_i \cup U_i$, where $W_i = \{w_1, \ldots, w_{d(y_i)}\}$ is the set of children of y_i in $\mathcal{T}_{alg}$. We include the edge (w_r, c_p) in H^i, for $w_r \in W_i, c_p \in U_i$, if $(c_p, s) \in E_1^{\mathcal{A}}$ for some species $s \in L(w_r)$. We construct for each $j \neq i$ a subgraph H_j^i of H^i induced on $W_i \cup (U_i \setminus U_j^i)$. All we need to report is whether H_j^i is connected. It can be shown that the overall complexity of the algorithm is $O(mn + |E_1^{\mathcal{A}}| \cdot \max_{v \in \mathcal{T}_{alg}} d(v))$. □

Acknowledgments. The first author was supported by the Clore foundation scholarship. The second author was supported in part by the Israel Science Foundation (grant number 565/99). The third author was supported by an Eshkol fellowship from the Ministry of Science, Israel.

References

1. C. Benham, S. Kannan, M. Paterson, and T.J. Warnow. Hen's teeth and whale's feet: generalized characters and their compatibility. *Journal of Computational Biology*, 2(4):515–525, 1995.
2. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
3. M. Henzinger, V. King, and T.J. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24:1–13, 1999.
4. C. A. Meecham and G. F. Estabrook. Compatibility methods in systematics. *Annual Review of Ecology and Systematics*, 16:431–446, 1985.
5. M. Nikaido, A. P. Rooney, and N. Okada. Phylogenetic relationships among cetartiodactyls based on insertions of short and long interspersed elements: Hippopotamuses are the closest extant relatives of whales. *Proceedings of the National Academy of Science USA*, 96:10261–10266, 1999.
6. I. Pe'er, R. Shamir, and R. Sharan. Incomplete directed perfect phylogeny. In *Eleventh Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, pages 143–153, 2000.

Sorting with a Forklift

M.H. Albert and M.D. Atkinson

Department of Computer Science, University of Otago

Abstract. A fork stack is a stack that allows pushes and pops of several items at a time. An algorithm to determine which sequences of input streams can be sorted by a fork stack is given. The minimal unsortable sequences are found (there are a finite number only). The results are extended to fork stacks where there are bounds on how many items can be pushed and popped at one time. Some enumeration results for the number of sortable sequences are given.

1 Introduction

A standard analogy for explaining the operation of a stack is to speak about stacks of plates, allowing one plate to be added to, or removed from, the top of the stack. At this point a disruptive member of the audience generally asks why it is not possible to move more than one plate at a time from the top of a stack. In this paper we address his concerns.

In particular we will consider the problem of a dishwasher and his helper. The dishwasher receives dirty plates, washes them, and adds them one at a time to a stack to be put away. The helper can remove plates from the stack, but she can move more than one plate at a time. It so happens, that all the plates are of slightly differing sizes, and her objective is to make sure that when they are placed in the cupboard, they range in order from biggest at the bottom, to smallest at the top. It is easy to see that if the dishwasher processes three dishes in order: middle, smallest, largest then his helper can easily succeed in the model given (simply waiting for the largest plate, and then moving the two smaller ones on top as a pair). However, if she is replaced by a small child who can only move a single plate at a time, then the desired order cannot be achieved.

The characterisation of those permutations that can be sorted using the standard single push and pop operations of a stack is a classical result found in [6]. Similar characterisations for sorting using other data structures are considered in [2], [7], and [9] among others. In this paper we will consider the corresponding problems when we allow multiple additions to, or removals from the stack according to the analogy above.

An alternative, slightly more general, analogy provides the source of our title. We begin with a stack of boxes, called the input, labelled 1 through n in some order. We have at our disposal a powerful forklift which can remove any segment of boxes from the top of the stack, and move it to the top of another stack, the working stack. From there another forklift can move the boxes to a final, output, stack. Physical limitations, or union rules, prevent boxes being moved from the

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 368–377, 2002.

working stack to the input, or from the output to the working stack. The desired outcome is that the output should be ordered with box number 1 on top, then 2, then 3, ..., with box n at the bottom. An example of a sorting procedure in progress is shown in Figure 1.

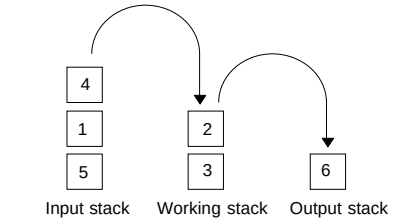


Fig. 1. A snapshot of sorting

The problems we wish to consider in this context are:

- How should such permutations be sorted?
- Which permutations can be successfully sorted?
- How many such permutations are there?

We will also consider these questions in the restricted context where one or both of the moves allowed are of limited power – for example, say, at most three boxes at a time can be moved from the input stack to the working stack, and at most six from the working stack to the output stack. The case where both moves are restricted to a single box, is, or course, the problem discussed above of sorting a permutation using a stack. We will omit most proofs entirely, or give a brief discussion of them – they will appear in subsequent papers.

The process of sorting 236415 is documented below. Note that, at the stage shown in Figure 1 it is essential that 41 be moved as a pair – moving either 4 alone, or the triple 415 would result (eventually) in 1 lying on top of 4 or 5 in the working stack, and thereby prevent sorting.

Input	Working	Output	
236415			
6415	23		
415	623		
415	23	6	(See Figure 1)
5	4123	6	
	54123	6	
	4123	56	
	123	456	
		123456	Finished

Some permutations, such as 35142 cannot be sorted. Here, we may move 3 to the working stack, and then 5 to the output, but now whether we move 1 alone, 14, or 142, we wind up with 1 lying on top of 3 or 4 in the working stack, and cannot complete the sorting procedure. We will see below that if we can avoid creating this type of obstruction in the working stack, then sorting is possible.

2 Definitions and Formalities

In the subsequent sections we will tend to continue to use the terminology of the introduction speaking of the input stack, forklifts, etc. However, it will be convenient to introduce a certain amount of basic notation in order to facilitate discussion. As we will always take the initial input to be a permutation of 1 through n for some n, the contents of each stack at any time can and will be represented by sequences of natural numbers (not containing repetitions). Our ultimate objective is always to reach a state where the contents of the output stack are the permutation

$$1\,2\cdots(n-1)\,n$$

and we will refer to this outcome as success.

In the basic situation where both forklifts are of unlimited capacity, we use $\mathcal{F}$ to denote the collection of all permutations for which success is possible. If the input to working stack forklift is limited to moving j boxes in a single move, and the working to output one to moving k, then we denote the corresponding class $\mathcal{F}(j,k)$. Here j and k are either natural numbers, or ∞. We do not concern ourselves with the case where either of the forklifts is broken and incapable of making any moves!

Given a permutation π as input, a sequence of operations is *allowed*, if it does not result in an output state which provides clear evidence that sorting is not being carried out. That is, a sequence of operations is allowed if at the end of the sequence the output stack contains some tail of $12\cdots n$.

Finally, in discussing the algorithms for sorting it will be useful to pretend that it is possible to move boxes directly from the input stack to the output stack – and such an operation, as well as the more normal type of output is called *direct output*. So a direct output move consists either of output from the working stack, or moving a part of the input stack to the working stack (in a single lift), and then moving exactly that set of boxes to the output stack, again in a single lift.

Just as the case where two forklifts of unit capacity corresponds to sorting with a stack, the general case we are considering corresponds to sorting with a data structure which has a more powerful set of operations than a normal stack. Namely, we are allowed to push a sequence of objects onto the top of the structure, and likewise pop such a sequence. This structure will be called a *fork stack* (and roughly it corresponds to the working stack in our analogy).

3 The Sorting Algorithms

How should a fork stack actually carry out its task of sorting a permutation when this is possible? It turns out that there is a straightforward algorithm to accomplish this operation. Broadly speaking, we may use a simple modification of a greedy algorithm:

- perform any output as soon as possible,
- otherwise move the maximum decreasing sequence from the head of the input onto the working stack.

Before we justify this claim (and make some technical changes to the second option) we need a slightly more abstract characterisation of unsortability.

Definition 1. *For positive integers a and b, $a << b$ means $a < b - 1$. In a series of fork stack moves, we say that the* dreaded 13 *occurs if at some point the working stack contains adjacent elements ab with $a << b$.*

Proposition 2. *A permutation π is unsortable if and only if every allowable sequence of fork stack operations that empties the input produces, at some point, the dreaded 13.*

This is easily justified. If we cannot avoid producing a 13, then we cannot sort π for there is no way to insert the missing elements into the gap between the elements $a << b$ witnessing the 13. On the other hand, if there is some allowable sequence of operations that empties the input stack and avoids producing a 13, then on completing them, the contents of the working stack will be a *decreasing sequence, except possibly for some blocks of consecutive increasing elements*. Such a stack is easily moved to the output in its sorted order.

We refer to a sequence of the type emphasised above, as a *near-decreasing* sequence. Suppose that no immediate output is possible and consider the maximal near-decreasing sequence at the top of the input stack. If the symbols occurring in this sequence do not form an interval, then it is easy to see that any move other than taking the whole sequence and transferring it to the top of the working stack will eventually cause the dreaded 13. If the symbols occurring in the sequence do form a consecutive interval, then we can arrange to place them on the working stack in order, with largest deepest (to subsequently be moved as a block to the output). This is preferable to any other arrangement on the working stack, for it makes the top element of the working stack as small as possible, minimising the possibility of later creating a dreaded 13.

Doing direct output as soon as it becomes available can never interfere with sorting. For if we have a successful sequence of sorting moves which we modify by doing some direct output earlier, we can simply continue to carry out the successful sequence, ignoring any effect on symbols which have already been moved to output – and we will still succeed. So we may assume that any sorting algorithm does in fact perform direct output whenever it can. Then the observations of the preceding paragraph imply that when direct output is not available, the

maximal near-decreasing sequence at the top of the input stack must be moved. If this sequence contains gaps, there is no choice in how to move it, and we have argued that if it does not, then moving it so that it forms an increasing sequence on the working stack is at least as effective as any other choice.

This establishes that Algorithm 1 will correctly sort any input stack, if it is sortable at all:

Algorithm 1 Sorting with a powerful fork-lift

repeat
  Perform as many direct output moves as possible.
  Move the maximal near-decreasing sequence from the top of the input stack to the working stack, as a block if it contains gaps, so that it becomes increasing if it does not.
until input stack is empty
if working stack is empty **then**
  Success!
else
  Failure.
end if

How does Algorithm 1 need to be modified in the case where either or both of the forklifts moving from input to working stack, or from working stack to output, are of limited power? The first issue is how to modify Proposition 2. The 13 configuration is bad regardless of the power of our forklifts, but if our output lift is limited to moving k boxes we must add the condition that the working stack should not contain an increasing sequence of length longer than k. Now modifying the algorithm is straightforward. In the case where the maximal near-decreasing sequence contains gaps it must be moved as a block to avoid 13's. So, if this block is larger than the capacity of our working forklift, we fail. In the non-gap case, we would normally attempt to make the sequence increasing. Of course this would be foolish if it overwhelmed the capacity of our output lift (and it could be impossible depending on the capacity of our input lift). The only other choice that does not create a 13 is to make it decreasing, so this should be attempted if the first choice is unavailable. Failure may later occur because we create a block that is too long to move in the working stack, or a 13 there, but if not, then the algorithm will succeed.

4 Finite Basis Results

We now begin our combinatorial investigation of the collections of permutations sortable by various combinations of forklifts. The problem which we address in this section is how to identify the sortable or unsortable permutations without reference to Algorithm 1. In the following section we will consider the problem

of enumerating these classes. For identification purposes we concentrate on producing a list of minimal unsortable permutations. We must first prepare the way with some definitions and notation:

Definition 3. *Given permutations σ and π, we say that σ is* involved in *π, and write $\sigma \preceq \pi$ if some subsequence of π, of the same length as σ, consists of elements whose relative order agrees with those of the corresponding elements of σ. A collection of permutations closed downwards under $\preceq$ is called a* closed class.

It is easy to see that each of the collections $\mathcal{F}(j,k)$ of sortable permutations for a particular combination of forklifts is a closed class. This is because we may sort any subsequence of a sortable sequence by simply ignoring any moves that do not affect members of the subsequence. This policy cannot increase the load on a forklift in any single move, so it still sorts the remaining elements. It follows, that if we take $U(j,k)$ to be the set of $\preceq$-minimal unsortable permutations then:

$$\pi \text{ is } (j,k)\text{-unsortable} \iff \sigma \preceq \pi \text{ for some } \sigma \in U(j,k).$$

In particular, $U(j,k)$ can be thought of as a description of $\mathcal{F}(j,k)$ (or rather of its complement, but that amounts to the same thing!) This description gains some power owing to the following result.

Proposition 4. *For any $1 \leq j,k \leq \infty$ the set $U(j,k)$ is finite.*

The detailed proof of this proposition is technical and dull. It is of course enough to show that there is some finite set of permutations such that every (j,k)-unsortable permutation involves one of them. The basic idea is to make use of the characterisation of unsortability provided by the failure of Algorithm 1 (modified as necessary for the limited power case). One considers the state of the working stack, and input, at the instant where the next move being attempted either fails completely or creates a bad configuration on the working stack. The first case corresponds to a near-decreasing part which is too long for the working lift. Obviously there are a finite number of minimal examples of such, and any occurrence of one of these prevents sorting. In the second case, the cause of the resulting bad configuration can be localised to a bounded number of elements of the original input. This gives a further finite set of obstructions, from which the desired result follows.

The actual sets $U(j,k)$ are not that difficult to compute. We may assume that $k \geq j$, since the class $\mathcal{F}(j,k)$ consists of the inverses of the elements of $\mathcal{F}(k,j)$, and inversion preserves the relation $\preceq$. For the case $k > j$, once we know $U(j,\infty)$, we obtain $U(j,k)$ simply by adding the single permutation:

$$(k+1)\,k\,(k-1)\cdots 2\,1\,(k+2),$$

and then deleting any elements of $U(j,\infty)$ in which it is involved. For $j = k$, direct computation is required.

The set $U(\infty, \infty)$ consists of the permutation 35142, together with 45 permutations of length six, and 6 of length seven. The sets $U(1, k)$ are of particular interest in connection with the next section and they are:

$$\begin{aligned} U(1,\infty) &= \{2314, 3124, 3142\} \\ U(1,k) &= \{2314, 3124, 3142, (k+1)\,k\,(k-1)\cdots 2\,1\,(k+2)\} \quad (k \geq 2) \\ U(1,1) &= \{213\}. \end{aligned}$$

5 Enumeration

Associated with any collection $\mathcal{C}$ of permutations is a generating function

$$f_{\mathcal{C}} = \sum_{n=0}^{\infty} c_n x^n = c_0 + c_1 x + c_2 x^2 + \cdots$$

where

$$c_n = |\{\pi \in \mathcal{C} : \pi \in S_n\}|.$$

Most often one sets $c_0 = 1$, though algebraic convenience may occasionally dictate $c_0 = 0$. Recurrences which define the sequence c_n often translate naturally into algebraic or differential equations for the generating function, and indeed it is frequently more illuminating to develop these equations directly rather than via an initial recurrence (for many examples and an exposition of the theory see [5] and [10]). Additionally, having an algebraic description of the generating function, possibly only impicit, allows one to use methods based on complex analysis to provide asymptotic expansions for the numbers c_n as explained in [3] and [4]. We will pursue this program for the classes $\mathcal{F}(1, k)$.

In particular, begin with the class $\mathcal{F}(1, \infty)$. Note that the sorting algorithm is completely determined – single elements are moved from input to working stack, and output is performed whenever possible. Suppose that we have some sortable permutation π. Choose t to be the maximum integer such that the elements 1 through t occur in π in decreasing order (thus, if 2 follows 1, $t = 1$). So the original input was of the form:

$$\pi = \sigma_t\, t\, \sigma_{t-1}\, (t-1) \cdots \sigma_2\, 2\, \sigma_1\, 1\, \sigma_0$$

for some sequences σ_0 through σ_t, where $t+1$ does not occur in σ_t.

Consider the sorting procedure. The elements of σ_t are processed, and then we come to t. Now by the choice of t, $t+1$ has not yet been processed, so we may not output t (except in the trivial case where all the σ_i are empty). So we must move t to the working stack. However, if it is non-empty at this time, that move would create a 13. So the working stack must be empty, and σ_t must have been a sortable permutation of a final subinterval of the values occurring in π. Now proceed to the stage where $t-1$ is about to be moved. Again, either $t+1$ has turned up by now, and the working stack is empty, or it contains only the value t. In either case σ_{t-1} is a sortable permutation of a final subinterval of

the remaining values. This argument persists inductively. So in the end we see that $t+1$ occurs in the first non-empty σ_j, and that the general structure of a sortable permutation is:

$$(\text{sortable})\, t\, (\text{sortable})\, (t-1) \cdots (\text{sortable})\, 1\, (\text{sortable})$$

where we are free to decide the sizes of the "sortable" parts, but having done so, their elements are uniquely determined. We distinguish two cases according to whether or not the final "sortable" part is empty. By standard arguments this yields an identity satisfied by the generating function for this class which we denote f_∞:'

$$f_\infty = 1 + x f_\infty + (f-1) \sum_{t=1}^{\infty} x^t f_\infty^t$$

or, after summing the geometric series:

$$f_\infty = 1 + \frac{x f_\infty^2 - x^2 f_\infty^2}{1 - x f_\infty}.$$

We can then solve the resulting quadratic to get:

$$f_\infty = \frac{1 + x - \sqrt{1 - 6x + 5x^2}}{4x - 2x^2} = \frac{2}{1 + x + \sqrt{1 - 6x + 5x^2}}.$$

The sequence that this generating function defines:

$$1,\, 1\,,2,\, 6,\, 21,\, 79,\, 311,\, 1265,\, 5275 \ldots$$

is number A033321 in [8], and the references provided for it there connect it with other interesting enumeration problems.

The only change that needs to be made to find the generating function f_k for $\mathcal{F}(1,k)$ is to change the upper limit of summation in the relationship above from ∞ to k, since the maximum increasing sequence that we can deal with on the working stack is of length k. This allows efficient exact enumeration of these classes using standard generating function techniques. It also allows asymptotic expansions of the form:

$$c_n = \frac{r^{-n}}{n^{3/2}} \left(\sum_{k=0} \frac{e_k}{n^k} \right)$$

to be computed to any desired degree of accuracy using the methods developed in [4].

The behaviour of the radius of convergence r (whose reciprocal gives the exponential part of the growth rate for the coefficients c_n), as k increases from 1 to ∞ is particularly interesting. It begins at $1/4$, since $k=1$ gives us the Catalan numbers and then decreases to $1/5$ at $k=\infty$. However, the rate of convergence to $1/5$ is very rapid indeed. The first six values are:

$$.2500,\ .2114,\ .2033,\ .2010,\ .2003,\ .2001$$

These observations can be formalized and justified using the fact that the radius of convergence in each case is the smallest positive root of the resultant of the polynomial that f_k satisfies (whose coefficients are linear functions of x). However, the intuitive content of the result is that once you have a forklift capable of moving six boxes, you gain little in terms of which initial configurations are sortable by the addition of more power!

6 Summary and Open Problems

We have defined a generalised form of stack depending on two parameters j, k. It allows multiple pushes of up to j elements at a time, and multiple pops of up to k elements. For each $1 \leq j, k \leq \infty$ we have shown how to test in linear time whether an input sequence is sortable, and we have determined the minimal set of unsortable sequences.

If either $j = 1$ or $k = 1$ we have precise enumeration results for the number of sortable permutations of every length.

The first open problem to attack is the enumeration question for $j, k \geq 2$. Based on counting valid sequences of fork lift operations we have some upper bounds for these questions. Suppose that we begin with n items of input. A valid sequence of forklift operations can be represented by a path (with marked vertices) from $(0, 0)$ to $(2n, 0)$ in the plane, consisting of segments of the form (s, s) or $(p, -p)$ for positive integers s and p, representing pushing s elements onto the working stack, or popping p elements from it respectively. The condition for validity is that the path never pass below the x-axis. Figure 2 illustrates the lattice path for sorting the sequence 236415.

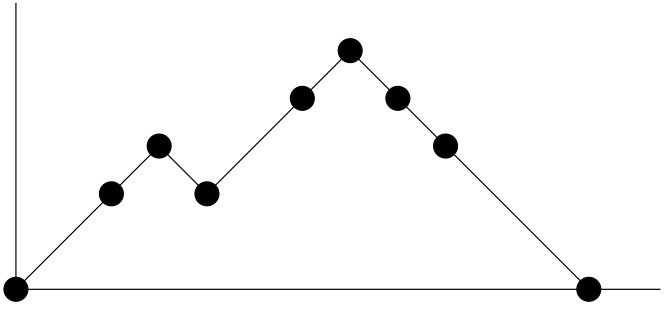


Fig. 2. The lattice path for sorting 236415

In the context of traditional stack sorting (where s and p are only allowed to equal 1) two different paths will produce two different permutations of the input. This is one of the methods for arguing that the stack-sortable permutations are enumerated by the Catalan numbers.

This same sort of analysis in fact applies to the case where only the push operation is restricted to be at most one element, and gives an alternative method of proving the enumeration results of the preceding section. However, when both s and p may take on multiple values, then it is no longer the case that two different paths produce different permutations. The simplest instance is with two input symbols 1 and 2 in that order. Then the paths:

$$(0,0) \to (2,2) \to (3,1) \to (4,0) \quad \text{and}$$
$$(0,0) \to (1,1) \to (2,2) \to (4,0)$$

both reverse the input. It is possible to reduce paths to a standard equivalent form, which reduces the problem to considering when two paths having the same outline represent the same permutation, but ambiguity is still present. None the less, by counting lattice paths we can certainly provide upper bounds for the number of sortable permutations. In particular for $j = k = \infty$, we get sequence A059231 from [8], which establishes that the number of sortable permutations of length n is $O(9^n)$.

References

1. M. D. Atkinson: Restricted permutations, Discrete Math. 195 (1999), 27–38.
2. M. D. Atkinson: Generalised stack permutations, Combinatorics, Probability and Computing 7 (1998), 239-246.
3. P. Flajolet and A. M. Odlyzko: Singularity analysis of generating functions. SIAM Jour. Disc. Math. 2 (1990), 216-240.
4. P. Flajolet and R. Sedgwick: The Average Case Analysis of Algorithms, Complex Asymptotics and Generating Functions. INRIA Research Report 2026, 1993.
5. I. P. Goulden, D. M. Jackson: *Combinatorial Enumeration*, John Wiley and Sons, New York, 1983.
6. D. E. Knuth: *Fundamental Algorithms, The Art of Computer Programming* Vol. 1 (First Edition), Addison-Wesley, Reading, Mass. (1967).
7. V. R. Pratt: Computing permutations with double-ended queues, parallel stacks and parallel queues, Proc. ACM Symp. Theory of Computing 5 (1973), 268–277.
8. N. J. A. Sloane: *The Online Encyclopedia of Integer Sequences*, `http://www.research.att.com/~njas/sequences/`, 2002.
9. R. E. Tarjan: Sorting using networks of queues and stacks, Journal of the ACM 19 (1972), 341–346.
10. H. S. Wilf: *generatingfunctionology*, Academic Press, New York, 1993.

Tree Decompositions with Small Cost*

Hans L. Bodlaender[1] and Fedor V. Fomin[2]

[1] Institute of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands, hansb@cs.uu.nl
[2] Graduiertenkolleg des PaSCo, Heinz Nixdorf Institut and University of Paderborn, Fürstenallee 11, D-33102 Paderborn, Germany, fomin@upb.de

Abstract. The f-cost of a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ for a function $f : \mathbf{N} \to \mathbf{R}^+$ is defined as $\sum_{i \in I} f(|X_i|)$. This measure associates with the running time or memory use of some algorithms that use the tree decomposition. In this paper we investigate the problem to find tree decompositions of minimum f-cost.
A function $f : \mathbf{N} \to \mathbf{R}^+$ is fast, if for every $i \in \mathbf{N}$: $f(i+1) \geq 2 \cdot f(i)$. We show that for fast functions f, every graph G has a tree decomposition of minimum f-cost that corresponds to a minimal triangulation of G; if f is not fast, this does not hold. We give polynomial time algorithms for the problem, assuming f is a fast function, for graphs that has a polynomial number of minimal separators, for graphs of treewidth at most two, and for cographs, and show that the problem is NP-hard for bipartite graphs and for cobipartite graphs.
We also discuss results for a weighted variant of the problem derived of an application from probabilistic networks.

1 Introduction

It is well known that many problems that are intractable on general graphs become linear or polynomial time solvable on graphs of bounded treewidth. These algorithms often have the following form: first a tree decomposition of small treewidth is made, and then a dynamic programming algorithm is used, computing a table for each node of the tree. The time to process one node of the tree is exponential in the size of the associated set of vertices of the graph; thus, when the maximum size of such a set is bounded by a constant (i.e., the width of the tree decomposition is bounded by a constant), then the algorithm runs in linear time. However, two different tree decompositions of the same graph with the same width may still give different running times, e.g., when one has many large vertex sets associated to nodes, while the other has only few large vertex sets associated to nodes.

In several applications, the same tree decomposition will be used for several successive runs of an algorithm, e.g., with different data. An important example of such an application is the PROBABILISTIC INFERENCE problem on probabilistic

* This research was partially supported by EC contract IST-1999-14186: Project ALCOM-FT (Algorithms and Complexity - Future Technologies).

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 378–387, 2002.

networks. (This application will be briefly discussed in Section 6.) Hence, in many cases it makes sense to do more work on finding a good tree decomposition, and to use a more refined measure on what is a 'good' tree decomposition. Suppose the time to process a node of the tree decomposition whose associated set has size k is $f(k)$. Then, processing a tree decomposition of the form $(\{X_i \mid i \in I\}, T = (I, F))$ costs $\sum_{i \in I} f(|X_i|)$ time. (For precise definitions, see Section 2.) We call this measure the *f-cost* of the tree decomposition; the treecost of a graph G with respect to f is the minimum f-cost of a tree decomposition of G. In other cases, the f-cost of the tree decomposition can represent the amount of space needed for the algorithm, in particular, the total size of all tables a specific dynamic programming algorithm uses with the tree decomposition. In this paper, we investigate the problem of finding tree decompositions of minimum f-cost.

It appears that it is important whether the function f satisfies a certain condition which we call *fast*: a function $f : \mathbf{N} \to \mathbf{R}^+$ is fast, if for every k, $f(k+1) \geq 2 \cdot f(k)$. Most applications of treewidth in our framework will have functions that are fast (in particular, many of the classical algorithms using tree decompositions for well known graph problems have fast cost functions.) To a tree decomposition we can associate a triangulation (chordal supergraph) of input graph G in a natural way. Now, every graph has a tree decomposition of minimum f-cost that can be associated with a *minimal* triangulation, if and only if f is fast. This will be shown in Section 3. This result for our proofs that that the problem of finding minimum f-cost tree decompositions can be solved in polynomial time for graphs that have a polynomial number of separators, and in linear time for graphs of treewidth at most two, assuming that f is fast and polynomial time computable (Section 4). There is also a linear time algorithm for the treecost of cographs, assuming that in linear time, one can compute $f(1), \ldots, f(n)$ (Section 4). In Section 5, we discuss a conjecture on the relation between triangulations of minimum f-cost and minimum treewidth, and show that for a fixed k, one can find a triangulation of minimum f-cost among those of treewidth at most k in polynomial time. A variant of the problems for weighted graphs with an application to probabilistic networks is discussed in Section 6. In Section 7, we show the unsurprising but unfortunate result that for each fast f, the TREECOST$_f$ problem is NP-hard for cobipartite graphs and for bipartite graphs. Also, in these cases there is no constant factor approximation algorithm, unless $P = NP$. Some final remarks are made in Section 8.

2 Preliminaries

We use the following notations: $G = (V, E)$ is an undirected and finite graph with vertex set V and the edge set E, assumed to be without self-loops or parallel edges. Unless otherwise specified, n denotes the number of vertices and m the number of edges of G. The *(open) neighborhood* of a vertex v in a graph G is $N_G(v) = \{u \in V : \{u, v\} \in E\}$ and the *closed neighborhood* of v is $N_G[v] = N_G(v) \cup \{v\}$. For a vertex set $S \subseteq V$ we denote $N_G[S] = \bigcup_{v \in S} N[v]$ and $N(S) = N[S] \setminus S$. If G is clear from the context, we write $N(v)$, $N[v]$, etc. $d_G(v) := |N_G(v)|$ is the degree of v in G. $G - v$ is the graph, obtained by removing v and its incident edges from G.

For a set $S \subseteq V$ of vertices of a graph $G = (V, E)$ we denote by $G[S]$ the subgraph of G induced by S. A set $W \subseteq V$ of vertices is a *clique* in graph $G = (V, E)$ if $G[W]$ is a complete graph, i.e. every pair of vertices from W induces an edge of G. A set $W \subseteq V$ of vertices is a *maximal clique* in $G = (V, E)$, if W is a clique in G and W is not a proper subset of another clique in G.

A *chord* of a cycle C is an edge not in C that has both endpoints in C. A *chordless cycle* in G is a cycle of length more than three that has no chord. A graph G is *chordal* if it does not contain a chordless cycle.

A *triangulation* of a graph G is a graph H on the same vertex set as G that contains all edges of G and is chordal. A *minimal* triangulation of G is a triangulation H such that no proper subgraph of H is a triangulation of G.

Definition 1. *A* tree decomposition *of a graph $G = (V, E)$ is a pair $(\{X_i \mid i \in I\}, T = (I, F))$, with $\{X_i \mid i \in I\}$ a family of subsets of V and T a tree, such that*

- $\bigcup_{i \in I} X_i = V$.
- *For all $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$.*
- *For all $i_0, i_1, i_2 \in I$: if i_1 is on the path from i_0 to i_2 in T, then $X_{i_0} \cap X_{i_2} \subseteq X_{i_1}$.*

The width *of tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ is $\max_{i \in I} |X_i| - 1$. The treewidth of a graph G is the minimum width of a tree decomposition of G.*

The following well known result is due to Gavril [5].

Theorem 1 ([5]). *Graph G is chordal if and only there is a* clique tree *of G, i.e. tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of G such that for every node i of T there is a maximal clique W of G such that $X_i = W$.*

Definition 2. *For a function $f : \mathbf{N} \to \mathbf{R}^+$, the f-*cost *of a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ is $\sum_{i \in I} f(|X_i|)$. The* treecost *with respect to f of a graph G is the minimum f-cost of a tree decomposition of G, and is denoted* $\mathrm{tc}_f(G)$.

Definition 3. *The f-*cost *of a chordal graph G is*

$$\mathrm{cost}_f(G) = \sum_{W \subseteq V; W \text{ is a maximal clique}} f(|W|)$$

We identify the following computational problem. Given a function $f : \mathbf{N} \to \mathbf{R}^+$, the TREECOST$_f$ problem is the problem, that given a graph $G = (V, E)$ and an integer K, decides whether $\mathrm{tc}_f(G) \leq K$. The proof of the following lemma is omitted.

Lemma 1. *The treecost of a graph G with respect to f equals the minimum f-cost of a chordal graph H that contains G as a subgraph.*

An interesting and important question is whether the treecost of a chordal graph equals its f-cost. We will see in Section 3 that this depends on the function f.

Definition 4. *A function $f : \mathbf{N} \to \mathbf{R}^+$ is* fast*, if for all $i \in \mathbf{N}$, $f(i+1) \geq 2 \cdot f(i)$.*

An example of a fast function is the function $f(i) = 2^i$.

Definition 5. *A tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of a graph $G = (V, E)$ is* minimal*, if there is no $\{i, j\} \in F$ with $X_i \subseteq X_j$.*

It is well known that there is always a minimal tree decomposition of minimum treewidth. Such a minimal tree decomposition can be obtained by taking an arbitrary tree decomposition of minimum width, and while there is an edge $\{i, j\} \in F$ with $X_i \subseteq X_j$, contracting this edge, taking for the new node i', $X_{i'} = X_i \cup X_j = X_j$. The same construction can also be obtained for obtaining a minimal tree decomposition of minimum f-cost.

3 Minimal Triangulations and Treecost

In this section, we investigate for which chordal graphs and which functions f, the treecost equals the f-cost. Using the obtained results, we will see that for every fast function f, there always exists a minimal triangulation with optimal f-cost.

Lemma 2. *Let $f : \mathbf{N} \to \mathbf{R}^+$ be a function that is not fast. Then there is a chordal graph G, such that the f-cost of G is larger than the treecost of G with respect to f.*

Proof. Suppose $f(i+1) < 2 \cdot f(i)$. Let G be the graph, obtained by taking a clique with $i+1$ vertices and remove one edge e. Then G has f-cost $2f(i)$, but the triangulation that is formed by adding the edge e has f-cost $f(i+1)$. □

The next two lemmas follow by observing which are the maximal cliques in the graphs G and $G - v$.

Lemma 3. *Let $f : \mathbf{N} \to \mathbf{R}^+$ be a function, and G be a chordal graph. Suppose v is a simplicial vertex in G, and suppose $N_G(v)$ is a maximal clique in the graph $G - v$. Then $\mathrm{cost}_f(G) = \mathrm{cost}_f(G - v) + f(d_G(v) + 1) - f(d_G(v))$.*

Lemma 4. *Let $f : \mathbf{N} \to \mathbf{R}^+$ be a function, and G be a chordal graph. Suppose v is a simplicial vertex in G, and suppose $N_G(v)$ is not a maximal clique in the graph $G - v$. Then $\mathrm{cost}_f(G) = \mathrm{cost}_f(G - v) + f(d_G(v) + 1)$.*

Lemma 5. *Suppose G and H are chordal graphs and G is a subgraph of H. Let v be a simplicial vertex in G. Let H' be the graph obtained from H by removing all edges incident to v that do not belong to G, i.e., $E_{H'} = E_H - \{\{v, w\} \mid w \notin N_G(v)\}$. Then H' is chordal.*

Proof. Consider a cycle in H' of length at least four. If the cycle contains v, then it has a chord between the vertices before and after v on the cycle, as these are neighbors of v in H' hence in G, and adjacent as v is simplicial v in G. If the cycle does not contain v, then it is a cycle in H and hence has a chord in H, which also is a chord in H'. □

Lemma 6. *Let $G = (V, E_G)$ and $H = (V, E_H)$ be chordal graphs, and $f : \mathbf{N} \to \mathbf{R}^+$ be a fast function. Suppose G is a subgraph of H. Then $\mathrm{cost}_f(G) \leq \mathrm{cost}_f(H)$.*

Proof. We use induction to $|V|$. Clearly, if $|V| = 1$, then $G = H$ and the result holds.

Suppose the result holds for graphs with up to $n-1$ vertices, and let G and H be chordal graphs with n vertices, with the same vertex set and $E_G \subseteq E_H$.

Take a vertex v that is simplicial in G. Let H' be the graph obtained from H by removing all edges, incident to v that do not belong to G, i.e., $E_{H'} = E_H - \{\{v, w\} \mid w \notin N_G(v)\}$. By Lemma 5 H' is chordal. Vertex v is a simplicial vertex in H' and $H' - v$ is chordal.

First, we show that $\text{cost}_{\text{f}}(G) \leq \text{cost}_{\text{f}}(H')$.

Claim. $\text{cost}_{\text{f}}(G) \leq \text{cost}_{\text{f}}(H')$.

Proof. v is also simplicial in H', and $H'-v$ is a chordal graph. Thus, by induction, $\text{cost}_{\text{f}}(G - v) \leq \text{cost}_{\text{f}}(H' - v)$,

Write $Z = N_G(v) = N_{H'}(v)$, and $d = d_G(v) = d_{H'}(v)$.

Note that if Z is not a maximal clique in $G - v$, then Z is a subset of some larger clique Z' in $G - v$. But, Z' also must be a clique in $H' - v$, and hence in this case Z is not a maximal clique in $H' - v$. It follows that Z is a maximal clique in $G - v$ or Z is not a maximal clique in $H' - v$.

Using Lemmas 3 and 4, we can observe the following. If Z is a maximal clique in $G - v$, then $\text{cost}_{\text{f}}(G) = \text{cost}_{\text{f}}(G - v) + f(d+1) - f(d)$ and $\text{cost}_{\text{f}}(H') \geq \text{cost}_{\text{f}}(H' - v) + f(d+1) - f(d)$, hence $\text{cost}_{\text{f}}(G) \leq \text{cost}_{\text{f}}(H')$. If Z is not a maximal clique in $H' - v$, then $\text{cost}_{\text{f}}(H') = \text{cost}_{\text{f}}(H' - v) + f(d+1)$, and $\text{cost}_{\text{f}}(G) \leq \text{cost}_{\text{f}}(G - v) + f(d+1)$, so again $\text{cost}_{\text{f}}(G) \leq \text{cost}_{\text{f}}(H')$. □

Claim. $\text{cost}_{\text{f}}(H') \leq \text{cost}_{\text{f}}(H)$.

Proof. When $d_H(v) = d_{H'}(v)$ the result holds trivially. Suppose that $d_H(v) > d_{H'}(v)$. So H is obtained from H' by adding one or more edges to vertex v. There is exactly one maximal clique in H' that contains v, namely $W_0 = N_{H'}(v) \cup \{v\}$. Suppose $W_0 \subseteq W_1$, with W_1 a maximal clique in H. We now have the following cases for sets W that form a maximal clique in H'. Each maximal clique in H' will be associated with a maximal clique in H.

- $W = W_0$. Associate W_0 with W_1.
- $v \notin W$, and $W \not\subseteq W_1$. Then, W is a maximal clique in H, or $W \cup \{v\}$ is a maximal clique in H, as H and H' differ only by some edges that have v as endpoint. Associate W with this maximal clique (i.e., W or $W \cup \{v\}$.)
- $v \notin W$ and $W \subseteq W_1$. As W_1 is a clique in H, the set $W_1 - \{v\}$ forms a clique in H and in H', so we must have $W = W_1 - \{v\}$. Associate W with W_1.

Every maximal clique W in H except for W_1 has exactly one maximal clique in H' associated with it, namely either W or $W - \{v\}$; we note that $f(|W - \{v\}|) < f(|W|)$. W_1 can have two maximal cliques in H' associated with it, namely W_0 and $W_1 - \{v\}$. There are two cases: if $W_0 = W_1$, then $W_1 - \{v\}$ is not a maximal clique in H', and it follows that $\text{cost}_{\text{f}}(H') \leq \text{cost}_{\text{f}}(H)$; if W_0 is a proper subset of W_1, then W_1 has exactly two maximal cliques associated with it, but both are of smaller size; we can use here that f is a fast function: $f(|W_0|) + f(|W_1 - \{v\}|) \leq 2f(|W_1| - 1) \leq f(W_1)$, and hence we have again $\text{cost}_{\text{f}}(H') \leq \text{cost}_{\text{f}}(H)$. □

Combining these two claims, we have $\text{cost}_f(G) \leq \text{cost}_f(H)$, which finishes the inductive proof of this lemma. □

Theorem 2. *Let $f : \mathbf{N} \to \mathbf{R}^+$ be a fast function. Every graph G has a minimal triangulation H, such that $\text{cost}_f(H) = \text{tc}_f(G)$.*

Proof. Suppose H' is a triangulation of G with $\text{cost}_f(H') = \text{tc}_f(G)$. H' contains a minimal triangulation H of G. Trivially, we have $\text{cost}_f(H) \geq \text{tc}_f(G)$. By the previous lemma, we have $\text{cost}_f(H) \leq \text{cost}_f(H')$. □

Corollary 1. *Let G be a chordal graph, and let f be a fast function. Then $\text{cost}_f(G) = \text{tc}_f(G)$.*

4 Treecost for Special Graph Classes

In this section, we discuss algorithms for computing the treecost for some special classes of graphs.

An important algorithmic consequence of Theorem 2 is that for fast functions the treecost of graphs with a polynomial number of minimal separators can be computed efficiently. Our approach to this problem follows the ideas of Bouchitté and Todinca [3]. (See also Parra and Scheffler [10].) This allows one to find the treecost efficiently when the input is restricted to cocomparability graphs, d-trapezoid graphs, permutation graphs, circle graphs, weakly triangulated graphs and many others graph classes. See [4] for an encyclopedic survey on graph classes.

A subset S of vertices of a connected graph G is called an *a, b-separator* for non adjacent vertices a and b in $V(G) \setminus S$ if a and b are in different connected component of the subgraph of G induced by $V(G) \setminus S$. If no proper subset of an a, b-separator S separates a and b in this way, then S is called a *minimal a, b-separator*. A subset S is referred to as a *minimal separator*, if there exist non adjacent vertices a and b for which S is a minimal a, b-separator. Notice that a minimal separator can be strictly contained in another minimal separator.

Let Δ_G be the set of all minimal separators in G. The proof of the following theorem is based on deep results and techniques of Bouchitté and Todinca [3]. We omit the proof of the theorem in this extended abstract.

Theorem 3. *Let f be a fast function and let $T_f(n)$ be the time needed to compute $f(1), \ldots, f(n)$. Then for every graph G there exists an $O(n^2|\Delta_G|^3 + T_f(n) + n^2 m|\Delta_G|^2)$ time algorithm for computing the treecost of G.*

From Theorem 2, it can be derived that graphs of treewidth at most two always have a triangulation of minimum f-cost that also has minimum treewidth (i.e., treewidth two), assuming that f is fast. Based upon this fact, one can obtain the following theorem, whose proof is omitted in this extended abstract.

Theorem 4. *Let f be a fast function, such that $f(1)$, $f(2)$, and $f(3)$ are computable. Then there is a linear time algorithm that computes the treecost with respect to f of a graph of treewidth at most two.*

From Theorem 3, it follows that cographs have a polynomial time algorithm for the treecost problem, assuming $f(1), \ldots, f(n)$ can be computed in polynomial time. Using techniques for computing the treewidth of a cograph from [2], one can obtain the following result. Note that f does not need to be fast.

Theorem 5. *Let $f : \mathbf{N} \to \mathbf{R}^+$ be a function. Let $T_f(n)$ be the time needed to compute $f(1), \ldots, f(n)$. Then there is an algorithm that computes $\mathrm{tc_f}(G)$ for a given cograph with n vertices and m edges in $O(n + m + T_f(n))$ time.*

5 Treewidth versus Treecost

An interesting question is whether there is always a triangulation with both optimal treecost and with optimal treewidth. Such a result would have had nice practical algorithmic consequences (e.g., in the algorithm of Theorem 3, we can ignore all separators larger than the treewidth plus one). Unfortunately, such triangulations do not always exist. For example, let G be a cograph that is formed as follows. G_1 is the disjoint union of four triangles (copies of K_3). G_2 is the disjoint union of a clique with four vertices and eight isolated vertices. G is the product of G_1 and G_2. Let f be the function $f(n) = 2^n$. Now, a triangulation of minimum treewidth is obtained by turning V_2 into a clique: this gives a maximum clique size of 15 (whereas when we turn V_1 into a clique, we have a triangulation with maximum clique size 16.) A triangulation of $G_1 \times G_2$ of minimum f-cost is obtained by turning V_1 into a clique: this gives an f-cost of $2^{12} \cdot (2^4 + 8)$; turning V_2 into a clique gives an f-cost of $2^{12} \cdot (4 \cdot 2^3)$.

More generally, let $\mathrm{tc}_{f,k}(G)$ be the minimum f-cost of a tree decomposition of G of width at most k. The cograph given above is an example of a graph where $\mathrm{tc}_{f,k}(G) \neq \mathrm{tc_f}(G)$, k the treewidth of G.

We conjecture that the width of a tree decomposition of optimal f-cost cannot be 'much' larger than the treewidth of a graph:

Conjecture 1. Let f be a fast function. There exists a function g_f, such that for all graphs G of treewidth at most k, $\mathrm{tc_f}(G) = \mathrm{tc}_{f,g_f(k)}(G)$.

Having such a function g_f would help to speed up the algorithm of Theorem 3. A proof of Conjecture 1 would imply that for every polynomial time computable fast function, the the treecost of graphs of bounded treewidth is polynomial time computable, because we have the following result.

Theorem 6. *Let $f : \mathbf{N} \to \mathbf{R}^+$ be function, such that for each n, $f(n)$ can be computed. Let $k \in \mathbf{R}^+$. There exists an algorithm that computes for a given graph G, $\mathrm{tc}_{f,k}(G)$ in $O(n^{k+2})$ time, plus the time needed to compute $f(1), \ldots, f(k+1)$.*

We omit the proof in this extended abstract.

There is also a constructive variant of the algorithm (it outputs the desired tree decomposition) that runs also in $O(n^{k+2})$ time.

6 Probabilistic Networks and Vertex Weights

Probabilistic networks are the underlying technology of several modern decision support systems. See e.g., [6]. Such a probabilistic network models independencies and dependencies between statistical variables with help of a directed acyclic graph. A central problem is the PROBABILISTIC INFERENCE problem: one must determine the probability distribution of a specific variable, possibly given the values of some other variables. As this problem is $\#P$-complete for general networks [11] but many networks used in practice appear to have small treewidth, an algorithm of Lauritzen and Spiegelhalter [9] is often used that solves the problem on networks with small treewidth.[1] As the same network is used for many computations, it is very useful to spend much preprocessing time and obtain a tree decomposition that allows fast computations. Thus, more important than minimizing the width is to minimize the 'cost' of the tree decomposition. While each vertex models a discrete statistical variable, variables may have a different valence. Let $w(v) \in \mathbf{N}$ be the *weight* of v. $w(v)$ models the number of values v can take, which directly reflects on the resources (time and space) needed for a computation. For instance, a binary variable corresponds to a vertex with weight two. In a tree decomposition of G, the time to process a node is basically the product of the weights of the vertices in the corresponding set X_i. In graph terms, we can model the situation as follows, after [8,12,7].

Given are a graph $G = (V, E)$, and a weight function $w : V \to \mathbf{N}$. The *total state space* of a triangulation H of G is the sum over all maximal cliques W in H of $\prod_{v \in W} w(v)$.

Note that when all vertices have weight two (i.e., all variables are binary), then the total state space is exactly the f-cost with for all i, $f(i) = 2^i$.

Some of the proofs and results of previous sections can be modified to give similar results for the problem to find a triangulation of minimum total state space.

Theorem 7. *(i) Let G be a graph, with vertices weighted with positive integers. Then there is a minimal triangulation H with total state space equal to the minimum total state space of a triangulation of G.*
(ii) There exists an algorithm to compute a triangulation with minimum total state space whose running time is polynomial in the number of minimal separators of G.
(iii) Given a cograph G with vertices weighted with positive integers, a triangulation of G with minimum total state space can be found in linear time.
(iv) For each k, there is an algorithm that runs in $O(n^{k+2})$ time, and that given a graph G with vertices weighted with positive integers, finds among the tree decompositions of G of width at most k finds one of minimum state space.

The method to compute the treecost of a graph of treewidth two of Theorem 4 cannot be used for the minimum state space problem when vertices have different weights.

[1] To be precise, first the moralization of the network is made: for every vertex, the set of its direct predecessors is turned into a clique, and then all directions of edges are dropped.

7 Hardness Results

Wen [12] showed that TREECOST$_f$ is NP-hard when f is the function $f(i) = 2^i$. To be precise, Wen showed that the problem of finding a triangulation of minimum total state space is NP-hard when all variables are binary. In this section, we show similar results for a larger class of functions f, using a different reduction, and we show that the problems remain NP-hard for cobipartite and for bipartite graphs.

We omit the proof of the following theorems in this extended abstract.

Theorem 8. *Let f be a fast function. The* TREECOST$_f$ *problem is NP-hard for bipartite and cobipartite graphs.*

Theorem 9. *If $P \neq NP$, then for every $c \in \mathbf{N}$, there is no polynomial time algorithm that approximates the treecost of a given graph G within a multiplicative factor c.*

8 Discussion

In this paper, we investigated a notion that gives a more refined view on what is a 'good' tree decomposition of a graph. For several algorithms on tree decompositions, the function that maps a tree decomposition to the amount of time spent by the algorithm when using that tree decomposition is actually somewhat more complicated than the f-costs as used in this paper, but the f-cost functions come close to these exact models. In addition, the f-cost often equals the total size of all tables computed by the algorithm, which in some cases equals the amount of space needed for the algorithm (discounting small additional overhead, like the pointers between the different nodes of the tree decomposition). Aspvall et al. [1] give techniques to reuse space; however, sometimes, we need to keep all tables to *construct solutions* corresponding to the computed optimal value. It may be worthwhile to study refined versions of treecost, as the time to process a node often depends on the sizes of the differences of its set with those of adjacent nodes in the tree, and/or its degree.

We have seen that in several interesting cases, tree decompositions with optimal f-cost can be computed in polynomial time, and we expect that in some practical cases, where it makes sense to spend sufficiently many preprocessing time on finding one good tree decomposition (in particular, in cases, where the same tree decomposition is used several times with different data on the same graph or network), some of our methods can be of practical use.

Acknowledgement. We thank Linda van der Gaag for discussions about probabilistic networks and introducing the notion of total state space to us.

References

1. B. Aspvall, A. Proskurowski, and J. A. Telle. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica*, 27:382–394, 2000.

2. H. L. Bodlaender and R. H. Möhring. The pathwidth and treewidth of cographs. *SIAM J. Disc. Math.*, 6:181–188, 1993.
3. V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: grouping the minimal separators. *SIAM J. Comput.*, 31(1), 2001.
4. A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph classes: a survey.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1999.
5. F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Comb. Theory Series B*, 16:47–56, 1974.
6. F. V. Jensen. *Bayesian Networks and Decision Graphs.* Statistics for Engineering and Information Science, Springer-Verlag, New York, 2001.
7. U. Kjærulff. Triangulation of graphs — algorithms giving small total state space. Research Report R-90-09, Dept. of Mathematics and Computer Science, Aalborg University, 1990.
8. U. Kjærulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2:2–17, 1992.
9. S. J. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *The Journal of the Royal Statistical Society. Series B (Methodological)*, 50:157–224, 1988.
10. A. Parra and P. Scheffler. How to use the minimal separators of a graph for its chordal triangulation. In *Automata, languages and programming (Szeged, 1995)*, pages 123–134. Springer, Berlin, 1995.
11. D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.
12. W. X. Wen. Optimal decomposition of belief networks. In P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer, editors, *Proceedings of the Sixth Workshop on Uncertainty in Artificial Intelligence*, pages 245–256, 1990.

Computing the Treewidth and the Minimum Fill-in with the Modular Decomposition

Hans L. Bodlaender[1] and Udi Rotics[2]

[1] Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, Utrecht, The Netherlands, hansb@cs.uu.nl
[2] School of Mathematics and Computer Science, Netanya Academic College
P.O. Box 120, 42100 Netanya, Israel, rotics@mars.netanya.ac.il

Abstract. Using the notion of modular decomposition we extend the class of graphs on which both the TREEWIDTH and the MINIMUM FILL-IN problems can be solved in polynomial time. We show that if $\mathcal{C}$ is a class of graphs which is modularly decomposable into graphs that have a polynomial number of minimal separators, or graphs formed by adding a matching between two cliques, then both the TREEWIDTH and the MINIMUM FILL-IN problems on $\mathcal{C}$ can be solved in polynomial time. For the graphs that are modular decomposable into cycles we give algorithms, that use respectively $O(n)$ and $O(n^3)$ time for TREEWIDTH and MINIMUM FILL-IN.

Keywords: Algorithms and data structures, graph algorithms, treewidth, minimum fill-in, modular decomposition, minimal separators.

1 Introduction

A graph is chordal if it does not contain a chordless cycle of length at least four as an induced subgraph. A triangulation of a graph is a chordal supergraph with the same vertex set. The *treewidth* of a graph G, denoted as *treewidth*(G) is the smallest clique number of all possible triangulations of G minus 1. The *minimum fill-in* of a graph G, denoted as *min-fill-in*(G), is the minimum of $|E(H) - E(G)|$ taken over all triangulations H of G. The TREEWIDTH problem is to find *treewidth*(G) for a given graph G. The MINIMUM FILL-IN problem is to find *min-fill-in*(G) for a given graph G. These problems have drawn much attention due to applications in areas such as Gaussian elimination of matrix, VLSI-layout, gate matrix layout and algorithmic graph theory (see e.g. [1,19]). Both problems are NP-hard in general [2,22] but polynomial time algorithms exist for many special graph classes such as: permutation graphs [4], circular arc graphs [21], circle graphs [16], distance hereditary graphs [10], $(q, q-4)$-graphs [3] and HHD-free graphs [9]. Bouchitté and Todinca [7,8] have shown that the treewidth and minimum fill-in of a graph can be computed in polynomial time if the graph has a polynomial number of minimal separators. This result generalizes several of the earlier results for special graph classes. In this paper we extend

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 388–397, 2002.

the class of graphs on which these two problems can be solved in polynomial time using the notion of modular decomposition as described below.

Let $V(G) = \{v_1, \ldots, v_r\}$ and let $H_1, \ldots, H_r$ be disjoint graphs. We denote by $G(H_1, \ldots, H_r)$ the graph G' obtained from G by substituting the graph H_i for the vertex v_i, for $1 \leq i \leq r$: $V(G') = V(H_1) \cup \ldots V(H_r)$, $E(G') = E(H_1) \cup \ldots E(H_r) \cup \{\{u, v\} | u \in V(H_i)$ and $v \in V(H_j)$ and $\{v_i, v_j\} \in E(G)\}$. This is called the modular decomposition operation; the graph G is called the *prime graph*, and $H_1, \ldots H_r$ are the *modules*. (In addition, we assume G cannot obtained by modular decomposition, except when using G itself as 'prime' graph.) A graph G is said to be modular decomposable into a class of graphs $\mathcal{D}$, if G consists of a single vertex or G can be obtained by modular decomposition with a prime graph in $\mathcal{D}$, and all modules also modular decomposable into $\mathcal{D}$. For more details on the modular decomposition of graphs, see for example [11,12,14,18]. The set of prime graphs that are used in the modular decomposition of G is denoted by $\pi(G)$. The modular decomposition of a graph is unique and can be found in linear time [12,18]. We denote by n and m the number of vertices and edges of a graph, respectively. We call a graph a clique-matching graph if it can be obtained by taking two cliques with the same number r of vertices and then adding a matching with r edges between the cliques.

Dahlhaus [13] has shown that the TREEWIDTH and the MINIMUM FILL-IN problems can be solved in polynomial time on the class of graphs which is modularly decomposable into chordal graphs. A result of a similar type is that TREEWIDTH can be solved in linear time on cographs, i.e., on graphs modular decomposable into graphs with two vertices [5]. In this paper, we extend these result to a much larger class of graphs. In particular, we show:

Theorem 1. *Let $\mathcal{C}$ be a class of graphs for which there exists a constant c such that for every graph $G \in \mathcal{C}$ all the graphs in $\pi(G)$ have at most n^c minimal separators or are a clique-matching graph. Then the* TREEWIDTH *and the* MINIMUM FILL-IN *problems on $\mathcal{C}$ can be solved in polynomial time.*

We think the most interesting part of this theorem is where we deal with graphs with a polynomial number of minimal separators; we added the result on clique-matching graphs as these have exponentially many minimal separators to show that computing TREEWIDTH and MINIMUM FILL-IN with help of the modular decomposition is not restricted to graphs with polynomially many separators. In fact, what we show is that we can compute in polynomial time the treewidth (minimum fill-in) of a given graph G, whenever G is constructed from a prime graph H, whose vertices correspond to modules in G, such that H either has polynomially many minimal separators or is a clique-matching graph. We expect that there are more types of graphs that have this property; if this property is established for a set of graphs $\mathcal{D}$, then Theorem 1 can be extended in the sense that we allow the graphs in $\pi(G)$ also belong to $\mathcal{D}$. In addition, we can allow a prime graph H in our modular decomposition with only singleton vertices below it, such that we can compute the treewidth (minimum fill-in) of H in any way (e.g., it is a regular grid, or it is a graph of treewidth bounded by some constant).

We only give two of the proofs in this extended abstract; namely for the case of cycles for TREEWIDTH and of clique-matching graphs for MINIMUM FILL-IN. The other proofs can be found in the full version[6]. The proofs for the case of prime graphs with polynomially many separators build upon results of Bouchitté and Todinca [7,8], and some other results, e.g. [17].

We define two new problems called the WI-TREEWIDTH and the WI-FILL-IN problems. We show (see Theorems 2 and 4) that an algorithm for solving the WI-TREEWIDTH (resp. the WI-FILL-IN) problem on a class of graphs $\mathcal{D}$ can be used to solve the TREEWIDTH (resp. the MINIMUM FILL-IN) problem on a class of graphs which is modularly decomposable into $\mathcal{D}$ with the same time complexity. We obtained polynomial time algorithms for the WI-TREEWIDTH and the WI-FILL-IN problems on the classes of graphs with polynomially many separators, cycles, and clique-matching graphs. (Only two of these are shown in this abstract.) Theorem 1 now follows by joining all these cases together. Cycles have $O(n^2)$ minimal separators, but the algorithm given here is faster than that obtained by applying the general result.

2 Definitions

The graphs we consider in this paper are undirected and loop-free. For a graph G we denote by $V(G)$ (resp. $E(G)$) the set of vertices (resp. edges) of G. For $X \subseteq V(G)$, we define by $G[X]$ the subgraph of G induced by X. The subgraph of G induced by $V(G) - X$ is denoted by $G - X$. $N(v)$ denotes the neighborhood of v in G, i.e., the set of vertices in G adjacent to v. A vertex v is *universal* in a graph G, if it is adjacent to all the vertices in the graph (except to itself), i.e., $N(v) = V(G) - \{v\}$. Let $S \subset V(G)$ and let C be a connected component of $G - S$. We say that C is a *full component* of S in G, if every vertex of S has a neighbor in C. We denote by $indp(G)$ the set of all independent sets in G.

Definition 1. *Let a and b be distinct nonadjacent vertices. A set $S \subset V$ is a* minimal a, b-separator *if a and b are in different connected components of $G - S$ and there is no subset of S with the same property. A* minimal separator *is a set S of vertices for which there exists vertices a and b such that S is a minimal a, b-separator.*

The following is an equivalent definition of treewidth which was introduced by Robertson and Seymour in their work on graph minors [20].

Definition 2. *A* tree decomposition *of $G = (V, E)$ is a pair $(\{X_i : i \in I\}, T)$, where $\{X_i : i \in I\}$ is a collection of subsets of V and $T = (I, F)$ is a tree such that:*

1. $\bigcup_{i \in I} X_i = V$.
2. $\forall \{u, w\} \in E, \exists i \in I : u, w \in X_i$.
3. $\forall i, j, k \in I$: *if j is on a path in T from i to k then* $X_i \cap X_k \subseteq X_j$.

The width *of a tree decomposition $(\{X_i : i \in I\}, T)$ is* $\max_{i \in I} |X_i| - 1$.
The treewidth of G *(denoted as treewidth(G)) is the minimum width over all tree decompositions of G.*

A tree decomposition $(\{X_i : i \in I\}, T)$ with T a path (i.e., every node in T has degree at most two) is called a *path decomposition*. A path decomposition is often denoted by listing the successive sets X_i: $(X_1, X_2, \ldots, X_r)$.

For disjoint graphs H and F we denote by $H \times F$ the graph G obtained by taking the union of H and F and connecting all vertices of H to all vertices of F.

For an independent set $X \in indp(G)$, we define $G : X$ to be the graph obtained from G by making each pair of neighbors of a vertex in X adjacent, and then removing all vertices in X. We assume that there are two weight functions w and t associating positive integer weights $w(v)$ and $t(v)$ for every vertex v of G. The motivation for these weights is that when G is considered as a prime graph in a modular decomposition then each vertex v of G corresponds to a module $M(v)$ and $w(v)$ and $t(v)$ will be the size and the treewidth of the module $M(v)$, respectively. For a set of vertices S we define $w(S) = \sum_{v \in S} w(v)$.

The *weighted width* of a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of G is $(\max_{i \in I} w(X_i)) - 1$. The *weighted treewidth* $wtw(G)$ of G is the minimum weighted width over all possible tree decompositions of G. Notice that $wtw(G)$ depends just on w and not on t.

For a set $X \in indp(G)$, the *weighted treewidth of G with independent set X* denoted as $wi(G, X)$ is defined by: $wi(G, X) = \max\{wtw(G : X), \max_{v \in X}\{t(v) + w(N(v))\}\}$.

The *weighted independent treewidth* of G (shortly the *wi-treewidth* of G) denoted as $wi(G)$ is defined by:

$$wi(G) = \min_{X \in indp(G)} wi(G, X)$$

For a set $X \in indp(G)$ such that $wi(G) = wi(G, X)$ we say that X *establishes* the wi-treewidth of G. The WI-TREEWIDTH problem is to find $wi(G)$ for a given graph G with weight functions w and t.

3 Treewidth

In the following text whenever we refer to $wtw(G)$ we assume that the weights of the vertices of G are defined by a weight function w. Similarly whenever we refer to $wi(G)$ we assume that the w-weights and the t-weights of the vertices of G are defined by weight functions w and t, respectively.

Lemma 1. *Let G be a graph, where $V(G) = \{v_1, \ldots, v_r\}$. Let $H_1, \ldots, H_r$ be disjoint cliques and let $G' = G(H_1, \ldots H_r)$. Let w be the weight function on the vertices of G defined by: $w(v_i) = |V(H_i)|$, for $1 \leq i \leq r$. Then $treewidth(G') = wtw(G)$.*

Lemma 2. *Let G be a graph, where $V(G) = \{v_1, \ldots, v_r\}$. Let $H_1, \ldots, H_r$ be disjoint graphs and let $G' = G(H_1, \ldots H_r)$. Let w and t be the weight functions on the vertices of G defined by: $w(v_i) = |V(H_i)|$ and $t(v_i) = treewidth(H_i)$, for $1 \leq i \leq r$. Then $treewidth(G') = wi(G)$.*

Theorem 2. *Let $\mathcal{C}$ and $\mathcal{D}$ be classes of graphs such that $\mathcal{C}$ is modularly decomposable into $\mathcal{D}$. Suppose that the* WI-TREEWIDTH *problem can be solved in $O(f(n,m))$ time on $\mathcal{D}$, where f is some polynomial function in n and m. Then the* TREEWIDTH *problem can be solved in $O(f(n,m)+n+m)$ time on $\mathcal{C}$.*

The idea is that we first build the 'modular decomposition tree', and then for every internal node, compute the treewidth of the associated graph/module, using lemma 2. For a node v in the prime graph, $w(v)$ is the treewidth of the corresponding module, and $t(v)$ the number of vertices in the module.

In the full paper, we give polynomial time algorithms for the WI-TREEWIDTH problems on weighted graphs with polynomially many minimal separators, and on clique-matching graphs. These, together with Theorem 2 imply Theorem 1 with respect to treewidth.

3.1 Solving Treewidth with Prime Graphs That Are Cycles

In this section, we show that for the class of graphs $\mathcal{C}$ which is modularly decomposable into the class of cycles, the TREEWIDTH problem on $\mathcal{C}$ can be solved in linear time. Note that a polynomial (but not linear) time algorithm follows from the result on graphs with polynomially many separators. Assume that G is a cycle with vertices $v_0, v_1, \ldots, v_{n-1}$ and with edges $\{v_i, v_{i+1}\}$, $0 \leq i \leq n-1$, where we identify v_n with v_0, and v_{-1} with v_{n-1}. For shorter notation, we write $w(i)$ for the weight $w(v_i)$ of v_i, and similar $t(i)$ for $t(v_i)$. Suppose, without loss of generality, that $w(0) = \min_{0 \leq i \leq n-1} w(i)$.

Lemma 3. *Let G be a weighted cycle with weight function w. Then*

$$wtw(G) = \min_{v \in V(G)} \{w(v)\} + \max_{\{v,x\} \in E(G)} \{w(v) + w(x)\} - 1.$$

Proof. (Sketch.) Take a tree, that is actually a path, with successive nodes $i_1, i_2, \ldots, i_{n-2}$, and take $X_{i_j} = \{v_j, v_{j+1}, v_0\}$. This is a tree decomposition of G with the desired treewidth. The proof that this is optimal is omitted. □

We assume that $n \geq 5$. (Hence for every independent set $I \subseteq V(G)$, $G : I$ is again a cycle.) We now discuss how to compute $wi(G)$.

Lemma 4. *There is an independent set $I \subseteq V(G) - \{v_0\}$, such that $wi(G) = wi(G, I)$.*

Now, assume $v_0 \notin I$. We now can write $wi(G, I)$ in more detail as:

$$wi(G,I) = \max \begin{cases} w(0) + w(j-1) + w(j+1) - 1 & | \ 1 \leq j \leq n-1, v_j \in I \\ w(0) + w(j) + w(j+1) - 1 & | \ 0 \leq j \leq n-1, v_j, v_{j+1} \notin I \\ t(j) + w(j-1) + w(j+1) & | \ 1 \leq j \leq n-1, v_j \in I \end{cases}$$

To find the independent set $I \subseteq V(G) - \{v_0\}$ for which this term is minimal, we use dynamic programming. For $i \geq 1$ and a set $I \subseteq \{v_1, \ldots, v_{i-1}\}$, define

$$h(I,i) = \max \begin{cases} w(0) + w(j-1) + w(j+1) - 1 & | \ 1 \leq j \leq i-1, v_j \in I \\ w(0) + w(j) + w(j+1) - 1 & | \ 0 \leq j \leq i-1, v_j, v_{j+1} \notin I \\ t(j) + w(j-1) + w(j+1) & | \ 1 \leq j \leq i-1, v_j \in I \end{cases}$$

For $i = 0$, define $h(\emptyset, 0) = w(0) - 1$.

Write $h(i) = \min_{I \subseteq \{v_1, \ldots, v_{i-1}\}, I \in indp(G)} h(I, i)$. Clearly, $h(n) = wi(G)$.

Lemma 5. *(i)* $h(0) = w(0) - 1$. $h(1) = 2w(0) + w(1) - 1$.
(ii) *For* $2 \leq i \leq n-1$, $h(i) = \min\{\max\{h(i-2), w(0) + w(i-2) + w(i) - 1, t(i-1) + w(i-2) + w(i), w(0) + w(i-1) + w(i) - 1)\}$.
(iii) $h(n) = \min\ \{\ \max\{h(n-1),\ 2w(0) + w(n-1) - 1\},\ \{h(n-2),\ 2w(0) + w(n-2) - 1,\ t(n-1) + w(n-2) + w(0)\}$.

The above lemma gives a dynamic programming algorithm to compute all values $h(i)$, and hence to compute the wi-treewidth of G (which equals $h(n)$) in $O(n)$ time. Hence by Theorem 2 we have:

Theorem 3. *Let $\mathcal{C}$ be a class of graphs which is modularly decomposable into the class of cycles. Then the* WI-TREEWIDTH *problem on $\mathcal{C}$ can be solved in $O(n+m)$ time.*

4 Minimum Fill-in

With some modifications of the techniques, the same results that we derive in this paper for TREEWIDTH can also be derived for the MINIMUM FILL-IN problem. We start by introducing more definitions and notations. Recall that the *minimum fill-in* of a graph G, denoted by *min-fill-in*(G), is the minimum of $|E(H) - E(G)|$ taken over all triangulations H of G. For a triangulation H of G such that $|E(H) - E(G)| =$ *min-fill-in*(G) we say that H *establishes* the minimum fill-in of G. The *complete fill-in* of G, denoted by *complete-fill-in*(G) is the number of edges that must be added to turn G into a clique:

$$\textit{complete-fill-in}(G) = |V(G)| \cdot (|V(G)| - 1)/2 - |E(G)|.$$

The MINIMUM FILL-IN problem is to find *min-fill-in*(G) for a given graph G.

Let G be a graph with weight functions w, f and c. The *weighted minimum fill-in* of G, denoted by $wf(G)$ is defined as the minimum over all triangulations H of G of $\sum_{\{v,x\} \in E(H)-E(G)} w(v) \cdot w(x)$. For a triangulation H of G such that $wf(G) = \sum_{\{v,x\} \in E(H)-E(G)} w(v) \cdot w(x)$ we say that H *establishes* the weighted fill-in of G. The *weighted complete fill-in* of G, denoted by $wcf(G)$ is defined by $wcf(G) = \sum_{\{u,v\} \notin E(G)} w(u) \cdot w(v)$. Notice that $wf(G)$ and $wcf(G)$ depends just on w and not on f and c.

Recall that for a set of vertices S we define $f(S) = \sum_{v \in S} f(v)$ and $c(S) = \sum_{v \in S} c(v)$. For a set $X \in indp(G)$, the *weighted minimum fill-in of G with independent set X* denoted as $wif(G, X)$ is defined as follows:

$$wif(G, X) = f(X) + c(V(G) - X) + wf(G : X) + \sum_{\{u,v\} \in E(G:X) - E(G)} w(u) \cdot w(v).$$

The *weighted independent minimum fill-in* of G (shortly the *wi-fill-in* of G) denoted as $wif(G)$ is defined by:

$$wif(G) = \min_{X \in indp(G)} wif(G, X).$$

For a set $X \in indp(G)$ such that $wif(G) = wif(G, X)$ we say that X *establishes* the wi-fill-in of G. The WI-FILL-IN problem is to find $wif(G)$ for a given graph G with weight functions w, f and c.

Lemma 6. *Let G be a graph, where $V(G) = \{v_1, \ldots, v_r\}$. Let $H_1, \ldots, H_r$ be disjoint graphs and let $G' = G(H_1, \ldots, H_r)$. Let w and f and c be the weight functions on the vertices of G defined by: $w(v_i) = |V(H_i)|$, $f(v_i) =$ min-fill-in(H_i) and $c(v_i) =$ complete-fill-in(H_i), for $1 \leq i \leq r$. Then min-fill-in$(G') = wif(G)$.*

Theorem 4. *Let $\mathcal{C}$ and $\mathcal{D}$ be classes of graphs such that $\mathcal{C}$ is modularly decomposable into $\mathcal{D}$. Suppose that the* WI-FILL-IN *problem can be solved in $O(f(n,m))$ time on $\mathcal{D}$, where f is some polynomial function in n and m. Then the* MINIMUM FILL-IN *problem on $\mathcal{C}$ can be solved in $O(f(n,m) + n + m)$ time.*

In this extended abstract, we will see how to handle clique-matching graphs. The proofs for cycles and graphs with polynomially many separators can be found in the full version.

4.1 Clique-Matching Graphs

In this section, we give a polynomial time algorithm for the WI-FILL-IN problem for clique-matching graphs. Suppose that $C_r = (V \cup W, E_r)$ is a weighted clique-matching graph with $2r$ vertices. We first give an algorithm to compute the weighted fill-in of a clique-matching graph C_r.

We first show that for every minimal triangulation of C_r, there is an ordering of the vertices of W such that the triangulation is formed by making every vertex w_j adjacent to all vertices v_i with w_i later than w_j in the ordering.

Lemma 7. *Let Q be a minimal triangulation of C_r. There is an ordering $\preceq$ of the vertices of W such that $E(Q) = E_r \cup \{\{w_i, v_j\} \mid w_i \preceq w_j,\ 1 \leq i \leq r,\ 1 \leq j \leq r,\ i \neq j\}$.*

Proof. Use induction to r. For $r = 1$ the lemma follows immediately since $E(Q) = E(C_r)$. Suppose the lemma holds for $r-1$. Let Q be a minimal triangulation of the graph C_r. First, note that W is a full component of V in Q. Since Q is chordal and no maximal clique in a chordal graph has a full component [8], there must be a vertex w_i in W such that $V \cup \{w_i\}$ forms a clique in Q. If v_i is adjacent in Q to any vertex w_j, $j \neq i$, then triangulation Q is not minimal. Hence the graph Q', obtained by removing v_i and w_i and their adjacent edges from Q is a minimal triangulation of $C_r - \{v_i, w_i\}$. Let $\preceq'$ be the ordering on $W - \{w_i\}$ such that $E(Q') = E(C_r - \{v_i, w_i\}) \cup \{\{w_{i'}, v_j\} \mid w_{i'} \preceq' w_j,\ 1 \leq i' \leq r,\ 1 \leq j \leq r,\ i', j \neq i\}$. Now, let $\preceq$ be the ordering on W such that for all $w_j, w_{j'} \in W - \{w_i\}$, $w_j \preceq w_{j'}$ if and only if $w_j \preceq' w_{j'}$, and for all $w_j \in W$, $w_i \preceq w_j$, i.e., we take ordering $\preceq'$ and add w_i as smallest element. One now easily sees that $\preceq$ fulfils the condition of the lemma. □

We also have that given an ordering $\preceq$ of W, the edge set $E_r \cup \{\{w_i, v_j\} \mid w_i \preceq w_j,\ 1 \leq i \leq r,\ 1 \leq j \leq r,\ i \neq j\}$ gives a triangulation of C_r. For an ordering $\preceq$ of W, let the *fill-in* of the ordering be $FI(\preceq) = \sum_{w_i \preceq w_j,\ 1 \leq i \leq r,\ 1 \leq j \leq r,\ i \neq j} w(w_i) \cdot w(v_j)$. $FI(\preceq)$ exactly denotes the total weight of all edges added in the triangulation corresponding to ordering $\preceq$, and thus the problem to compute the weighted fill-in of C_r becomes the problem to find an ordering $\preceq$ of W with minimum fill-in $FI(\preceq)$.

Lemma 8. *An ordering $\preceq$ has minimum fill-in among all orderings of W, if and only if for all w_i, $w_j \in W$: $w_i \preceq w_j \Rightarrow w(w_i)/w(v_i) \leq w(w_j)/w(v_j)$.*

Proof. Consider an ordering $\preceq$ of W. Suppose w_i and w_j are successive elements in this ordering with $w_i \preceq w_j$, i.e., there is no $w_{i'} \notin \{w_i, w_j\}$ with $w_i \preceq w_{i'} \preceq w_j$. Let $\preceq'$ be the ordering obtained from $\preceq$ by switching the order of w_i and w_j (and keeping the relative order for all other pairs). Considering all the terms that appear in $FI(\preceq)$ and $FI(\preceq')$, we see that $FI(\preceq') - FI(\preceq) = w(w_j) \cdot w(v_i) - w(w_i) \cdot w(v_j)$.

If $w(w_i)/w(v_i) > w(w_j)/w(v_j)$ then $w(w_j) \cdot w(v_i) - w(w_i) \cdot w(v_j) < 0$, hence $FI(\preceq') < FI(\preceq)$. Thus, if $\preceq$ has minimum fill-in among all orderings of W, it must order the vertices of W with respect to non-decreasing values of $w(w_i)/w(v_i)$.

If $w(w_i)/w(v_i) = w(w_j)/w(v_j)$, then $w(w_j) \cdot w(v_i) - w(w_i) \cdot w(v_j) = 0$, so $FI(\preceq') = FI(\preceq)$. As all orderings of W that give the vertices in order of non-decreasing values $w(w_i)/w(v_i)$ can be obtained from each other by a number of switches of successive elements of equal such values, we have that all such orderings have the same fill-in, so all of these have minimum fill-in. □

Lemma 8 directly gives an $O(r^2)$ algorithm to solve the WEIGHTED FILL-IN problem on clique-matching graphs: sort the vertices in W with respect to the values $w(w_i)/w(v_i)$, and then build the corresponding triangulation as described above. As the triangulated graph has $\Theta(r^2)$ edges, this latter step dominates the running time. Only computing the value of the weighted fill-in can be done a little faster: one can compute in total linear time all terms $\sum_{w_i \preceq w_j} w(v_j)$ for all $w_i \in W$, and then directly compute $FI(\preceq) = \sum_{w_i \in W} w(w_i) \cdot \sum_{w_i \preceq w_j} w(v_j)$.

Lemma 9. *The* WEIGHTED FILL-IN *problem on clique-matching graphs can be solved in $O(n \log n)$ time. A triangulation of a clique-matching graph with minimum weighted fill-in can be found in $O(n^2)$ time.*

Note that a clique-matching graph has independent sets of size at most two, and that for every $w_i \in X$, v_i is universal in $C_r : X$, and for every $v_i \in X$, w_i is universal in $C_r : X$. Moreover, we can use the following simple lemma.

Lemma 10. *Let v be a universal vertex in G. Then the weighted fill-in of G equals the weighted fill-in of $G - v$.*

Thus, we can try all $O(n^2)$ independent sets X of G, and as the graphs obtained after removing universal vertices from $G : X$ are again clique-matching graphs, use in each case the algorithm to compute the weighted fill-in of each of these graphs. Note that we can reuse the orderings of the vertices; i.e., we need to sort the vertices only once for their values $w(w_i)/w(v_i)$. Thus, we have:

Lemma 11. *The* WI-FILL-IN *problem can be solved in $O(n^3)$ time for clique-matching graphs.*

5 Conclusions

Consider the following operation, that given a graph G and for every $v_i \in V(G)$ a graph H_{v_i}, gives the graph $G(H_{v_1}, \ldots, H_{v_n})$. Theorems 2 and 4 show that

the treewidth and minimum fill-in of the composed graph are a function of the numbers of vertices and treewidths (respectively, minimum fill-ins) of the graphs H_{v_i}. These functions are expressed by respectively the WI-TREEWIDTH and WI-FILL-IN problems. In this paper, we have shown for a number of classes of graphs that for graphs in these classes these notions are computable, i.e., graphs in these classes can play the role of the graph G in the substitution operation when we want to compute the treewidth or minimum fill-in. In particular, we looked at graphs with a polynomial number of separators (a fairly large class of graphs, including several well known classes, like permutation graphs, weakly chordal graphs, HHD-graphs, graphs with $O(\log n)$ vertices, etc.) and at clique-matching graphs (a rather restricted class of graphs, introduced just to show that there are also solvable cases with an exponential number of separators.) A natural question is to solve the WI-TREEWIDTH and WI-FILL-IN problems on other interesting classes of graphs in polynomial time, for instance the class of graphs of treewidth at most some fixed number k. A related open problem, that appears to be hard, is whether treewidth can be solved in polynomial time for graphs with small cliquewidth. Espelage et al. [15] have shown that there exists a linear time algorithm for deciding whether a graph of bounded treewidth has cliquewidth k for fixed integers k.

Acknowledgements. The first author was partially supported by EC contract IST-1999-14186: Project ALCOM-FT (Algorithms and Complexity - Future Technologies). The second author wishes to thank the Natural Science and Engineering Research Council of Canada and the Fields Institute for financial assistance.

References

1. S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey. *BIT*, 25:2–23, 1985.
2. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.
3. L. Babel. Triangulating graphs with few P_4's. *Disc. Appl. Math.*, 89:45–57, 1998.
4. H. L. Bodlaender, T. Kloks, and D. Kratsch. Treewidth and pathwidth of permutation graphs. *SIAM J. Disc. Math.*, 8(4):606–616, 1995.
5. H. L. Bodlaender and R. H. Möhring. The pathwidth and treewidth of cographs. *SIAM J. Disc. Math.*, 6:181–188, 1993.
6. H. L. Bodlaender and U. Rotics. Computing the treewidth and the minimum fill-in with the modular decomposition. Technical Report CS-UU-2001-22, Institute of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands, 2001. ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-2001/2001-22.pdf.
7. V. Bouchitté and I. Todinca. Listing all potential maximal cliques of a graph. In H. Reidel and S. Tison, editors, *Proceedings STACS'00*, pages 503–515. Springer Verlag, Lecture Notes in Computer Science, vol. 1770, 2000.
8. V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: grouping the minimal separators. *SIAM J. Comput.*, 31:212–232, 2001.

9. H. Broersma, E. Dahlhaus, and T. Kloks. Algorithms for the treewidth and minimum fill-in of HHD-free graphs. In *Proceedings 23nd International Workshop on Graph-Theoretic Concepts in Computer Science WG'97*, pages 109–117. Springer Verlag, Lecture Notes in Computer Science, vol. 1335, 1997.
10. H. Broersma, E. Dahlhaus, and T. Kloks. A linear time algorithm for minimum fill in and treewidth for distance hereditary graphs. *Disc. Appl. Math.*, 99:367–400, 2000.
11. H. Buer and R. H. Möhring. A fast algorithm for the decomposition of graphs and posets. *Mathematics of Operations Research*, 8(2):170–184, 1983.
12. A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In T. Sophie, editor, *Trees in algebra and programming, CAAP'94*, pages 68–84. Springer Verlag, Lecture Notes in Computer Science, vol. 787, 1994.
13. E. Dahlhaus. Minimum fill-in and treewidth for graphs modularly decomposable into chordal graphs. In *Proceedings 24nd International Workshop on Graph-Theoretic Concepts in Computer Science WG'98*, pages 351–358. Springer Verlag, Lecture Notes in Computer Science, vol. 1517, 1998.
14. E. Dahlhaus, J. Gustedt, and R. M. McConnell. Efficient and practical modular decomposition. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 26–35, 1997.
15. W. Espelage, F. Gurski, and E. Wanke. Deciding clique-width for graphs of bounded treewidth. In *Proceedings WADS 2001*, pages 87–98. Springer Verlag, Lecture Notes in Computer Science, vol. 2125, 2001.
16. T. Kloks. Treewidth of circle graphs. *Int. J. Found. Computer Science*, 7:111–120, 1996.
17. T. Kloks and D. Kratsch. Listing all minimal separators of a graph. *SIAM J. Comput.*, 27(3):605–613, 1998.
18. R. M. McConnell and J. Spinrad. Modular decomposition and transitive orientation. *Disc. Math.*, 201:189–241, 1999.
19. R. H. Möhring. Graph problems related to gate matrix layout and PLA folding. In E. Mayr, H. Noltemeier, and M. Sysło, editors, *Computational Graph Theory, Comuting Suppl. 7*, pages 17–51. Springer Verlag, 1990.
20. N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of treewidth. *J. Algorithms*, 7:309–322, 1986.
21. R. Sundaram, K. Sher Singh, and C. Pandu Rangan. Treewidth of circular-arc graphs. *SIAM J. Disc. Math.*, 7:647–655, 1994.
22. M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

Performance Tuning an Algorithm for Compressing Relational Tables

Jyrki Katajainen[1] and Jeppe Nejsum Madsen[2]

[1] Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk
[2] Array Technology A/S
Fruebjergvej 3, DK-2100 Copenhagen East, Denmark
jnm@arraytechnology.com

Abstract. We study the behaviour of an algorithm which compresses relational tables by representing common subspaces as Cartesian products. The output produced allows space to be saved while preserving the functionality of many relational operations such as select, project and join. We describe an implementation of an existing algorithm, propose a slight modification which with high probability produces the same output, and present a performance study showing that for all test instances used both adaptations are considerably faster than the current implementation in a commercial software product.

1 Introduction

Tables of relational data play an important role in many applications.

Definition 1 ([5]). *A* relation *consists of a* scheme *and an* instance*:*

1. *A* scheme *is a finite set of attributes. Each attribute is associated with a set of values, called its* domain*.*
2. *A* tuple *over a scheme is a mapping that associates with each attribute of the scheme a value from the corresponding domain.*
3. *An* instance *over a scheme is a finite set of tuples over that scheme.*

Here we study the use of relations in the context of constraint satisfaction problems (CSPs) [8], where relations are used to specify legal combinations of variables. The decision version of the Boolean CSP is $\mathcal{NP}$-complete (see, e.g., [9]). In many CSP algorithms, it is usually not feasible to use a direct representation of relations due to the combinatorial explosion of the solution space (e.g., an unconstrained Boolean CSP with m variables has 2^m possible solutions).

In the case of finite-domain CSPs, the variable values can be encoded as integers since the domain sizes are known in advance. Therefore, it is assumed that attribute values fit in a machine word. The model of computation used is the *word RAM*, a term used by Hagerup [6] among others.

Møller [11] described how relations could be represented by Cartesian products, which is illustrated in the following example.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 398–407, 2002.

Example 1. Consider a relation with the following tuples:

$$\{< 0,0,0 >, < 0,0,1 >, < 0,1,1 >, < 1,0,1 >, < 1,1,1 >\}.$$

An alternative representation is to use, where feasible, a Cartesian product to generate the set of tuples. Thus the same set can be represented as follows:

$$\{< 0,0,0 >\} \cup (\{0,1\} \times \{0,1\} \times \{1\}).$$

The last representation is what we refer to as a *compressed relation*. When using a tabular notation, we display the relations as shown below:

Original relation:

A	B	C
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

Compressed relation:

A	B	C
0	0	0
$\{0,1\}$	$\{0,1\}$	1

In the tabular notation a Cartesian product is implied between sets on the same row. In addition, the set delimiters $\{\}$ are omitted when the set is a singleton. □

As our starting point we use Møller's algorithm for compressing relations. It was originally implemented in APL and later in C++ in the commercial software product Array Database$^{\text{TM}}$[2]. In this paper we analyse the complexity of the algorithm (Sect. 3), propose a modification (Sect. 4), and present a new implementation (Sect. 5) which according to our experiments (Sect. 6) is significantly faster than earlier implementations. It is planned that a future release of the software will include the tuned implementation.

2 Preliminaries

We first need to define when two relation representations are equivalent.

Definition 2 (Equivalent relation representation). *Two relation representations are said to be* equivalent *if the underlying relation instances are equal after the evaluation of all Cartesian products.*

We also need to establish a measure on the size of a representation. We ignore any overhead needed to administrate the data structure, and therefore arrive at the following definition, which is valid for both uncompressed and compressed representations.

Definition 3. *Let k denote the number of attributes, n_i the number of scalar values in the ith column and d_i the domain size of the ith attribute. The* size *of a relation, measured in bits, is $\sum_{i=1}^{k} n_i \lceil \log_2 d_i \rceil$.*

Using this definition on the relations of Example 1, we see that the uncompressed relation has size 15 while the compressed relation has size 8.

Ranking different compressed representations leads to a (not necessarily unique) optimal representation defined as follows:

Definition 4. *A representation is said to be* minimal *if no equivalent representation exists which has a smaller size.*

In general, the Cartesian arguments can be selected from subsets of the relation scheme. We formalize this selection as follows:

Definition 5. *A* compression scheme *for a relation with scheme S is a (possibly empty) sequence of pairwise disjoint subsets of S. Each subset defines the scope of a Cartesian argument. An* optimal compression scheme *is a compression scheme, which yields a minimal representation of that relation.*

Note that the subsets need not cover the whole relation scheme, i.e. attributes, which are not members of the compression scheme, will not have their values compressed. However, since we are interested in decreasing the size of a representation and compression never increases the size, we only consider compression schemes which actually cover the whole relation scheme.

An exhaustive search for an optimal compression scheme is intractable for all but the smallest relation schemes, as shown by the following proposition.

Proposition 1. *The number of compression schemes that cover a k-ary relation is B_k, where B_k is defined by the following recurrence relation:*

$$B_0 = B_1 = 1, \text{ and } B_k = \sum_{i=1}^{k} \binom{k}{i} B_{k-i} \text{ for } k > 1.$$

Proof. The cases for $k = 0$ and $k = 1$ are obvious. For $k > 1$, the size of the first subset can be i, $i \in \{1, \ldots, k\}$, and each such subset can be selected in $\binom{k}{i}$ ways. There are B_{k-i} ways to cover the remaining elements. From these facts the recurrence follows. □

3 Description and Analysis of Møller's Heuristic

When using the information in a relation (i.e., when joining two relations), we usually need the values columnwise. Therefore, we only consider the compression schemes consisting of singletons spanning the relation scheme, i.e., the Cartesian arguments span a single attribute.

Møller [11] introduced a heuristic for compressing relations where columns are considered one at a time. To describe the heuristic we need the concept of complement for a relation:

Definition 6. *The* complement *of a (possibly compressed) relation R with respect to attribute A is the tuples of R with the values corresponding to A removed.*

The heuristic works in two phases:

Phase 1: The input relation is analysed to determine the order in which the columns are to be considered. This is done by computing, for each attribute, the number of unique tuples in the corresponding complement.

Phase 2: To get the compressed relation the columns are considered in non-decreasing order of the number of unique tuples in the uncompressed complements. The column being considered is removed from the relation and duplicates are eliminated from the complement. During duplicate elimination each unique tuple in the complement is associated with the different values of the attribute considered. Using this information the compressed relation is constructed.

Example 2. We now illustrate the heuristic by compressing the relation from Example 1. First, we determine the number of unique tuples in each attribute's complement:

A	B	C
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{\text{remove column } A}$

B	C
0	0
0	1
1	1
0	1
1	1

$\xrightarrow{\text{remove duplicates}}$

B	C
0	0
0	1
1	1

$\xrightarrow{\#\text{tuples}}$ 3

A	B	C
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{\text{remove column } B}$

A	C
0	0
0	1
0	1
1	1
1	1

$\xrightarrow{\text{remove duplicates}}$

A	C
0	0
0	1
1	1

$\xrightarrow{\#\text{tuples}}$ 3

A	B	C
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{\text{remove column } C}$

A	B
0	0
0	0
0	1
1	0
1	1

$\xrightarrow{\text{remove duplicates}}$

A	B
0	0
0	1
1	0
1	1

$\xrightarrow{\#\text{tuples}}$ 4

Hence, the compression schemes $\langle\{A\},\{B\},\{C\}\rangle$ and $\langle\{B\},\{A\},\{C\}\rangle$ are both valid. If we choose compression scheme $\langle\{A\},\{B\},\{C\}\rangle$, the compression proceeds as follows.

A	B	C
0	0	0
0	0	1
0	1	1
1	0	1
1	1	1

$\xrightarrow{\text{handle column } A}$

A	B	C
0	0	0
$\{0,1\}$	0	1
$\{0,1\}$	1	1

$\xrightarrow{\text{handle column } B}$

A	B	C
0	0	0
$\{0,1\}$	$\{0,1\}$	1

$\circlearrowright$ handle column C

□

For the purpose of analysis, we assume that the input relation is uncompressed and does not contain any identical tuples. Let k denote the number of attributes and n the number of tuples in the relation.

The essential part of Phase 1 is duplicate elimination, which can be carried out by sorting the tuples using any vector sorting algorithm (see, for instance,

[3]), scanning the result, and counting the number of unique tuples. The running time is dominated by the cost of sorting, which is $\mathcal{O}(kn + n \log_2 n)$. Since the elimination must be performed k times, the running time of Phase 1 is $\mathcal{O}(k^2 n + kn \log_2 n)$.

In our application we sometimes need to compress relations, which are already partially compressed. It is not obvious how to modify vector sorting algorithms to handle the situation where the elements are sets. Hence, in our implementation we rely on hashing. We eliminate duplicates by maintaining a hash table of size $\Theta(n)$. Each tuple in the complement is checked against the hash table and, if it is not present, it is inserted into the table. At the end the number of elements stored in the hash table is equal to the number of unique tuples in the complement. The computation of a hash value takes $\mathcal{O}(k)$ time. A lookup and an insert both take $\mathcal{O}(k)$ expected time. In total, the duplicate elimination requires $\mathcal{O}(kn)$ expected time. Thus, the expected running time of Phase 1 is $\mathcal{O}(k^2 n)$. The worst-case running time for Phase 1 becomes $\mathcal{O}(k^2 n \log_2 n)$ if we implement each entry in the hash table as a balanced binary search tree.

Since, in general, complements contain compressed columns, we also rely on hashing in Phase 2. The number of scalar values contained in a single complement is never larger than kn. Hence, the calculation of the hash values for all tuples in a complement takes $\mathcal{O}(kn)$ time. The running time of a lookup and an insert depends on the size of the tuple. If there are other tuples with the same hash value, we need to compare whether any of these are equal to the searched tuple. If we keep the sets sorted, the cost of a comparison is linear in the size of the searched tuple. Lookups and possible inserts for all tuples take $\mathcal{O}(kn)$ expected time. The sorting cost is $\mathcal{O}(n \log_2 \min\{d_{\max}, n\})$, where $d_{\max}$ is the size of the largest domain.

To summarize, the running time of Phase 2 is $\mathcal{O}(k^2 n + kn \log_2 \min\{d_{\max}, n\})$ in the average case. The running time of Phase 2 becomes $\mathcal{O}(k^2 n \log_2 n)$ in the worst case if we implement each entry in the hash table as a balanced binary search tree.

4 A Modification of Phase 1

Instead of calculating the number of unique tuples in the complement exactly, we propose that only an approximation is computed. We use a hash function to compute a signature for each tuple in the complement, and the number of unique signatures is used as an approximation for the number of unique tuples. This way we save the equality tests for colliding tuples in the hash table.

The computation of the approximation for one complement is carried out as follows. First, the signatures are computed for each tuple. Second, the signatures are sorted using radix sort [1, p. 77 ff.]. Third, in a final scanning the number of distinct signatures is calculated.

To compute the signatures fast, we employ *strongly universal hashing*, defined by Carter and Wegman [13]:

Definition 7 ([4]). *Let U and T be subsets of the natural numbers. A class $\mathcal{H}$ of hash functions from U to T is said to be* universal *if, on any pair of distinct elements from U, for a randomly chosen hash function h from $\mathcal{H}$ the probability of a collision is $\mathcal{O}(1/|T|)$, i.e., for all $x, y \in U$, $x \neq y$:* $\Pr(h(x) = h(y)) = \mathcal{O}(1/|T|)$.

Definition 8 ([13]). *Let U and T be subsets of the natural numbers. A class $\mathcal{H}$ of hash functions from U to T is said to be* strongly universal *if a randomly chosen hash function h from $\mathcal{H}$ maps elements pairwise independently, i.e., for all $x, y \in U$, $x \neq y$, and for all $\alpha, \beta \in T$:* $\Pr(h(x) = \alpha \text{ and } h(y) = \beta) = \mathcal{O}(1/|T|^2)$.

Strongly universal hash functions support the following type of vector hashing:

Proposition 2 ([4]). *Let q be a positive integer, U and T subsets of the natural numbers, and $\mathcal{H}$ a strongly universal class of hash functions from U to T. Furthermore, let $\mathcal{H}^q$ denote the class of hash functions from U^q to T such that $(h_1, \ldots, h_q)(x_1, \ldots, x_q) = h_1(x_1) \oplus \cdots \oplus h_q(x_q)$ where $\oplus$ is the binary XOR operation and $h_i \in \mathcal{H}$ for all $i \in \{1, \ldots, q\}$. Then $\mathcal{H}^q$ is strongly universal.*

We can utilize vector hashing as follows. Let $h_*(r_{i*})$ denote the vector hash value for the ith tuple, $h_j(r_{ij})$ the hash value for the ith tuple and the jth attribute using a strongly universal hash function h_j. Then we have

$$h_*(r_{i*}) = h_1(r_{i1}) \oplus \cdots \oplus h_k(r_{ik}).$$

We can compute the vector hash value h_* for each tuple in the relation in time $\mathcal{O}(kn)$ assuming that the computation of $h_j(r_{ij})$ takes $\mathcal{O}(1)$ time. When we need the hash value $h_{\bar{j}}(r_{i*})$ for the ith tuple in the complement with respect to the jth attribute, we can calculate this as

$$\begin{aligned} h_{\bar{j}}(r_{i*}) &= h_1(r_{i1}) \oplus \cdots \oplus h_{j-1}(r_{i,j-1}) \oplus h_{j+1}(r_{i,j+1}) \oplus \cdots \oplus h_k(r_{ik}) \\ &= h_*(r_{i*}) \oplus h_j(r_{ij}). \end{aligned}$$

This means that we can calculate the hash values for each tuple in the complement in $\mathcal{O}(n)$ time if we use $\mathcal{O}(kn)$ time to precompute the vector hash values. Thus, the signatures for all k complements are computed in $\mathcal{O}(kn)$ time in the worst case.

As the following proposition shows, the compression scheme obtained with our modification is, with high probability, the same as that produced by Phase 1 of Møller's heuristic.

Proposition 3. *Assume that the input relation has k attributes and n tuples. For a signature universe T, for which $|T| = n^{2+\varepsilon}$ for $\varepsilon > 0$, the probability that the outcome of our modification is the same as that of Phase 1 of Møller's heuristic is at least $1 - 1/n^{\varepsilon}$. The worst-case running time of our modification is $\mathcal{O}((2 + \varepsilon)kn)$.*

Proof. For any universal hash function, the probability of a collision is $\mathcal{O}(1/|T|)$. Assume that of the n tuples m are unique. Therefore, the probability that any of the $\binom{m}{2}$ pairs of unique tuples collides is bounded by $\binom{m}{2} \cdot \mathcal{O}(1/|T|)$. For $|T| = n^{2+\varepsilon}$ the claimed bound follows since $m \leq n$.

As discussed above the computation of the signatures for all k complements takes $\mathcal{O}(kn)$ time. As shown in [1, p. 79], the sorting of n integers in the range $\{0, \ldots, n^{\gamma} - 1\}$ is done in $\mathcal{O}(\gamma n)$ time using radix sort. The final scanning of the signatures requires $\mathcal{O}(\gamma n)$ time. For $\gamma = 2 + \varepsilon$, the total running time for handling all k complements is $\mathcal{O}((2 + \varepsilon)kn)$. □

5 Implementation Details

We need a strongly universal hash function that calculates a hash value for arbitrary b-bit tuples. A standard trick is to tabulate hash functions for 8-bit characters. A b-bit tuple is then viewed as a string of $\lceil b/8 \rceil$ characters, and the hash value for the ith character is found by a table lookup in the ith table using the 8-bit character as index. The hash value for the entire tuple can then be calculated as described in Proposition 2. This hash function is very fast on the platform considered, the Intel Pentium, since only table lookups and XOR are involved. For a performance study of strongly universal hash functions, see [12].

The drawback of the chosen class of hash functions is that we need independent tables for each character in the tuple being hashed, which makes the class unsuitable for long tuples. As in [7], we are pragmatic and fix the number of tables (to 8 in our case) and use the generated tables cyclically. Furthermore, the signatures are full words and, when sorting the signatures, we use the routine provided by the C++ standard library.

Uncompressed columns are stored as a sequence of integers. For the sake of efficiency, we store the integers using 1, 2, or 4 bytes, 8 bits each. A compressed column is stored as a sequence of either fixed-size bit vectors or sorted integer sequences. The fixed-size bit vectors are 1, 2, or 4 bytes wide and used whenever the domain has no more than 8, 16, and 32 values, respectively. When the domain has more than 32 values, a sorted integer sequence is used, where the integers are stored using 1, 2, or 4 bytes. It may be beneficial to encode even larger domains as bit vectors, since we avoid memory allocation for set elements and we exhibit better cache behaviour. The exact bound, when the encoding should switch to using a sorted integer sequence, is to be determined experimentally.

The performance critical parts of the code are the inner loops in the functions that implement the compress (and join) functionality. Since different C++ types are used to represent the column data (3 types for the uncompressed columns and 6 for the compressed columns), the compress function should potentially be implemented in 9 different versions, one for each type of column.

By using the concept of traits classes, originally introduced by Meyers [10], it is possible to combine speed of execution with ease of maintenance. Here we use the term *traits class* to refer to a class that aggregates the basic operations (copy, intersection, hash value etc.) on a single cell in a column. By supply-

ing different traits classes to the same template function, this function can be applied to different column types. The compiler will generate an appropriate implementation of the function, specialized for the types passed. Since the compiler knows the types at the time of compilation, full optimization and inlining can be applied in the critical inner loops. Virtual functions are thus not invoked in the performance critical parts of the code.

6 Performance Study

In this section we present the results of a performance study in which we compared the performance of our C++ implementations of the algorithms described in Sect. 3 and 4, respectively, to existing implementations written in APL and C++. The following four implementations were considered:

APL: The original APL implementation described in [11].

Current: The C++ implementation used in the commercial software product Array Database^{TM} version 5.5. This is a straightforward translation of the APL code using optimizations, like hashing and reference data types, not available in APL.

Tuned: A performance-tuned C++ implementation of Møller's heuristic.

Approx: A C++ implementation of our approximation heuristic discussed in Sect. 4.

All the experiments were performed on a Dell Inspiron 5000e with a 750 MHz Pentium III processor and 384 MByte memory. The operating system used was Windows 2000, Service Pack 2. For the C++ tests, the compiler used was Microsoft Visual C++ version 7. For the APL tests, the compiler was Dyadic APL for Windows version 8.2 release 4. Each experiment was run multiple times, with no significant variance observed in the execution times.

The relations used in the experiments were collected from real-life CSP instances, which have been used in various projects. The characteristics of the input data used for comparing the programs are shown in Table 1, where k denotes the number of attributes in the relation scheme, d_i the size of the domain for the ith attribute, n the number of tuples, s the size of the relation as defined in Definition 3, n_c the number of tuples in the compressed relation, and s_c the size of the compressed relation. The two last columns show n_c and s_c as percentages of n and s, respectively. The compressed output was the same for all methods in all runs.

The performance results are presented in Table 2. The speedup factor is calculated using the existing C++ implementation as a base. As can be seen, the new implementations are significantly faster than both existing implementations. The difference seems to increase as the size of the relation increases. It is also evident that the speedup is larger when the domains are small. This seems to indicate that encoding the compressed columns as bit vectors is beneficial.

Table 1. Characteristics of the input data used in our performance comparison.

Input	k	$\sum_{i=1}^{k} d_i$	n	s	n_c	s_c	n_c %	s_c %
heq	10	1643	151374	5903586	5020	1362258	3.3%	23.1%
plan31	14	28	8192	114688	14	274	0.2%	0.2%
q10a	8	80	149552	4785664	13144	661944	8.8%	13.8%
q10b	8	80	55658	1781056	6632	318672	11.9%	17.9%
ns11	11	65	333322	9666338	102	7304	0.03%	0.08%

Table 2. Execution times of compression programs. All times are measured in CPU-seconds.

	APL		Current		Tuned		Approx	
Input	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
heq	36.5	0.33	12.01	1	2.63	4.6	0.91	13.2
plan31	9.10	0.11	1.031	1	0.11	9.4	0,06	17.1
q10a	233	0.32	74.02	1	1.74	42.5	0.96	77.1
q10b	47.8	0.21	9.884	1	0.63	15.7	0.45	22.9
ns11	660	0.57	377.5	1	5.80	65.1	1.87	196

7 Conclusion

We provided a detailed description and complexity analysis of a heuristic to compress relational tables. This heuristic was previously only informally described, with no complexity results. Furthermore, we provided an enhanced implementation of the heuristic and benchmarked it against existing implementations. The results obtained show that our baseline implementation of the heuristic is considerably faster than previous implementations for the data sets used. Also, we proposed a modification to the heuristic, which further improves the running time while producing the same compressed output with high probability.

In an extension of this paper, we have studied the performance of the join operation on compressed relations. The results obtained show considerable improvements over existing implementations although not as dramatic as the results for the compress operation. For further details, see [9].

References

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[2] Array Technology A/S, Array technology, Website accessible at `http://www.arraytechnology.com` (2002).

[3] J. L. Bentley and J. B. Saxe, Algorithms on vector sets, *SIGACT News* **11**,9 (1979), 36–39.

[4] J. L. Carter and M. N. Wegman, Universal classes of hash functions, *Journal of Computer and System Sciences* **18**,2 (1979), 143–154.

[5] E. F. Codd, A relational model of data for large shared data banks, *Communications of the ACM* **13**,6 (1970), 377–387.

[6] T. Hagerup, Sorting and searching on the word RAM, *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1373**, Springer-Verlag (1998), 366–398.

[7] J. Katajainen and M. Lykke, Experiments with universal hashing, Technical Report 96/8, Department of Computer Science, University of Copenhagen (1996).

[8] A. K. Mackworth, Constraint satisfaction, *Encyclopedia of Artificial Intelligence*, 2nd Edition, John Wiley & Sons (1992), 285–293.

[9] J. N. Madsen, Algorithms for compressing and joining relations, CPH STL Report 2002-1, Department of Computing, University of Copenhagen (2002). Available at `http://www.cphstl.dk`.

[10] N. C. Meyers, Traits: A new and useful template technique, *C++ Report* (1995). Available at `http://www.cantrip.org/traits.html`.

[11] G. L. Møller, On the technology of array-based logic, Ph. D. Thesis, Technical University of Denmark (1995). Available at `http://www.arraytechnology.com/documents/lic.pdf`.

[12] M. Thorup, Even strongly universal hashing is pretty fast, *Proceedings of the 11th Annual Symposium on Discrete Algorithms*, ACM-SIAM (2000), 496–497.

[13] M. N. Wegman and J. L. Carter, New hash functions and their use in authentication and set equality, *Journal of Computer and System Sciences* **22**,3 (1981), 265–279.

A Randomized In-Place Algorithm for Positioning the kth Element in a Multiset*

Jyrki Katajainen[1] and Tomi A. Pasanen[2]**

[1] Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk
[2] Turku Centre for Computer Science, University of Turku
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
tomi.pasanen@cs.utu.fi

Abstract. A variant of the classical selection problem, called the *positioning problem*, is considered. In this problem we are given a sequence $A[1{:}n]$ of size n, an integer k, $1 \leq k \leq n$, and an ordering function $\lessdot$, and the task is to rearrange the elements of the sequence such that $A[k] \lessdot A[j]$ is false for all j, $1 \leq j < k$, and $A[\ell] \lessdot A[k]$ is false for all ℓ, $k < \ell \leq n$. We present a Las-Vegas algorithm which carries out this rearrangement efficiently using only a constant amount of additional space even if the input contains equal elements and if only pairwise element comparisons are permitted. To be more precise, the algorithm solves the positioning problem in-place in linear time using at most $n + k + o(n)$ element comparisons, $k + o(n)$ element exchanges, and the probability for succeeding within stated time bounds is at least $1 - e^{-n^{\Omega(1)}}$.

1 Introduction

In the *selection problem* the task is to find, given a multiset and an integer k, the kth smallest element of the multiset. We consider a variant of this problem — and call it the *positioning problem* — examined by Hoare [7]: given a sequence $A[1{:}n]$ of n elements, an integer k, $1 \leq k \leq n$, and an ordering function $\lessdot$ returning true or false, rearrange the sequence in such a way that $A[k] \lessdot A[j]$ is false for all j, $1 \leq j < k$, and $A[\ell] \lessdot A[k]$ is false for all ℓ, $k < \ell \leq n$. We will assume that $k \leq \lceil n/2 \rceil$ since the case $k > \lceil n/2 \rceil$ can be solved symmetrically by positioning the $(n-k)$th element in the reverse sequence.

If we had extra space available, the positioning problem for multisets could be solved by tagging each element with its index, comparing the pairs lexicographically, and applying any of the existing algorithms for the set of pairs.

* Partially supported by Danish Natural Science Research Council under contract 9701414 (project Experimental Algorithmics) and contract 9801749 (project Performance Engineering).

** Current address: Institute of Medical Technology, University of Tampere, Lenkkeilijänkatu 8, FIN-33520 Tampere, Finland. Email: Tomi.Pasanen@uta.fi

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 408–417, 2002.

However, if the rearrangement of the elements is to be done *in-place*, i.e., using only constant amount of extra space, the problem becomes nontrivial.

The efficiency of sorting algorithms is traditionally measured by calculating the *number* of *element comparisons* performed. In particular, observe that we allow only pairwise element comparisons. When the movement of elements is done in a strictly in-place manner, i.e., by swapping the elements wordwise, the *number* of *element exchanges* is another natural performance measure.

Of the classic selection algorithms [1,4,7,18] only the algorithm by Hoare [7] operates in-place. If in his algorithm the partitioning is carried out using the indices to make the elements distinct, and if the median of three random elements is used as the partitioning element in each partitioning, the algorithm performs at most $2.75n + o(n)$ element comparisons on an average when positioning the $\lceil n/2 \rceil$th element in a sequence of n elements; for the exact bounds for general k, see [12]. It is well-known that for a permutation of n distinct elements the average number of element exchanges performed during each partitioning is 1/6 times that of element comparisons (see, for example, [19, pp. 333–334]). That is, when positioning the $\lceil n/2 \rceil$th element the average number of element exchanges performed is bounded by $0.46n + o(n)$.

The algorithm of Floyd and Rivest [4] (see also [15, Section 3.3] and [17]), on which our algorithm is based, selects the kth smallest of n elements using at most $n + k + o(n)$ element comparisons. This algorithm can also be used for positioning, not only for selection, but it was designed for sets. Furthermore, it is of the *Las Vegas* type, i.e., it may fail to finish in time, but it will always produce a correct result. From the lower bound of [3] and Yao's minimax principle (see, e.g., [15, Proposition 2.5]) it follows that $n + k - O(1)$ is a lower bound on the expected number of element comparisons performed by any Las-Vegas algorithm for selecting the kth smallest of n elements.

In this paper we describe an in-place adaptation of the algorithm of Floyd and Rivest [4] which can handle multiset data efficiently. It carries out the positioning of the kth element in a multiset of size n in $O(n)$ time using at most $n + k + o(n)$ pairwise element comparisons and at most $k + o(n)$ element exchanges; the probability that these resource bounds are exceeded is at most $e^{-n^{\Omega(1)}}$. To achieve these bounds two ordering functions $\lessdot$ and $\eqslantless$ must be provided. If only $\lessdot$ is provided, the algorithm may require $2n + o(n)$ element comparisons. Due to the above-mentioned lower bound, if we ignore the lower order terms, the number of element comparisons performed is best possible. Also, the number of element exchanges performed is optimal if we do not make any assumptions about the input.

For the selection problem several deterministic in-place algorithms have been proposed [2,5,13], but none of these solve the positioning problem as such. First, the algorithm of Lai and Wood [13] can be used for finding the kth smallest element, but it will not carry out the partitioning required. Of course, a simple solution is to carry out a three-way partitioning after the selection, but this may require $2n$ additional element comparisons and k additional element exchanges. Second, both the algorithms of Carlsson and Sundström [2], and Lai

and Wood [13], as well as the algorithm of Geffert [5] which uses one of the above-mentioned algorithms as a subroutine, rely on three-way element comparisons. A naive solution is to replace each three-way comparison with two binary comparisons, but this will double the comparison count. In the full version of this paper we describe a deterministic in-place positioning algorithm which requires $3.64n + 0.72k + o(n)$ pairwise element comparisons — improving the earlier results — but this is still far away from the bound achieved by the randomized method.

2 Bit Encoding

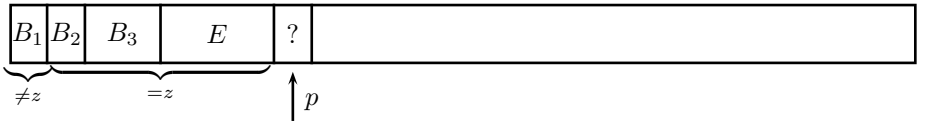
Our in-place positioning algorithm relies on the bit encoding technique introduced by Munro [16]. This technique has turned out to be crucial in many other in-place algorithms (see, for example, [8,9,11,13,14]). In this section we describe how the technique is used in the present context.

Two distinct elements x and y, $x \lessdot y$, can be used to represent a 0-bit by storing them in two consecutive locations in order xy, and a 1-bit by storing them in order yx. By using $\lceil \log_2(n+1) \rceil$ such pairs an integer value up to n can be represented. To read the value or to update the value of such an integer, $O(\log_2 n)$ element comparisons and $O(\log_2 n)$ element exchanges might be necessary.

Assume that n is the input size of the positioning problem. For fixed integer $c \geq 1$ and real number $0 < \beta < 1$, our algorithm will need cn^β integers whose value is between 0 and n. In order to represent these integers using the bit encoding technique we find $cn^\beta 2\lceil \log_2(n+1) \rceil$ distinct elements and transfer them into the beginning of the input sequence. Next we describe how this preprocessing is done.

For the sake of brevity, let $b = cn^\beta 2\lceil \log_2(n+1) \rceil$. First, we sort the b first elements of the input; let B be the resulting section. Here any in-place sorting algorithm can be used, e.g., heapsort [20] or in-place mergesort [10]. This requires $O(b \log_2 b)$ time. If each element in B has less than $b/2$ duplicates — this check requires $O(b)$ time — the elements $B[i]$ and $B[b/2 + i]$ must be distinct for all $i = 1, 2, \ldots, b/2$, and each of these pairs can be used to represent a bit. The interleaving of the first and the second half of B, i.e., moving the pairs $B[i]$ and $B[b/2 + i]$ into consecutive locations for $i = 1, 2, \ldots, b/2$, is easily carried out in-place in $O(b \log_2 b)$ time. At the same time the bits can be initialized to zero. In this case the construction is complete.

Second, if some element, say z, appears in B at least $b/2$ times, we scan the remaining sequence to find $b/2$ elements that are different from z. Before starting the scan we move all elements of B different from z to the beginning of B forming a section B_1. This requires a single block interchange, i.e., $O(b)$ time. The remaining elements, elements equal to z, are divided to two consecutive sections, B_2 and B_3, so that B_3 includes $b/2$ elements and B_2 what is left. The section B_2 between B_1 and B_3 is to be filled with elements different from z. After B_3 there is a section, call it E, that also contains elements equal to z. That is, during the scan the input sequence has the form:



If the element pointed to by cursor p is equal to z, the cursor is incremented by one, E is made one larger, and the element at the new cursor position is examined next. Otherwise, the element pointed to is swapped with the first element of B_2, B_1 becomes one larger, B_2 one smaller, E one larger, the cursor is incremented by one, and the new cursor position is considered next. The scan is stopped when B_2 gets full or when the end of the input sequence is reached.

If the end of the sequence is reached before B_2 gets full, we know that most of the elements are equal to z and the underlying positioning problem is easy to solve. The elements in B_1 are sorted; let $B_<$ denote the section of elements less than z in the sorting result and $B_>$ the section of elements larger than z. Then the contents of $B_>$ is swapped with a block of equal size at the end of $B_2 \cup B_3 \cup E$. Since after this the whole sequence is in sorted order, the kth smallest element is correctly positioned. The sorting of B_1 requires at most $O(b \log_2 b)$ time, and the block swap involving $B_>$ at most $O(b)$ time. During the scan one element comparison is made for each element kept in E if the ordering function $\ominus$ is provided. Since $b \log_2 b = o(n)$, this special case is solved in $O(n)$ time using at most $n + o(n)$ element comparisons and at most $o(n)$ element exchanges. If only the ordering function $\oslash$ is provided, two comparisons per element might be necessary.

Let us hereafter assume that there are enough elements different from z. Assume that the configuration after the scan is $B_1 B_3 E U$, where U denotes the rest of the sequence not yet touched. As earlier, the interleaving of B_1 and B_3 requires $O(b \log_2 b)$ time. This completes the preprocessing and the final configuration is BEU, B containing the consecutive pairs that can be used for representing $b/2$ bits. To get to this configuration $O(n)$ time has been used and it is necessary to carry out one element comparison for each element in E (or two if only the ordering function $\oslash$ is provided). Potentially, E can be large, but later on we can avoid the comparisons involving the elements in E since these are known to be equal to z.

3 Randomized Positioning

As our starting point we use a nonrecursive variant of the selection algorithm of Floyd and Rivest [4] described, for example, in [15, Section 3.3]. The basic idea is to draw a small random sample from the input sequence with replacement, choose two sample elements x and y, $x \oslash y$, and then partition the sequence into three sections: the elements lexicographically less than x, the elements between x and y, including themselves, and the elements lexicographically greater than y. The crux is how to choose elements x and y such that the number of elements between them is $o(n/\log_2 n)$, x is lexicographically less than the kth element,

and y is lexicographically greater than the kth element. If these conditions are satisfied, the kth element can be correctly positioned by sorting the middle section after the partitioning, and this sorting takes only $o(n)$ time.

Let us now turn our attention to the implementation details. Assume that we have preprocessed the input sequence as described in Section 2 for $b = 4n^{\beta}\lceil \log_2(n+1) \rceil$, where the constant β, $0 < \beta < 1$, will be determined later, and that the sequence has the form BEU. Let T be a shorthand notation for the union of sections E and U, and let $m = |T| = n - b$. The positioning of the kth element is accomplished as follows.

Take a random sample. Let $s = m^{\beta}$. Draw s random integers, independently and uniformly, from the range $\{1, 2, \ldots, m\}$ and store these numbers in B in encoded form. Clearly, this requires $O(s \log_2 n)$ time.

Sort the s integers in B using any in-place sorting algorithm. Each time two numbers are swapped their encodings in B are swapped as well. Therefore, this sorting requires $O(s \log_2 s \log_2 n)$ time.

Scan the sorted number sequence, starting from the back, determine the multiplicity of each number, and save the original number together with its multiplicity in encoded form in B. Because the B section is is so large, the old sequence and the new sequence cannot overlap during this process. Due to the manipulation of the encoded number representations the scan requires $O(s \log_2 n)$ time.

The integers in the number sequence indicate the indices of the elements to be chosen to the random sample, and the multiplicities tell how many times each element appears in the sample. That is, the sampling is done with replacement. Now scan the number sequence and gather the elements chosen to the sample together to the beginning of T, call the resulting section S. Again, since encoded representations are manipulated, this scan requires $O(s \log_2 n)$ time.

Choose elements x and y. Sort the elements in S using any in-place sorting algorithm. When in this process two elements are swapped, also the encoded indices and multiplicities in B are swapped. Thus, this requires $O(s \log_2 s \log_2 n)$ time.

Let α and γ be some fixed constants, $0 < \alpha < \beta < \gamma < 1$. If $k < m^{\gamma}$, let $\lambda = \nu = 2m^{\gamma}s/m$. On the other hand, if $k \geq m^{\gamma}$, let $\mu_\ell = (k-b)s/m$, $\mu_r = (k+b)s/m$, $\Delta_\ell = m^{\alpha}\mu_\ell^{1/2}$, $\Delta_r = m^{\alpha}\mu_r^{1/2}$, $\lambda = \max\{1, \lfloor \mu_\ell - \Delta_\ell \rfloor\}$, and $\nu = \min\{\lceil \mu_r + \Delta_r \rceil, s\}$. Now scan the S section to find the λth element of the sample, call it x, and the νth element, call it y. Since in this scan we have to determinate the value of multiplicities one by one, it takes $O(s \log_2 n)$ time. After this the indices of x and y are saved.

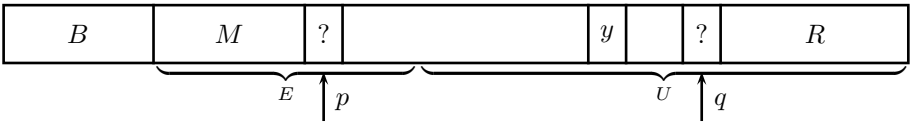
Undo element moves in T. Once more sort the elements in S, but now use their indices as keys. Again the encoded indices and multiplicities in B are moved accordingly. After this, move the elements in S back to their original positions. This is done backwards starting from the rear. In total, this requires $O(s \log_2 s \log_2 n)$ time.

Carry out two-way partitioning. If $k < m^{\gamma}$, perform a two-way partitioning of T using y as the partitioning element. Let M (R) denote the section

of elements lexicographically smaller than or equal to (larger than) y. Initially, both M and R are empty. We use a modification of the meeting cursors strategy by Hoare [7]. Initially, the two cursors p and q are at their respective ends of T, and they are gradually moved toward each other until they meet. The partitioning should be done carefully so that each element of U is compared to y exactly once, and that the comparison between the elements of E and y are avoided as far as possible. To achieve this, the relationship between z and y is determined before partitioning, and during the partitioning, when cursors p or q are in the E section, the elements pointed to by them are not compared to y, but the outcome of the initial comparison is recalled.

The partitioning procedure consists of three loops which all are similar in structure. In the first loop the cursors p and q are moved toward the centre until one of them meets the index of y. Depending on whether p or q meets the index first, there are two other loops. In both of these the movement of the cursors is continued until the cursors meet each other. Let us now consider the first loop in detail; the other two loops are similar and hence their consideration is omitted.

We maintain the following invariant in the first loop:



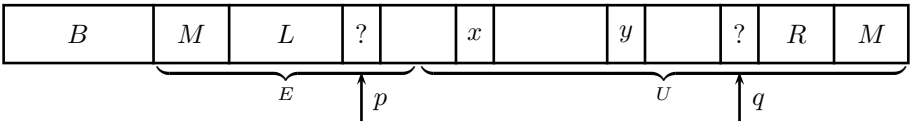
If the element pointed to by p is larger than y, the cursor is stopped; otherwise M is made one larger and the cursor p is incremented by one. If the element pointed to by q is smaller than y, also this cursor is stopped; otherwise R is made one larger and the cursor q is decremented by one. After both cursors are stopped, the elements pointed to by them are swapped. This makes both M and R one larger. Cursor p is incremented by one and cursor q is decremented by one, and the process is repeated until one of the cursors meets the index of y.

Clearly, this two-way partitioning is carried out in $O(m)$ time and at most $|U| + O(1)$ element comparisons are performed. For each element moved to M an element exchange might be necessary. With high probability the final size of the M section will be $o(m)$, so only $o(m)$ element exchanges will be necessary in this case.

Perform three-way partitioning. If $m^\gamma \leq k \leq \lceil m/2 \rceil$, perform three-way partitioning using x and y as the partitioning elements and collect the elements falling between them (including x and y) into M. Let L denote the section containing elements lexicographically smaller than x and R the section containing those larger than y. There are two symmetric cases depending on whether the index of x is smaller than or larger than the index of y. We consider only the first case here; the other case is similar.

Again the partitioning is based on the meeting cursors strategy. The partitioning routine consists of several symmetric loops depending on the relative positions of the cursors p and q and the index of x and the index of y. We con-

sider here only the first loop when the cursors have not yet met the index of x or the index of y. During the process we maintain the following invariant:



We maintain an M section both in the beginning of the sequence and in the end of the sequence, and then at the end swap the M sections into the middle. The elements in the E section are handled as in two-way partitioning in order to avoid unnecessary element comparisons.

Consider the invariant maintained in the first partitioning loop. Each element under consideration is always compared first to y, and thereafter to x if necessary. If the element pointed to by cursor p is larger than y, the cursor is stopped; if not and if it is larger than x, it belongs to M and it is exchanged with the first element of L, which makes the first M section one larger, moves the L section one position to the right, and thereafter cursor p is incremented by one. Otherwise, the element under consideration belongs to L, and that section is made one larger and cursor p is incremented by one. The movement of q backwards is done equally carefully using the M section at the end. After both cursors are stopped, the underlying elements are exchanged, and the process is repeated until one of the cursors meets the index of x or the index of y.

From this implementation it is clearly seen that three-way partitioning runs in $O(m)$ time. Each element of U is compared to y and possibly to x. Hence, the total number of element comparisons is bounded by $|U|+\min\{|U|, k+o(m)\}+O(1)$. With high probability the final size of the M section will be $o(m)$. Therefore, only the element exchanges involving an element of L are significant. With high probability the L section will get at most $k + o(m)$ elements, so the number of element exchanges performed is bounded by $k + o(m)$.

Reordering. Change the configuration of the sequence from $BVXZ$ (where L might be empty if $k < m^\gamma$) to $VBXZ$ by swapping the elements in B with a block of the same size at the end of L; if $|L| < |B|$, simply interchange the order of the blocks in these sections. Since B contains $O(s \log_2 n)$ elements, this interchange requires $O(s \log_2 n)$ time.

Finishing up. If $|L| \geq k - b$ or $|R| \geq n - k - b$, the correct positioning of the kth element is guaranteed by sorting the whole sequence using any in-place sorting algorithm. This takes $O(n \log_2 n)$ time, but the sorting will be necessary with negligible probability.

Otherwise, the correct positioning of the kth element is guaranteed by sorting the section BX in-place. This is the normal mode. With high probability the final size of the M section will be $o(m/\log_2 n)$, and since the size of the B section is bounded by $O(s \log_2 n)$, which is also $o(m/\log_2 n)$, this sorting requires $o(m)$ time.

4 Analysis

Now we are ready to prove our main theorem.

Theorem 1. *Our Las-Vegas algorithm carries out the positioning of the kth element in a multiset of size n in-place in $O(n)$ time using at most $n + k + o(n)$ pairwise element comparisons, and at most $k + o(n)$ element exchanges; these resource bounds are exceeded with probability at most $e^{-n^{\Omega(1)}}$.*

Proof. The space bound is obvious since all computations are performed in-place. The final sorting guarantees that the algorithm is of the Las-Vegas type. Since $s = m^\beta$, $0 < \beta < 1$, and $m \leq n$, all the phases where we operate with the sample take $O(s(\log_2 n)^2) = o(n)$ time. Hence, the computational costs are dominated by the preprocessing phase, one of the partitioning phases, and the final sorting phase. Consider first the normal mode, i.e., the case that the kth element falls in M and $|M| < n^\varepsilon$ for some ε, $0 < \varepsilon < 1$. Clearly, in this case the running time of the algorithm is $O(n)$, the number of pairwise element comparisons sums to $n + k + o(n)$, and that of element exchanges to $k + o(n)$.

The algorithm may fail to meet these resource bounds in six ways:

1. If $k < m^\gamma$ and M gets too small, so that the kth element is not in the union of B and M. We will be pessimistic and say that a failure occurs if y is lexicographically larger than the m^γth element of T.
2. If $k < m^\gamma$ and M gets too large, so that the sorting phase is too costly. Here we say that a failure occurs if $|M| > 4m^\gamma$.
3. If $k \geq m^\gamma$ and L gets too large, so that L can contain the kth element. This failure occurs if $|L| \geq k - b$.
4. If $k \geq m^\gamma$ and R gets too large, so R can contain the kth element. This failure occurs if $|R| \geq n - k - b$.
5. If $k \geq m^\gamma$ and the left boundary of M gets too far away to the left, so that the sorting phase becomes costly. We say that this failure occurs if x is lexicographically smaller than the $(k - b - \Delta_\ell m/s)$th element in T.
6. Finally, it is possible that $k \geq m^\gamma$ and the right boundary of M gets too far away to the right. We say that this failure occurs if y is lexicographically larger than the $(k + b + \Delta_r m/s)$th element in T.

The probabilities of these failures can be bounded above by using Chernoff bounds (see [15, Theorem 4.2 and Theorem 4.3]). We consider here the failure modes 2 and 3; the other four modes are handled in a similar way.

Failure mode 2. If this failure occurs, more than $4m^\gamma s/m$ of the sample elements are lexicographically larger than the $(2m^\gamma)$th element of T. Let $X_i = 1$, if the ith sample element is smaller than or equal to the $(2m^\gamma)$th element of T, and $X_i = 0$ otherwise. Thus, $\Pr[X_i = 1] = 2m^\gamma/m$, and $\Pr[X_i = 0] = 1 - 2m^\gamma/m$. For $X = \sum_{i=1}^{s} X_i$, $\mathbf{E}[X] = 2m^\gamma s/m$. Since X is binomially distributed and $X_1, X_2, \ldots, X_s$ are all independent, we can use [15, Theorem 4.2] to bound its upper tail probability:

$$\Pr\left[X > 4m^\gamma s/m\right] \qquad = \qquad \Pr\left[X > 2\mathbf{E}[X]\right]$$

$$\begin{aligned} &\overset{\text{Theorem 4.2}}{\leq} && e^{-m^{\gamma+\beta-1}/2} \\ &\overset{\beta=2/3 \text{ and } \gamma=5/6}{=} && e^{-m^{1/2}/2}. \end{aligned}$$

Failure mode 3. Recall that $\mu_\ell = (k-b)s/m$ and $\Delta_\ell = m^\alpha \mu_\ell^{1/2}$. If this failure occurs, less than $(k-b)s/m - \Delta_\ell$ of the sample elements are lexicographically smaller than the $(k-b)$th element of T. Define $X_i = 1$, if the ith sample element is lexicographically smaller than the $(k-b)$th element of T, and $X_i = 0$ otherwise. For $X = \sum_{i=1}^{s} X_i$, $\mathbf{E}[X] = \mu_\ell = (k-b)s/m$. Now we can bound the lower tail probability of X using [15, Theorem 4.3]:

$$\begin{aligned} \Pr[X < \mu_\ell - \Delta_\ell] &\overset{\delta=\Delta_\ell/\mu_\ell}{=} \Pr[X < (1-\delta)\mu_\ell] \\ &\overset{\text{Theorem 4.3}}{\leq} e^{-\mu_\ell \delta^2/2} \\ &= e^{-m^{2\alpha}/2}. \end{aligned}$$

For parameters $\alpha = 1/6$, $\beta = 2/3$, and $\gamma = 5/6$, we have that $\delta \leq 2e - 1$, so we can use the simplified Chernoff bound [15, Theorem 4.3].

Summing up. Since the probability of the union of events is at most the sum of their probabilities, the probability that some of the mentioned failures occurs is still negligible. Up to now we have expressed the failure probabilities as a function of m, but $m = n - s \geq n/2$ when n is large enough, so the failure probability is of the form $e^{-n^{\Omega(1)}}$. □

5 Concluding Remarks

Since the selection problem has been extensively studied in the literature, it was a surprise for us when we observed that Hoare's randomized algorithm was the only one that solves the positioning problem, operates in-place, is able to handle multiset data, and relies only on pairwise element comparisons. In this paper we described a more efficient randomized algorithm for positioning having all these desirable properties.

Assuming that assignments of the form $x \leftarrow A[i]$ and $A[i] \leftarrow A[j]$ are considered as element moves, an element exchange may require three moves. Using the hole technique (see, e.g., [6]), it is possible to implement our randomized positioning algorithm so that with high probability it will carry out at most $2k + o(n)$ element moves. If we are only interested in selection, it would be easy to modify the algorithm so that it will perform $o(n)$ element moves with high probability.

Due to the in-place requirement our algorithm is quite complicated but, if $o(n)$ extra space is available, the bit emulation can be avoided. Actually, in the CPH STL the implementation of the `nth_element` function — solving the positioning problem — is based on the randomized algorithm using $o(n)$ extra space. The only difference is that instead of a sorting routine a deterministic linear-time positioning routine is invoked, which guarantees $O(n)$ worst-case running time. For further details, see [21].

References

1. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, Time bounds for selection, *Journal of Computer and System Sciences* **7** (1973), 448–461.
2. S. Carlsson and M. Sundström, Linear-time in-place selection in less than $3n$ comparisons, *Proceedings of the 6th International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science* **1004**, Springer-Verlag, Berlin/Heidelberg (1995), 244–253.
3. W. Cunto and J. I. Munro, Average case selection, *Journal of the ACM* **36** (1989), 270–279.
4. R. W. Floyd and R. L. Rivest, Expected time bounds for selection, *Communications of the ACM* **18** (1975), 165–172.
5. V. Geffert, *Linear-time in-place selection in $\varepsilon \cdot n$ element moves*, Unpublished typescript (2000).
6. V. Geffert, J. Katajainen, and T. Pasanen, Asymptotically efficient in-place merging, *Theoretical Computer Science* **237** (2000), 159–181.
7. C. A. R. Hoare, Algorithm 65: Find, *Communications of the ACM* **4** (1961), 321–322.
8. J. Katajainen and T. Pasanen, Stable minimum space partitioning in linear time, *BIT* **32** (1992), 580–585.
9. J. Katajainen and T. Pasanen, Sorting multiset stably in minimum space, *Acta Informatica* **31** (1994), 301–313.
10. J. Katajainen, T. Pasanen, and J. Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996), 27–40.
11. J. Katajainen and T. A. Pasanen, In-place sorting with fewer moves, *Information Processing Letters* **70** (1999), 31–37.
12. P. Kirschenhofer, H. Prodinger, and C. Martínez, Analysis of Hoare's Find algorithm with median-of-three partition, *Random Structures and Algorithms* **10** (1997), 143–156.
13. T. W. Lai and D. Wood, Implicit selection, *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **318**, Springer-Verlag, Berlin/Heidelberg (1988), 14–23.
14. C. Levcopoulos and O. Petersson, Exploiting few inversions when sorting: Sequential and parallel algorithms, *Theoretical Computer Science* **163** (1996), 211–238.
15. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge (1995).
16. J. I. Munro, An implicit data structure supporting insertion, deletion, and search in $o(\log^2 n)$ time, *Journal of Computer and System Sciences* **33** (1986), 66–74.
17. J. T. Postamus, A. H. G. Rinnooy Kan, and G. T. Timmer, An efficient dynamic selection method, *Communications of the ACM* **26** (1983), 878–881.
18. A. Schönhage, M. S. Paterson, and N. Pippenger, Finding the median, *Journal of Computer and System Sciences* **13** (1976), 184–199.
19. R. Sedgewick, The analysis of Quicksort programs, *Acta Informatica* **7** (1977), 327–355.
20. J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7** (1964), 347–348.
21. L. Yde, Performance engineering the `nth_element` function, CPH STL Report 2002-4, Department of Computing, University of Copenhagen, Copenhagen (2002). Available at `http://www.cphstl.dk`.

Paging on a RAM with Limited Resources

Tony W. Lai*

IBM Toronto Laboratory
8200 Warden Avenue, Markham, Ontario L6G 1C7, Canada
tonylai@ca.ibm.com

Abstract. The paging problem is that of deciding which pages to keep in a cache of k pages to minimize the number of page faults for a two-level memory system. We consider efficient paging algorithms that require limited space and time on a unit-cost RAM with word size w. We present an implementation of the randomized marking algorithm of Fiat et al. that uses only $k + o(k)$ bits of space while supporting page requests in $O(1)$ worst-case time. In addition, we present a novel k-competitive deterministic algorithm called Surely Not Recently Used (SNRU) that approximates the Least Recently Used (LRU) algorithm. For any constant $1/k \leq \epsilon < 1/2$, $\mathrm{SNRU}(\epsilon)$ ensures that the $\lceil (1/2 - \epsilon)k \rceil$ most recently requested pages are in the cache, while using only $O(1/\epsilon)$ worst-case time per request and $2k + o(k)$ bits of space.

1 Introduction

In the paging problem, we have a two-level memory system consisting of a fast memory (known as the *cache*) that can hold k pages and a slow memory that can hold n pages, where n is typically much larger than k. A *paging algorithm* is presented with a sequence of requests to memory pages. If a requested page is already in the cache, then the algorithm incurs no cost; otherwise it incurs a *page fault* to evict a page in the cache and replace it with the requested page. The goal of a paging algorithm is to minimize the number of page faults by carefully deciding which pages to evict. The eviction decisions must be made *online*, without knowing future requests.

In this paper, we consider paging algorithms that require limited time and limited space. In particular, we seek algorithms that use minimal time per page access. Furthermore, we want algorithms that require only $O(k)$ bits of state information; we note that virtual memory systems typically use k or more bits to indicate whether pages are modified, are read-only, and so forth. For our model of computation, we assume a unit-cost RAM with w-bit words using a *restricted instruction set* according to the terminology of Andersson et al. [3]. That is, we allow comparison, addition, subtraction, bitwise AND and OR, and unrestricted bit shifts, but not multiplication or division. For randomized algorithms, we also allow an additional instruction $\mathrm{random}(x)$, which returns a random integer uniformly distributed in $\{1, \ldots, x\}$.

Paging has been extensively studied using *competitive analysis*. Roughly speaking, an online algorithm A is said to be *c-competitive* if, for every request sequence, the

* This research was supported in part by the IBM Toronto Centre for Advanced Studies.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 418–427, 2002.

cost of A is bounded (asymptotically) by c times the optimal cost for the sequence. The *competitive ratio* of A is the smallest c for which A is c-competitive. One of the best known algorithms is the Least Recently Used (LRU) strategy, which Sleator and Tarjan [17] showed to be k-competitive; Sleator and Tarjan also showed k to be the best possible competitive ratio for any deterministic algorithm. However, because LRU evicts the least recently requested page in the cache, LRU requires $\lceil \lg k! \rceil = \Theta(k \log k)$ bits of state information to store the order of page requests. Many newer competitive algorithms have been devised [14,11,13,4,9,7,1]; however, their space requirements exceed that of LRU.

Sleator and Tarjan proved that another well-known deterministic paging strategy, First-In First-Out (FIFO), is k-competitive. FIFO evicts the page that has been in the cache longest and requires only $\lceil \lg k \rceil$ bits of space. Another common paging algorithm is CLOCK, which conceptually maintains pages of the cache in a circular list. CLOCK tries to keep the more recently requested pages in the cache by maintaining a use-bit for each page and a global clock-hand pointer, for a total of $k + \lceil \lg k \rceil$ bits. Whenever a page is accessed, its use-bit is set to 1. During a page fault, CLOCK selects the page to evict by repeatedly advancing its clock hand and resetting use-bits to 0 until it finds a page whose use-bit is unset; hence CLOCK requires $\Theta(k)$ worst-case time. CLOCK is k-competitive since it is *conservative* [18], i.e. it avoids evictions during requests for pages in the cache, and it performs at most k evictions during any consecutive subsequence of requests to k or fewer distinct pages. Karlin et al. [10] proposed the Flush-When-Full (FWF) algorithm, which evicts all pages in the cache when the cache is full and a requested page causes a fault. FWF is k-competitive and requires only $\lceil \lg k \rceil$ bits. Although FIFO, CLOCK, and FWF all have the same competitive ratio as LRU, in practice the performance of CLOCK is comparable to that of LRU [5], but the performance of FIFO and FWF is substantially worse than that of LRU [18].

Perhaps the simplest randomized algorithm is RAND, which evicts a random page from the cache and hence requires no state information; Raghavan and Snir [15] showed that RAND has competitive ratio k. Fiat et al. [6] proposed the Randomized Marking Algorithm (RMA) and showed it to be $2H_k$-competitive, where $H_k = \sum_{i=1}^{k} 1/i$ is the kth harmonic number. Briefly, RMA divides the page request sequence into *phases* and maintains the set of *marked* pages, or the set of pages requested in the current phase. During a page fault, RMA evicts a random unmarked page. A simple implementation of RMA uses $O(1)$ worst-case time but $\Theta(k)$ words (or $\Theta(k \log k)$ bits) of space. It is straightforward to reduce the space bound to $k + O(\log k)$ bits: we maintain the set of marked pages using a bit vector, and we select a page to evict by randomly choosing pages in the cache until we find an unmarked page. Unfortunately, it can be shown that this implementation requires $\Theta(\log k)$ expected amortized time per page request.

We present new upper bounds on randomized and deterministic RAM paging algorithms in this paper. We present a new implementation of RMA that uses only $k + o(k)$ bits of space while supporting page requests in $O(1)$ worst-case time. Also, we present a new k-competitive deterministic algorithm called Surely Not Recently Used (SNRU). For any constant $1/k \leq \epsilon < 1/2$, $\mathrm{SNRU}(\epsilon)$ supports page requests in $O(1/\epsilon)$ worst-case time while using only $2k + o(k)$ bits of space. Unlike the FIFO, CLOCK, and FWF algorithms, which may evict the most recently requested pages, $\mathrm{SNRU}(\epsilon)$ prov-

ably approximates LRU by ensuring that the $\lceil (1/2 - \epsilon)k \rceil$ most recently requested pages remain in the cache.

The remainder of this paper is organized as follows. In Section 2, we first describe a useful structure called a *packed block summary*. In Section 3, we describe and analyze our new implementation of RMA. In Section 4, we present and analyze the SNRU algorithm. We finally give some concluding remarks.

2 Packed Block Summaries

In our paging algorithms, multiple pages in the cache may be eligible for eviction, so we first consider how to efficiently find a page in the set of eligible pages. Although the exact method depends on the particular paging algorithm, we describe a useful component called a *packed block summary* that supports operations in $O(1)$ worst-case time.

Without loss of generality, assume that the cache size k is fixed and that $4 \le k \le 2^w$. For $1 \le i \le k$, let P_i be the page in the ith location of the cache. We associate each P_i with a b-bit number λ_i called a *label*, for some chosen constant $b \le \lg k/2$, and we pack $\lambda_1, \ldots, \lambda_k$ together into $\lceil bk/w \rceil$ words. We assume that the pages eligible for eviction are all associated with some label c; hence, we want to efficiently find a page in the set $E = \{P_i : \lambda_i = c\}$.

We divide the cache into subblocks of $q = 2^{\lfloor \lg \lg k - \lg b - 1 \rfloor}$ pages, which are grouped into blocks of $r = \left\lfloor \frac{\lg k}{2\lceil \lg \lg k \rceil} \right\rfloor$ subblocks; hence each block has $\Theta(\log^2 k/(b \log \log k))$ pages. The packed block summary of a block B consists of r fields packed together, one for each subblock in B; each field contains $\lceil \lg(q+1) \rceil \le \lfloor \lg \lg k \rfloor$ bits to count the number of subblock pages with label c. Thus, a packed block summary contains at most $\lg k/2$ bits, and the total space required for all packed block summaries is $O(k/(qr) \cdot \log k) = O(bk \log \log k / \log k)$ bits.

A key observation is that, for any block, we can determine the ith page with label c in $O(1)$ time, for any i. Let B_j be the set of pages of the jth block, and let S_{jl} be the set of pages of the lth subblock of B_j; that is, $S_{jl} = \{P_{qr(j-1)+q(l-1)+1}, \ldots, P_{qr(j-1)+ql}\}$ and $B_j = \bigcup_{l=1}^{r} S_{jl}$. We seek the ith page of the set $B_j \cap E$. Observe that the packed block summary for B_j gives $e_{j1}, \ldots, e_{jr}$, where $e_{jl} = |S_{jl} \cap E|$ for all l. We use a precomputed lookup table that contains the smallest m such that $\sum_{l=1}^{m} e_{jl} \ge i$ and the sum $t = \sum_{l=1}^{m-1} e_{jl}$ for each possible value of i and $e_{j1}, \ldots, e_{jr}$. Thus, with one table lookup, we reduce the problem of finding the ith page with label c in block B_j to that of finding the $(i-t)$-th page with label c in subblock S_{jm}, or finding the $(i-t)$-th occurrence of the value c in $\lambda_{qr(j-1)+q(m-1)+1}, \ldots, \lambda_{qr(j-1)+qm}$. Note that we can extract this subsequence of labels in $O(1)$ time using shifts and bit operations. To obtain the offset of the jth occurrence of c among some labels $\lambda_{a+1}, \ldots, \lambda_{a+q}$, we simply look up a second precomputed table with entries for each possible value of j and $\lambda_{a+1}, \ldots, \lambda_{a+q}$. The total space required by the two lookup tables is $O((qr \cdot 2^{\lg k/2} + q \cdot 2^{bq}) \log k) = O(\sqrt{k} \log^3 k/(b \log \log k))$ bits.

We can update the packed block summaries in $O(1)$ time whenever we update a label. If we change the value of page P_i's label from $\lambda_i = c$ to $\lambda_i' \ne c$, then we decrement by 1

the packed block summary field corresponding to the subblock containing P_i; if $\lambda_i \neq c$ and $\lambda_i' = c$, then we increment this field by 1; otherwise we do nothing.

Note that the preceding implicitly assumes that we can multiply and divide by q and r in $O(1)$ time. Andersson [2] observed that, for a fixed integer divisor v, we can compute $u \operatorname{DIV} v$ in $O(1)$ time for any $0 \leq u \leq k$ if we can perform multiplication in constant time and we precompute $\lceil 2^{\lceil \lg k \rceil}/v \rceil$. Although our model of computation does not support multiplication directly, by maintaining a precomputed lookup table of all products of $\lceil \lg k/3 \rceil$-bit numbers, we can use standard multiple-precision techniques to multiply any two $\lceil \lg k \rceil$-bit numbers in constant time. The space required by the lookup table is only $O(k^{2/3} \log k)$ bits.

3 A New Implementation of RMA

As its name suggests, the randomized marking algorithm (RMA) belongs to a family of paging algorithms known as *marking algorithms*. A marking algorithm processes its page request sequence in phases. Initially, all pages are unmarked. Whenever a page is requested, it is marked. During a page fault, a marking algorithm can evict only unmarked pages. A phase ends just before the $(k+1)$-th distinct page is requested in the phase; at this point all pages are unmarked and a new phase begins. Thus, the only difference between marking algorithms is the method for choosing unmarked pages for eviction. RMA in particular evicts a page chosen uniformly at random from the set of unmarked pages.

Intuitively, in our implementation of RMA, we divide the cache into blocks of $s = \Theta(\log^2 k / \log\log k)$ pages, and we divide the set of blocks into $s+1$ subsets such that blocks within a subset have the same number of unmarked pages. To evict an unmarked page randomly with uniform distribution, we use three steps, each of which can be performed in $O(1)$ worst-case time. First, we randomly select a subset from the $s+1$ subsets of blocks using an appropriate distribution. Second, we select a block within the chosen subset randomly with uniform distribution. Finally, we select an unmarked page within the chosen block randomly with uniform distribution.

We proceed to describe our data structure for maintaining unmarked pages in detail. We use a bit vector of k bits to indicate marked and unmarked pages. In different phases, we alternate between using 1 and using 0 for marked pages. We divide the cache into blocks of $s = \left\lfloor \frac{\lg k}{2\lfloor \lg\lg k \rfloor} \right\rfloor \cdot 2^{\lfloor \lg\lg k - 1 \rfloor}$ pages. For convenience, assume k is a multiple of s, so that the cache consists of blocks $B_1, \ldots, B_{k/s}$. We use two auxiliary structures, one for finding pages labelled 0 and one for pages labelled 1. We describe only the former, since the latter is analogous.

Our label-0 auxiliary structure has three parts corresponding to the three eviction steps discussed above. Before describing the different parts of the structure, we consider how to select a block subset in $O(1)$ worst-case time. We define the *rank* $\mathcal{R}(B)$ of a block B to be the number of pages labelled 0 in the block. For $0 \leq i \leq s$, let R_i be the subset of blocks with rank i, and let $r_i = |R_i|$. Also, let $a_i = ir_i$ be the total number of pages labelled 0 in R_i; we call a_i the *weight* of R_i. To randomly select a block subset with probability proportional to its weight, we generate a random integer j in $\{1, \ldots, \sum_{i=1}^{s} a_i\}$ uniformly and choose the block subset R_l where l satisfies the

inequality $\sum_{i=1}^{l-1} a_i < j \leq \sum_{i=1}^{l} a_i$. To compute l efficiently, we rely on a data structure for maintaining partial sums devised by Raman et al. [16]. We state an (easily derived) generalized version of their result.

Lemma 1. *(Raman et al.) Suppose we have a RAM with word size w bits. For any given $N \leq 2^w$ and $\delta_{\max} = \log^{O(1)} N$, there is a data structure that maintains a sequence $A_1, \ldots, A_m$ of nonnegative integers, where $\sum_{i=1}^{m} A_i < N$, while supporting the following operations in $O(\log m / \log\log N)$ worst-case time.*

- $\mathrm{sum}(i)$*: Return* $\sum_{j=1}^{i} A_i$.
- $\mathrm{update}(i, \delta)$*: Add* δ *to* A_i*, for some integer* δ *such that* $|\delta| \leq |\delta_{\max}|$.
- $\mathrm{select}(j)$*: Return the smallest value of* i *such that* $\mathrm{sum}(i) \geq j$.

Furthermore, this data structure uses $O(m \log N)$ bits of space, but requires a precomputed table of $N^{o(1)}$ bits.

Therefore, the first part of our label-0 auxiliary structure consists of Raman et al.'s partial sums structure for the sequence of weights $a_1, \ldots, a_s$. Since the number of weights is $s < \lg^2 k / \lg\lg k$, the sum of weights is bounded by $\sum_{i=1}^{s} a_i \leq k$, and the absolute change in any weight after marking a page is bounded by s, we obtain a worst-case time bound of $O(1)$ for any operation, while using only $k^{o(1)} = o(k)$ bits, including space for precomputed tables. We note that Raman et al.'s structure uses Fredman and Willard's q-heap structure [8], which requires multiplication. However, because our numbers are all at most $\lceil \lg k \rceil$ bits, we can perform multiplications using a precomputed table as described in Section 2.

The second part of the label-0 structure consists of an array of partial sums of $r_0, \ldots, r_s$ and an array I containing a permutation of the k/s block indices such that $\mathcal{R}(B_{I[1]}) \leq \cdots \leq \mathcal{R}(B_{I[k/s]})$. Observe that, for any j, block $B_{I[j]}$ has rank i if $\sum_{l=0}^{i-1} r_l < j \leq \sum_{l=0}^{i} r_l$. Thus, to select a random block with rank i, we simply generate a random integer j uniformly in $\{1, \ldots, r_i\}$ and select the $(I[\sum_{l=0}^{i-1} r_l + j])$-th block of the cache, which requires $O(1)$ worst-case time.

The third part of the label-0 structure consists of an array of packed block summaries to keep track of pages labelled 0. To select a random page labelled 0 in a block B with rank i, we generate a random integer j uniformly in $\{1, \ldots, i\}$ and determine the jth page with label 0 in B using B's packed block summary as described in Section 2.

We observe that we can update the label-0 structure in $O(1)$ time whenever we relabel a page. Suppose we change the label of some page P from 0 to 1, where P is in some block B that has rank i. Since B's rank decreases to $i-1$ afterward, B moves from block subset R_i to subset R_{i-1}; since only two block subsets and thus only two weights a_i and a_{i-1} change, we update the partial sums of weights in $O(1)$ time. Next, we observe that only one of the partial sums of block subset sizes changes, namely $\sum_{l=0}^{i-1} r_l$; furthermore, we observe that exchanging two block indices in the array I is sufficient to ensure that I is ordered by nondecreasing block rank. Hence, we update the array of partial sums of block subset sizes and the array I of block indices in $O(1)$ time. Finally, we update the packed block summary of B in $O(1)$ time, so the total worst-case time required is $O(1)$. Similarly, the worst-case time required to update the label-0 structure when relabelling a page from 1 to 0 is only $O(1)$.

Theorem 1. *The implementation of RMA described above requires $O(1)$ worst-case time per page request and $k + o(k)$ bits of space, including the space for precomputed tables.*

Proof. We consider three possible types of page requests. After a request for a marked page in the cache, we do nothing. After a request for an unmarked page in the cache, we mark the page and update the label-0 and label-1 structures in $O(1)$ time as described above. After a request for a page not in the cache, we find an unmarked page to evict, bring in the requested page, mark the page, and update the label-0 and label-1 structures in $O(1)$ time as described above. Therefore, the worst-case time required for any page request is $O(1)$.

The space used is k bits for the bit vector of page labels, $O(\log^3 k/\log\log k)$ bits for the partial sums structure of weights, $O(\log^3 k/\log\log k)$ bits for the array of partial sums of block subset sizes, $O(k\log\log k/\log k)$ bits for the array of block indices, and $O(k\log\log k/\log k)$ bits for the packed block summaries. The space required for precomputed tables is $k^{o(1)} + O(k^{2/3}\log k) = o(k)$ bits. Therefore, the total space used is $k + o(k)$ bits. □

4 The Surely Not Recently Used Algorithm

In this section, we describe the Surely Not Recently Used (SNRU) paging algorithm, a deterministic marking algorithm that uses only $2k + o(k)$ bits of space and $O(1)$ worst-case time per page request while ensuring that the most recently requested pages remain in the cache. Because of the space bound, we retain only very coarse information about the history of page requests. When the number of unmarked pages is large compared to the number of marked pages, we avoid evicting recently requested pages by keeping track of the most recently requested unmarked pages, while retaining only a bare minimum of information about marked pages. When the number of unmarked pages is small compared to the number of marked pages, we "forget" the history of unmarked page requests, since all recently requested pages are marked; instead, we keep track of the most recently requested subset of marked pages in preparation for the start of the next marking phase.

We divide the pages into three classes U, U', and U'' of unmarked pages and three classes M, M^*, and M^{**} of marked pages, for a total of six classes. Intuitively, U, U', and U'' represent the least recently, semi-recently, and most recently requested unmarked pages, respectively; and M^{**}, M^*, and M represent the least recently, semi-recently, and most recently requested marked pages, respectively. We ensure that, at any time, at most four classes are nonempty. In particular, we ensure either that U' and U'' are empty or that M^{**} and M^* are empty. Hence, a 2-bit label is sufficient to indicate the class of each page.

For any constant $1/k \le \epsilon < 1/2$, the $\mathrm{SNRU}(\epsilon)$ algorithm works as follows. Let $t = \lceil (1/2 - \epsilon)k \rceil - 1$. We map each class to one of four possible label values or to a null value. We also maintain the counts u, u', u'', m^{**}, m^*, and m of the pages in classes U, U', U'', M^{**}, M^*, and M, respectively. There are five cases for handling a request for a page P.

Case 1a: $u'' + m < t \leq u' + u'' + m$, and M^{**} and M^* are mapped to nulls.
If P is not in the cache, evict an arbitrary page from U.

Case 1b: $m < t \leq u'' + m$, and M^{**} and M^* are mapped to nulls.
If P is not in the cache, evict an arbitrary page from $U \cup U'$.

Case 1c: $m \geq t$, and M^{**} and M^* are mapped to nulls.
Let c_I, c_{II}, c_{III}, and c_{IV} be the label values to which U, U', U'', and M are mapped, respectively. Over $\lfloor \epsilon k \rfloor$ page requests, move all pages in classes U' and U'' to class U by relabelling pages. Afterward, U' and U'' are empty; remap classes U' and U'' to nulls and classes M^{**}, M^*, and M to labels c_{II}, c_{IV}, and c_{III}, respectively. Meanwhile, if P is not in the cache, evict an arbitrary page from $U \cup U' \cup U''$.

Case 2a: $m < t \leq m^* + m$, and U' and U'' are mapped to nulls.
Let c_I, c_{II}, c_{III}, and c_{IV} be the labels to which U, M^{**}, M^*, and M are mapped, respectively. If P is not in the cache, there are two subcases. If U is not empty, evict an arbitrary page from U. Otherwise, the current phase has ended; unmark all pages by remapping classes U, U', U'', and M to labels c_{II}, c_{III}, c_{IV}, and c_I, respectively, and remapping M^{**} and M^* to nulls; go to case 1a.

Case 2b: $m \geq t$, and U' and U'' are mapped to nulls.
Let c_I, c_{II}, c_{III}, and c_{IV} be the labels to which U, M^{**}, M^*, and M are mapped, respectively. Over $\lfloor \epsilon k \rfloor$ page requests, move all pages in class M^* to class M^{**} by relabelling pages. Afterward, M^* is empty; exchange the labels of M^* and M, i.e. remap classes M^* and M to labels c_{IV} and c_{III}, respectively. Meanwhile, if P is not in the cache, there are two subcases. If U is not empty, evict an arbitrary page from U. Otherwise, the current phase has ended; unmark all pages by remapping classes U, U', U'', and M to labels c_{II}, c_{III}, c_{IV}, and c_I, respectively, and remapping M^{**} and M^* to nulls; go to case 1b.

Afterward, we add page P to class M by relabelling P, and we update the appropriate class size counts.

To evict pages efficiently, we divide the cache into blocks of size $s = \left\lfloor \frac{\lg k}{2\lfloor \lg \lg k \rfloor} \right\rfloor \cdot 2^{\lfloor \lg \lg k - 2 \rfloor}$. For each possible label value c, we maintain an array of packed block summaries for c, along with two arrays of block indices for maintaining an unordered, doubly-linked list of all blocks with nonzero summary entries or, equivalently, all blocks containing pages labelled c. Observe that we can update these auxiliary arrays in $O(1)$ worst-case time whenever we relabel a page, since we can update the packed block summaries and (if necessary) the doubly-linked lists in $O(1)$ time. Furthermore, we can locate a page in any given class in $O(1)$ worst-case time as follows. We determine the label c to which the class maps, find the block referenced by the head of the doubly-linked list for c, and then use the block's summary entry to locate the first page labelled c in the block.

Before proving that SNRU ensures the most recently requested pages are in the cache, we first state some useful properties of SNRU. Observe that SNRU is indeed a marking algorithm since it marks each requested page, it evicts only unmarked pages, and it unmarks all pages only if a page fault occurs when all pages are marked.

Lemma 2. SNRU *is a marking algorithm.*

In the following, we say that a page request is of *type-1a* if the request is handled entirely by case 1a of the SNRU algorithm. We define *type-1b*, *type-1c*, *type-2a*, and *type-2b* requests similarly. We say that a page request is of *type-2a/1a* if the preconditions of case 2a are initially satisfied but the page request causes the current phase to end, resulting in a transition to case 1a. Similarly, we say that a request is of *type-2b/1b* if the preconditions of case 2b are initially satisfied but the request causes the current phase to end, resulting in a transition to case 1b.

Lemma 3. *During any page request, the following statements are true.*

1. *The class M contains the $|M| = m$ most recently requested pages.*
2. *During any type-2a request, or immediately before any type-2a/1a request, the set $M^* \cup M$ contains the $|M^* \cup M| = m^* + m$ most recently requested pages.*
3. *During any type-1a request, or after remapping classes in any type-2a/1a request, the set $U'' \cup M$ contains the $|U'' \cup M| = u'' + m$ most recently requested pages, and the set $U' \cup U'' \cup M$ contains the $|U' \cup U'' \cup M| = u' + u'' + m$ most recently requested pages.*
4. *During any type-1b request, or after remapping classes in any type-2b/1b request, the set $U'' \cup M$ contains the $|U'' \cup M| = u'' + m$ most recently requested pages.*

Proof. For brevity, we merely sketch the proof; details are in [12]. Observe that, if we maintain a set S initially containing the $|S|$ most recently requested pages, such that we always add the most recently requested page to S, and we never delete any pages from S without deleting all pages in S, then S always contains the $|S|$ most recently requested pages. Clearly these conditions are true for M, so statement (1) follows immediately.

For a given type-2a or type-2a/1a request, the previous page request is either of type-1c, type-2a, or type-2b; if the previous request is of type-1c or type-2b, then we must have performed class remapping, and immediately after remapping classes we know that $M^* \cup M$ contains the $|M^* \cup M|$ most recently requested pages. Therefore, we can show statement (2) by induction, noting that our observation above for class M can be applied to the set $M^* \cup M$.

Similarly, it is straightforward to show statement (3) by induction. We note that a type-1a request must be preceded by a type-1a or type-2a/1a request. Furthermore, for a type-2a/1a request, initially we know that M contains the $|M|$ most recently requested pages and that $M^* \cup M$ contains the $|M^* \cup M|$ most recently requested pages; hence immediately after class remapping, the set $U'' \cup M$ contains the $|U'' \cup M|$ most recently requested pages, and $U' \cup U'' \cup M$ contains the $|U' \cup U'' \cup M|$ most recently requested pages.

Lastly, it is straightforward to show statement (4) by induction. We note that a type-1b request must be preceded by a type-1a, type-1b, or type-2b/1b request. Furthermore, for a type-2b/1b request, initially M contains the $|M|$ most recently requested pages; hence immediately after class remapping, the set $U'' \cup M$ contains the $|U'' \cup M|$ most recently requested pages. □

Theorem 2. *For any constant $1/k \leq \epsilon < 1/2$, the $\lceil (1/2 - \epsilon)k \rceil$ most recently requested pages are in the cache after any page request using* $\mathrm{SNRU}(\epsilon)$.

Proof. Assume that, immediately before the current page request, the $\lceil (1/2 - \epsilon)k \rceil$ most recently requested pages prior to the current request are in the cache. If the current request is for a page in the cache, then $\mathrm{SNRU}(\epsilon)$ does not evict any pages, so the $\lceil (1/2 - \epsilon)k \rceil$ most recently requested pages must be in the cache after the request.

Otherwise, if the current page request causes a page fault, it is sufficient to show that the $t = \lceil (1/2 - \epsilon)k \rceil - 1$ most recently requested pages prior to the current request are not eligible for eviction. If the current page request is of type-1a or type-2a/1a, then from Lemma 3 and the construction of SNRU, we know that the $u' + u'' + m \geq t$ most recently requested pages are not eligible for eviction. If the current request is of type-1b or type 2b/1b, then again Lemma 3 and the construction of SNRU imply that the $u'' + m \geq t$ most recently requested pages are not eligible for eviction. For all other types of requests, the number of marked pages is at least t, and we know that marked pages are not eligible for eviction. The theorem follows. □

We now analyze the performance of SNRU. Karlin et al. [10] proved that any marking algorithm is k-competitive, which immediately implies the following.

Corollary 1. SNRU *is k-competitive.*

Finally, we analyze the time and space bounds for SNRU.

Theorem 3. *For any constant $1/k \leq \epsilon < 1/2$, $\mathrm{SNRU}(\epsilon)$ requires $O(1/\epsilon)$ worst-case time per page request and $2k + o(k)$ bits of space, including the space for precomputed tables.*

Proof. To handle a page request, we may relabel up to $\lceil k/\lfloor \epsilon k \rfloor \rceil < 2/\epsilon$ pages, remap classes to different label values, locate a page to evict, relabel the requested page, and update the counts of class sizes. The worst-case time to remap classes, locate a page to evict, and update class size counts is $O(1)$. The worst-case time to relabel a single page is $O(1)$, so the worst-case time for all page relabellings is $O(1/\epsilon)$. The claimed time bound follows.

The space required is $2k$ bits for the page labels, $O(1)$ bits for the mapping of classes to label values, $O(\log k)$ bits for the counts of class sizes, $O(k \log\log k / \log k)$ for the packed block summaries, and $O(k \log\log k / \log k)$ for the arrays of block indices used to maintain doubly-linked lists. The space required for precomputed tables is $O(k^{2/3} \log k)$ bits. Therefore, the total space used is $2k + o(k)$ bits. □

5 Concluding Remarks

We have presented new upper bounds for deterministic and randomized paging algorithms using the RAM model of computation by demonstrating a new implementation of RMA and a new deterministic algorithm approximating LRU that require only $O(1)$ worst-case time per page request and $O(k)$ bits of space. Many open questions remain concerning upper and lower bounds on the competitive ratio of paging algorithms with limited state information or limited time. For memoryless algorithms, Raghavan and Snir [15] showed a matching upper and lower bound of k for the competitive ratio. For randomized algorithms in general, McGeoch and Sleator [14] and Fiat et al. [6] obtained

a matching upper and lower bound of H_k for the competitive ratio. For paging algorithms with $O(k)$ bits of space, one open question in particular is whether we can obtain a lower bound on the competitive ratio greater than H_k or an upper bound less than $2H_k - 1$, which Achlioptas et al. [1] proved to be the competitive ratio of RMA.

References

1. D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234:203–218, 2000.
2. A. Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 135–141, 1996.
3. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, pages 427–436, 1995.
4. A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50:244–258, 1995.
5. W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9:560–595, 1984.
6. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.
7. A. Fiat and M. Mendel. Truly online paging with locality of reference (extended abstract). In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 326–335, 1997.
8. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimumn spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.
9. S. Irani, A. R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25:477–497, 1996.
10. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:70–119, 1988.
11. A. R. Karlin, S. Phillips, and P. Raghavan. Markov paging. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 208–217, 1992.
12. T. W. Lai. Paging on a RAM with limited resources. Technical Report TR-74.187, IBM Toronto Laboratory, 2002. To appear.
13. C. Lund, S. Phillips, and N. Reingold. IP-paging and distributional paging. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 424–435, 1994.
14. L. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Journal of Algorithms*, 6:816–825, 1991.
15. P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. *IBM Journal of Research and Development*, 38:683–707, 1994.
16. R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Proceeding of the 7th International Workshop on Algorithms and Data Structures*, pages 426–437, 2001.
17. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
18. N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

An Optimal Algorithm for Finding NCA on Pure Pointer Machines*

A. Dal Palú, E. Pontelli, and D. Ranjan

Department of Computer Science
New Mexico State University
{apalu,epontell,dranjan}@cs.nmsu.edu

Abstract. We present a simple, arithmetic-free, efficient scheme to compress trees maintaining the nearest common ancestor (NCA) information. We use this compression scheme to provide an $O(n + q \lg \lg n)$ solution for solving the NCA problem on *Pure* Pointer Machines (PPMs) (i.e., pointer machines with no arithmetic capabilities) in both the static and dynamic case, where n is the number of add-leaf/delete operations and q is the number of NCA queries. This solution is optimal.

1 Introduction

The Nearest Common Ancestor (NCA) Problem can be broadly defined as follows: Given a tree T and two nodes $x, y \in T$, find the nearest common ancestor of x and y in T. In the *static* version of the problem, T is known in advance. In the *dynamic* version T is modified via some pre-defined operations. The difficulty of the problem depends on what kind of operations are allowed for tree modification. Some typical operations considered in this context are *add-leaf* that allows addition of leaves to the tree, *delete* which allows deletion of a node, *link* which allows linking of a tree as a subtree of a node in another tree, etc. In the *offline* version, both T and the NCA queries are known in advance.

NCA problem has been studied extensively. For the static case, the original work by Harel and Tarjan [9] provides a constant-time algorithm for performing the $nca(x, y)$ operation after a linear-time preprocessing of the tree. This result was later simplified and parallelized by Schieber and Vishkin [14,8]. Bender and Farach-Colton [3] provided an effectively implementable algorithm which provides $O(1)$ time execution of $nca(x, y)$ operation with linear-time preprocessing of the tree. In all these algorithms, complexity analysis is done assuming the RAM model. For the dynamic case, Tsakalidis [16] provides algorithms with $O(\lg h)$ worst-case time for the *nca* operation and almost amortized $O(1)$ time for add-leaf and delete in a dynamic tree, where h is the height of the tree. The algorithm is developed for an *Arithmetic Pointer Machine (APM)* model under the uniform cost measure assumption (constant time arithmetic for $\Theta(\lg n)$-size integers). This result on APMs has been recently improved in [1], where it is shown that the NCA problem can be solved in worst-case $O(\lg \lg n)$ time per

* Research is supported by NSF grants CCR-9900320, EIA-0130887, CCR-9875279, CCR-9820852, and EIA-9810732.

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 428–438, 2002.

operation, and that it can be solved in $O(n+q \lg \lg n)$ time on APMs, where n is the number of link operations and q is the number of *nca* queries. The work by Cole and Hariharan [5] provides the ability to insert (leaves and internal nodes) and delete nodes (leaves and internal nodes with one child) in a tree, and execute the $\mathsf{nca}(x, y)$ operation in worst-case constant time. Both methods make use of arithmetic capabilities of the respective machine models. For the offline version, a linear-time algorithm on APMs was given by Buchsbaum et al. [4].

In this work we focus on solving the problem on *Pure Pointer Machines (PPMs)*, i.e., pointer machines that do not allow constant-time arithmetic operations (see [2] for details about this model). We present a simple, arithmetic-free, efficient scheme to compress trees maintaining the nearest common ancestor (NCA) information. This compression scheme is different from the ones previously used in literature [9,5]. In particular, it does not make any use of arithmetic and it is very local in nature and hence seems eminently parallelizable. We use this compression scheme to provide an $O(n + q \lg \lg n)$ solution for solving the NCA problem on PPMs in both the static and the dynamic case, where n is the number of add-leaf/delete operations and q is the number of queries. This solution is optimal because of a known matching lower bound [9]. Moreover, it has the same complexity as that of an optimal solution on APMs. Hence, our result shows that use of arithmetic is not essential for doing NCA calculations optimally. This is intellectually satisfying because intuitively NCA is a structure and not an arithmetic problem. We use this optimal solution to the NCA problem to improve the solutions for several other problems. Proofs and details of the results presented are omitted due to lack of space and can be found in [6].

2 A Compression Scheme for Trees

The compression algorithm we propose starts from the initial tree $T = T_0$ and repeatedly performs two types of compressions, generating a sequence of trees: $T_0, T_1, T_2, \ldots$ until a tree T_k containing a single node is obtained. The trees in this sequence are used to build a second tree structure (H-tree), that summarizes the NCA information of T. The key property of the H-tree is that its depth is at most logarithmic in the number of nodes T. This allows a fast NCA calculation.

Given T_i, T_{i+1}^L the result of *leaf-compression* of T_i,is obtained by merging each leaf of T_i with its parent. If a leaf ℓ is merged with its parent $parent(\ell)$, then $parent(\ell)$ is said to be the *direct representative* of ℓ. A *path-compression* of a tree T_{i+1}^L returns a tree T_{i+1}, where each path containing only nodes with a single child and ending in a leaf of T_{i+1}^L is replaced by the head of such path. If a path containing nodes $v_0, v_1, \ldots, v_k$ is compressed to the node v_0, then v_0 is said to be the *direct representative* of $v_0, \ldots, v_k$. A *compression* of a tree T_i is the tree T_{i+1}, where T_{i+1} is the path-compression of T_{i+1}^L, and T_{i+1}^L is the result of a leaf-compression on T_i. In this notation let $T = T_0$.

Fig. 1 shows an example of repeated compression of T. Both leaf and path-compressions start at the frontier of each tree. Each time a leaf-compression is applied, all leaves are merged with their parents. For example, in Fig. 1 leaf-

compression removes nodes $10-15$ (Fig. 1.1) by merging them with their parents (Fig. 1.2). A path-compression merges all paths ending in a leaf into their heads. For example, in Fig. 1.3 the path composed by nodes $4, 6, 9$ has been collapsed to the single node 4 (node 4 is the direct representative of 4,6 and 9). The tree is compressed starting from the leaves and moving towards the root. In Fig. 1 we have marked the representatives of each compression with darker nodes.

The H-Tree: In order to compute *nca* queries in optimal time, it is useful to collect the information about representatives in a separate tree, called *Horizontal Tree (H-tree)*. The H-tree, H, can be constructed from the sequence of trees obtained during the compression process (e.g., the trees shown in Fig. 1). If a leaf-compression is applied to node v in tree T_i and ℓ is the direct representative of v in such compression, then node v is connected to the last occurrence of ℓ in a tree T_j $(i < j)$, where ℓ appears in T_j as a direct representative of a leaf-compression. If all the children of a node a in T_i are leaf-compressed at the same time, then the representative of such children is node a in T^L_{i+1} (as for leaves $10, 11$ in Fig. 1). If the children of a are leaf-compressed at different points in time (e.g., the children of 1 in Fig. 1), then the representative of such leaf is the last occurrence of its direct representative in a tree as representative in a leaf-compression. If a path-compression is applied, then all nodes in the path are connected to the head of the list in the next tree (see Fig. 1). Such node is the representative of all nodes in the path. H is obtained using the single node in the last compressed tree (e.g., the node in T_2 in Fig. 1) as the root and using the links between nodes and representatives as edges (e.g., the dark edges in Fig. 1).

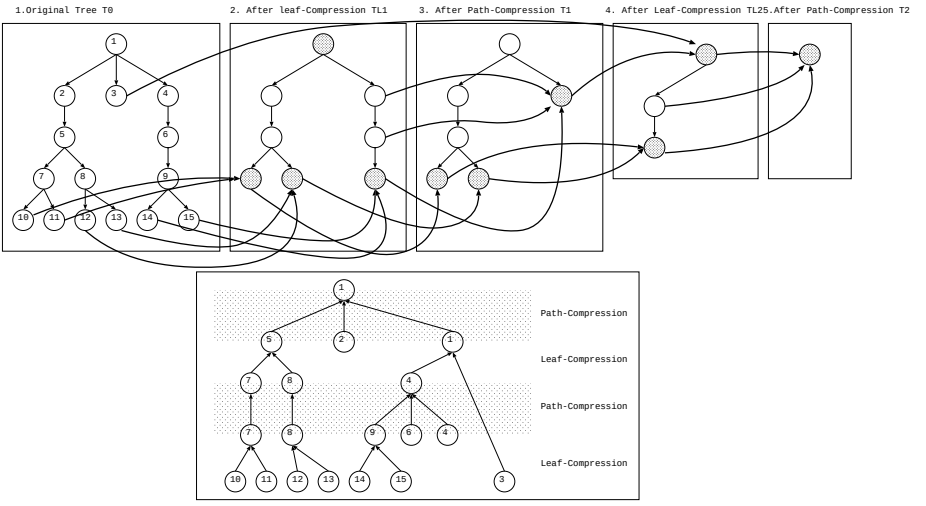


Fig. 1. Building the H-tree

Observe that the leaves of the original tree are leaves in H although H might have additional leaves. Also, each internal node in T is present in H as each internal node is either a representative in a leaf compression or is involved in a path compression. More importantly, observe that H has at most $2n$ nodes,

since each node can appear in H because of (possibly many) leaf-compressions at most once and can be involved in a path-compression at most once. Moreover, if a node $v \in T$ appears twice in H, then it must be the case that one occurrence of v in H is due to the fact that v was a head in a path compression and is a direct representative in leaf compressions which precede the aforementioned path compression. Note that one occurence of v in H must be a child of the other occurrence of v in H. Next lemma provides a result critical to the efficiency of the compression scheme. Let $subtree_T(v)$ be the subtree of T rooted at node v.

Lemma 1. *If a node v of T still exists in T_k then the $subtree_T(v)$ has at least 2^k nodes. Furthermore, let n be the number of nodes in T and let k be the minimum integer such that T_k has a single node. Then $k \leq \lg n$. In other words, T gets compressed to a single node within $\lg n$ compressions.*

Answering NCAs Using H-trees: Given the query $nca(x, y)$, where $x, y \in T$, it is possible to answer the query using H. In particular, computation of the NCA of two nodes in T can be computed by first computing an NCA of the "entry-points" for x and y in H. The entry-point in H for x is simply the lower (or the only) occurence of x in H. We provide an intuitive description of this method—the algorithm called nca_H is presented in Section 3. We show now that the H-tree preserves enough nca information from T. Let z be the $nca_H(x, y)$. If z is a representative of a leaf-compression, then z is also the nca of $x, y \in T$. Otherwise let $z_0, z_1, \ldots, z_k$ be the nodes belonging to the path that has been compressed to z. There are two distinct nodes z_i, z_j in this path such that the subtree rooted at z_i (z_j) contains x (y) Thus, the nca of x and y is the highest node between z_i and z_j, and answering an NCA query in T boils down to computing an NCA query in H. From Lemma 1, we can infer that the height of H is $O(\lg n)$. In [11] it was shown that there is a (dynamic) PPM algorithm given a tree with height h allows the computation of the NCA of any two nodes in the tree in worst case time complexity $O(\lg h)$ per query. Using this result, we can compute the nca of x, y in H in worst-case time $O(\lg \lg n)$. This allows the computation of the NCA in T with worst-case time complexity $O(\lg \lg n)$.

3 An Algorithm for NCA Queries in the Static Case

In the next sections, as in [1], we use a more general definition of nca: $nca(x, y) = (z, z_x, z_y)$, where z is the nca of x and y, and if $x = z$ ($y = z$) then $z_x = z$ ($z_y = z$) else z_x (z_y) is the ancestor of x (y) such that z_x (z_y) is a child of z.

In the static case T is pre-processed before any query is executed. Conceptually, the preprocessing creates $2k + 1$ trees, named $T_0, T_1^L, T_1, T_2^L, \ldots, T_k^L, T_k$. The T_0 tree is equal to T and each other tree is the result of the corresponding compression. To improve the time and space required, T_i and T_i^L trees are not explicitly created. Each time only the nodes being encountered for the first time are created anew, except that during a path compression a new node is created for the head of the path. H is composed of the union of all nodes created during the various compression phases.

Data structures: For each w in T, let $entry(w)$ be the entry point of w in H. Note that each node in a tree T_i or T_i^L is a copy of a node existing in T;

if the node $v \in H$ is a copy of node $u \in T$, then $node(v)$ is a pointer to u. Let $children(v)$ be a pointer to a list of nodes containing the children of node v in H, and let $parent(v)$ denote the parent of node v in H. During the preprocessing phase, we will also make use of two flags associated to each node of H: *(i)* a flag $leaf\text{-}compr(v)$ that indicates whether the node v is the result of a leaf compression; and *(ii)* a flag $is\text{-}leaf(v)$ that indicates if the representative produced by a leaf compression is a leaf in the new tree.

Construction of the H-tree: To answer the NCA queries, we require the ability to efficiently compare the depth of nodes that appear on the same branch. This can be accomplished by making use of the data structures developed to solve the *Temporal Precedence ($\mathcal{TP}$)* problem [13,10]. Using the optimal scheme from [10] we can build a data structure in $O(n)$ time that allows us to compare depths in $O(\lg \lg n)$ time. The preprocessing algorithm used to construct H is described in detail in [6]. With one visit of T, one can create the leaves of T_0, by simply copying each leaf v of T to a new node u and updating both $entry(v) = u$ and $node(u) = v$. After this, the process proceeds by repeatedly applying leaf-compression and path-compression. The process stops as soon as we are left with a tree containing a single node. The last $T_k, k \leq \lg(n)$, has only one node.

Lemma 2. *The time needed to construct H for an n-node tree T is $O(n)$.*

Leaf Compression: For each leaf v in T_i, a leaf-compression is applied. A new representative w for the parent of v in T ($entry(parent(v)) = w$) is added to the tree T^L_{i+1}, if this is the first time a node is compressed to w. After this check, the node v is compressed to its parent w. Once all the leaves of T_i are processed, for each node w in T^L_{i+1} we set the flag $is\text{-}leaf(w) = true$ if w is a leaf in T^L_{i+1}. The $is\text{-}leaf$ flag indicates the nodes where the next path-compression will start.

Path compression: Path-compression is initiated from each node v in T^L_i, such that $is\text{-}leaf(v) = true$. Execution of path-compression starting from v leads to the addition of a node w to T_{i+1}. w will be the representative of the compressed path starting from v. When the path compression stops, the node w will be assigned to the correct node in T—i.e., the representative of the path. The following iteration assigns to each v in the current path the direct representative w, and tries to extend the path leading to the root with $parent(v)$, if $parent(v)$ is still part of the path. When the iteration stops, the current node v is the head of the path, and its representative w in T_{i+1} is a copy of the node v.The check that verifies whether a node is part of a path can be implemented in constant time without the use of arithmetic (as described before).

Final Preprocessing Step: Once H has been constructed, it is further processed to to enrich it with the data structures required to support the computation of NCAs in $O(\lg h)$ time (h being the height of H). The details of this data structure have been described in [11]. The process requires the creation of the p-list data structure for each node in H. Since the height of H is at most $O(\lg n)$, the p-list of each node v contains $depth(v)$ elements, and each element of a p-list can be built in $O(1)$ time, the process of creating these additional data structures requires $O(n \lg \lg n)$ time. It is possible to improve this pre-processing time, reducing it to $O(n)$, through the use of the MicroMacroUniverse approach

described in [1] and in Section 4: in this case H is partitioned in μTrees (Micro-Trees) with depth at most $\lg \lg n$.

Answering NCA Queries: To compute the *nca* of $x, y \in T$, the algorithm $nca_H(x, y)$ works as follows: **(1)** Compute the $nca(entry(x), entry(y)) = (z, z_x, z_y)$, $z \in H$. The computation can be performed using the algorithm in [11]. **(2)** If $leaf\text{-}compr(z) = true$ then return ($node(z)$,$node(z_x)node(z_y)$). **(3)** Otherwise z is the result of a path compression. In this case if z_x is higher than z_y in H, the *nca* is (node(z_x), w_1, w_2), where w_1 is the node corresponding to the child of z_x that is ancestor of $entry(x)$ in H and w_2 is the node in T corresponding to the left sibling of z_x in H. w_1 can be obtained by using p-lists in time $O(\lg \lg n)$ and w_2 can be found in constant time. The case where z_y is higher than z_x is symmetric. The test to check if z_x is higher than z_y can be performed in time $O(\lg \lg n)$ using the previously maintained depth information.

4 Dynamic NCA Algorithm

The algorithm follows an approach similar to the MicroMacroUniverse described in [1,7] in conjunction with repeated use of the static algorithm described in Section 3. Let n be the number of nodes in T. T is partitioned into a forest S of disjoint subtrees, called *μTrees* (Micro-Trees). The roots of μTrees are collected in another tree, called the *MTree* (Macro-Tree) (see Fig. 2). The MTree essentially compresses the nodes of each μTree into its root node and preserves the structure of T on these resulting root nodes. The height of each μTree is restricted to be at most $c_T(n)$. When a node v is added to T, if the μTree containing $parent(v)$ has a depth greater than $c_T(n)$, then a new μTree rooted in v is created. To obtain the optimal result, the MicroMacroUniverse approach is applied again to the μTrees. For the MTree the scheme used is based again on partitioning the tree into disjoint subtrees. However this partitioning is more dynamic in nature, since the subtree to which a node belongs can change.

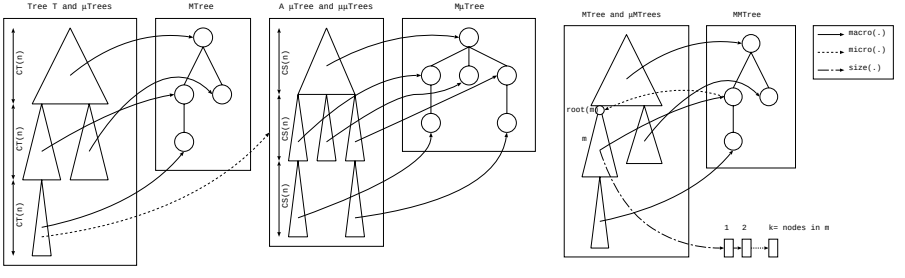


Fig. 2. Tree Structures Involved In The Dynamic Algorithm

In order to answer an *nca* query, our algorithm first solves the problem in the MTree and then refines the solution by working in the appropriate μTree. We denote by nca_S the nca algorithm used for the μTrees in S and nca^* the NCA algorithm used for the MTree. Each time a node v is inserted in T, v is also inserted in a data structure that collects the relative height information

of the nodes, using a Temporal Precedence list. As described in Section 3, this information will be required to perform an *nca* query using p-lists.

The nca^* algorithm: We present an algorithm to compute the *nca* with a cost of $O(N + Q \lg \lg N)$, with N add-leaf/delete operations and Q *nca* queries in the MTree. The problem is solved by using another MicroMacroUniverse approach applied to the MTree. The intuitive idea is to dynamically maintain a set of trees pre-processed with the static algorithm presented in Section 3. Let us call each of these trees *μMTree* (Micro-Macro-Tree). The preprocess of a μMTree allows us to efficiently solve each *nca* query on that tree, using the nca_H algorithm presented earlier. Each root of a μMTree is represented by a node in another tree, called *MMTree* (Macro-Macro-Tree). We will show that the MMTree has a "small" depth—thus, simpler NCA algorithms (e.g., the one based on p-lists [11]) can be used here to provide efficient NCA computations.

Let the *preprocess* of a μMTree T_m be the static preprocess described in Section 3 applied to the tree $subtree_{MTree}(root(T_m))$. Thus, when T_m is pre-processed, all other $\mu\mu$Trees hanging on T_m are merged in a single new μMTree. The basic idea is to wait to re-preprocess a μMTree T_m until the number of nodes in $subtree_{MTree}(root(T_m))$ has doubled since the last preprocess of T_m. To answer an *nca* query we first solve the problem with the p-list nca algorithm [11] on the MMTree and then we "refine" that solution using the nca_H on the μMTree associated to the result obtained from the MMTree.

Dynamic insertions: Recall that the MTree is partitioned into a set of μMTrees. Each μMTree is represented by its root in another tree, called MMTree. Let T_m be a μMTree and v_m the node representative of T_m in the MMTree. Let us also define $root(T_m)$ the root of T_m in MTree, $micro(v_m)$ a pointer to $root(T_m)$, $macro(T_m)$ a pointer to v_m. We also maintain the size of T_m by keeping a pointer $size(T_m)$ that points to a list that has length $|T_m|$. $size(T_m)$ is created and inserted with a number of nodes equal to the number of nodes in the $subtree_{MTree}(root(T_m))$, when T_m is pre-processed. The $size(T_m)$ list is used a decrementing counter to decide when to do another preprocess.

Each time a node v is inserted in the MTree as child of w, a new μMTree T_m is created and the representative of $root(T_m) = v$ is added in the MMTree as child of $macro(T_w)$, where T_w is the μMTree containing w. For each μMTree T'_m corresponding to a node on the path P from $macro(T_m)$ to the root of the MMTree, we update the number of new nodes added in the $subtree_{MTree}(root(T'_m))$ by 1. This can be done decrementing the "counter" $size(T'_m)$ by one, that is, shifting the pointer $size(T'_m)$ to the next node in that list. If a μMTree T_m has consumed all nodes in $size(T_m)$, then T_m has to be pre-processed. Let us call v_h the highest node in the path P considered, such that $micro(v_h)$ has to be processed. The preprocess is applied on $subtree_{MTree}(v_h)$, which become the new μMTree. All nodes in $subtree_{MTree}(v_h)$ are deleted and replaced by the node v_h. The $size(micro(v_h))$ list is initialized with the insertion of a number of nodes equal to the number of nodes contained in $subtree_{MTree}(v_h)$.

As we will show in the next Lemma, the MMTree has depth less or equal to $O(\lg h)$, where h is the depth of the MTree. Thus the update of counters may

be performed $O(\lg h)$ times for each insertion of a node in the MTree. Since a node is added in the MTree every $c_T(n)$ insertions in T and $h = n/c_T(n)$, if $c_T(n) \geq \lg n$, then the total time spent in updating the counters in all insertions in T is $O(n)$. Notice that each time a node v is involved in a preprocess resulting in a tree T', the size of T' is at least twice the size of the tree T'' which contained v before the preprocess. Then it follows:

Lemma 3. *A node v in the MTree of size t is involved in at most $\lg t$ distinct preprocesses. Immediately after a preprocess, if a path P starting from a node in MMTree and ending on a leaf has k nodes, then the total number of nodes in μMTrees represented by nodes in P is at least 2^{k-1}.*

It follows that the MMTree has depth at most $\lg N$, where N is the number of nodes of the MTree. The MTree has at most $n/c_T(n)$ nodes. Choosing $c_T(n) = \lg n \lg\lg n$, the MTree has depth at most $O(n/(\lg n \lg\lg n))$. Applying lemma 3 we conclude that the MMTree has depth at most $\lg n$. Thus the *nca* in the MMTree can be computed in time $O(\lg\lg n)$ using p-lists.

We now show that n insertions in T will cost $O(n)$ to maintain the MTree structure. We showed that a preprocess of a tree with t nodes in the static case costs $O(t \lg\lg t)))$. Let N be the number of nodes in the MTree. From Lemma 3 we know that a node v in the MTree is involved in at most $\lg N$ preprocesses. Each of them will cost $l \lg\lg l$, where l is the size of the tree preprocessed. Thus for each v the amortized cost per process is $\lg\lg l \leq \lg\lg N$ and the cost per node is $\lg N \lg\lg N$. Recalling that $N = n/c_T(n)$ and $c_T(n) = \lg n \lg\lg n$, the total amortized cost is $O(1)$ per insertion in T.

nca **queries:** Let us show how to compute the $nca^*(x, y)$ with x and y in MTree. It is possible to find x' and y' μMTrees containing x and y respectively in constant time—e.g. once a node is pre-processed, we can directly set in the node a pointer to the corresponding μMTree. If x' and y' are in the same μMTree T_m return $nca_H(x, y)$ using the previously pre-processed H tree for T_m. Otherwise, we can compute the $nca(root(x'), root(y')) = (z, z_x, z_y)$ on the MMTree, using p-lists. If $z_x = z$ then the result is given by $nca_H(x, parent(micro(z_y)))$. Otherwise, the result is $nca_H(parent(micro(z_x), parent(micro(z_y))$. The algorithm nca_H requires $O(\lg\lg n)$ time and the p-list algorithm used for the MMTree requires $O(\lg t)$, where $t \leq \lg(n/c_T(n))$ and $c_T(n) = \lg n \lg\lg n$. This allows us to conclude that total time is $O(\lg\lg n)$.

The nca_S **algorithm:** In this section we provide an algorithm with an amortized time complexity of $O(1)$ per insertion and worst-case complexity of $O(\lg\lg n)$ per query for the μTrees. The scheme uses the standard Micro-MacroUniverse approach on the μTrees. The optimal solution is computed combining an optimal solution on MμTree and an optimal solution on $\mu\mu$Tree. Choosing $c_S(n) = \lg\lg n$ and recalling that $c_T(n) = \lg n \lg\lg n$, all the μTrees can be processed in $O(n)$ time. To find the nca_S of two nodes x and y in a μTree, we combine a p-list search on MμTree and a brute force search applied on the resulting $\mu\mu$Tree. Clearly this requires $O(\lg\lg n)$ time.

Observe that the deletions are not performed explicitly, instead the deleted nodes are just marked as such. The marked nodes are deleted at the time when

they are involved in the next preprocessing. We don't update the counters when nodes are deleted. This doesn't affect our analysis, because the number of operations is greater than the number of nodes in T.

5 Applications of Optimal NCA Algorithm

The availability of an optimal solution to the NCA problem allows us to improve the solution of other interesting problems on PPMs. In particular it allows us to obtain an optimal solution to the generalized linked list problem [15]. This problem is the maintenance of a linked list where new elements x can be inserted immediately after any existing element y. The two operations allowed are: $\mathsf{insert}(x, y)$ and $\mathsf{compare}(x, y)$ which returns true iff x occurs before y in the list. The following algorithm allows us to optimally solve this problem on PPMs.

To preserve the relationships between inserted nodes, we maintain a tree T' processed to answer *nca* queries and a data structure maintaining a TP order. Every time an $\mathsf{insert}(x, y)$ is done, the node x is inserted as rightmost child of y in T'. Since the tree T' cannot support an ordering of children of a node, we also insert the node y in the Temporal Precedence data structure. Note that a leftmost depth first visit of T' reconstructs the list. Thus the *nca* of two nodes either precedes both the nodes in the order or is equal to one of them. To answer a query $\mathsf{compare}(x, y)$, we find the $nca(x, y)=(z, z_x, z_y)$ in T. If $z=z_x$ then return true, because $x=z$ and x is an ancestor of y in T'. If $z=z_y$ then return false, because y is an ancestor of x and y cannot have been inserted before x. Otherwise return $\mathsf{precedes}(z_x, z_y)$ in the TP order.

Another problem whose solution can be improved using this optimal NCA solution is the $\mathcal{OP}$ problem described in [12]. The $\Theta(\lg^2 n)$ solution proposed in [12] can be improved to a $O(\lg n \lg\lg n)$ solution by using the optimal generalized linked list scheme proposed above. All the open problems described in [11] can be solved optimally on PPM using the optimal NCA solution presented here.

6 APMs vs. PPMs

The commonly used APM model allows constant time arithmetic on $\Theta(\lg n)$ sized integers. The PPM does not allow such arithmetic, and one has to account for simulating any arithmetic needed, when analyzing the running time. The arithmetic can be simulated in PPMs by explicitly representing the integers via $\Theta(\lg n)$ sized lists. This entails that a generic translation (that just simulates the arithmetic) of APM algorithms to PPMs will incur a polylog penalty. More precisely an algorithm $\mathcal{A}$ that runs in time $t(n)$ on an APM and uses any arithmetic at all, will take time $t(n)\lg^k n$ for some $k > 0$ on a PPM. We present an interesting result about the NCA problem. We show that any optimal APM algorithm for the NCA problem can be converted into a PPM algorithm without incurring any penalty.

Theorem 1. *An APM algorithm $\mathcal{A}$ solving the NCA problem with amortized cost of $O(\lg^k n)$ per insertion and worst-case cost $O(\lg \lg n)$ per query, can be translated into a PPM algorithm with an amortized cost of $O(1)$ per insertion and worst-case cost $O(\lg \lg n)$ per query.*

7 Conclusions and Remarks

We have defined a novel compression scheme and used it for solving the NCA problem optimally on PPMs both in the static and the dynamic case. The compression scheme is interesting due to its simplicity, locality properties, efficiency and arithmetic-free nature. However, it is not essential for obtaining the optimal NCA algorithm for the PPMs due to the following remarkable theorem that is proved in the appendix C making use of the MicroMacroUniverse scheme presented in Section 4. We have also shown that for the NCA problem, it is possible to *totally* avoid the polylog penalty that one has to incur in a generic translation of an algorithm designed for APMs to PPMs. This gives rise to the question: Is there any natural problem for which the optimal solution on PPMs is provably logarithmically worse as compared to the optimal solution on APMs. As of now, we believe that the worst such known penalty incurred is $O(\lg \lg n)$ [13]. It will be especially interesting if there is no problem at all where the logarithmic penalty has to be incurred because that will show that the generic translation is non-optimal.

References

1. S. Alstrup and M. Thorup. Optimal Pointer Algorithms for Finding Nearest Common Ancestors in Dynamic Trees. *Journal of Algorithms*, 35:169–188, 2000.
2. A.M. Ben-Amram. What is a Pointer Machine? In *SIGACT News*, 26(2), 1995.
3. M.A. Bender and M. Farach-Colton. The LCA Problem Revisited. In *Proceedings of LATIN 2000*, pages 88–94. Springer Verlag, 2000.
4. A.L. Buchsbaum et al. Linear-Time Pointer-Machine Algorithms for Least Common Ancestors. In Procs. ACM STOC, ACM Press, 1998.
5. R. Cole and R. Hariharan. Dynamic LCA Queries on Trees. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 235–244. ACM/SIAM, 1999.
6. A. Dal Palù, E. Pontelli, D. Ranjan. An Optimal Algorithm for Finding NCA on Pure Pointer Machines. NMSU-TR-CS-007/2001, `www.cs.nmsu.edu`, 2001.
7. H.N. Gabow and R. E. Tarjan A linear-time algorithm for a special case of disjoint set union *J. Comput. System Sci* 30 (1985), 209-221.
8. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge Press, 1999.
9. D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal of Computing*, 13(2):338–355, 1984.
10. E. Pontelli and D. Ranjan. A Simple Optimal Solution for the Temporal Precedence Problem on Pure Pointer Machines. TR-CS-006/2001, New Mexico State U., 2001.
11. E. Pontelli and D. Ranjan. Ancestor Problems on Pure Pointer Machines. In *LATIN*, 2002.
12. D. Ranjan et al. An Optimal Data Structure to Handle Dynamic Environments in Non-deterministic Computations. *Computer Languages*, (to appear).

13. D. Ranjan, E. Pontelli, L. Longpre, and G. Gupta. The Temporal Precedence Problem. *Algorithmica*, 28:288–306, 2000.
14. B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors. *SIAM J. Comp.*, 17:1253–1262, 1988.
15. A. Tsakalidis. Maintaining Order in a Generalized Linked List. *ACTA Informatica*, (21):101–112, 1984.
16. A.K. Tsakalidis. The Nearest Common Ancestor in a Dynamic Tree. *ACTA Informatica*, 25:37–54, 1988.

Amortized Complexity of Bulk Updates in AVL-Trees

Eljas Soisalon-Soininen[1] and Peter Widmayer[2]

[1] Department of Computer Science and Engineering, Helsinki University of Technology, P.O.Box 5400, FIN-02015 HUT, Finland
ess@cs.hut.fi

[2] Institut für Theoretische Informatik, ETH Zentrum/CLW, CH-8092 Zürich, Switzerland
widmayer@inf.ethz.ch

Abstract. A bulk insertion for a given set of keys inserts all keys in the set into a leaf-oriented AVL-tree. Similarly, a bulk deletion deletes them all. The bulk insertion is simple if all keys fall in the same leaf position in the AVL-tree. We prove that simple bulk insertions and deletions of m keys have amortized complexity $O(\log m)$ for the tree adjustment phase. Our reasoning implies easy proofs for the amortized constant rebalancing cost of single insertions and deletions in AVL-trees. We prove that in general, the bulk operation composed of several simple ones of sizes $m_1, \ldots, m_k$ has amortized complexity $O(\Sigma_{i=1}^{k} \log m_i)$.

1 Introduction

A *bulk* (or *batch* or *group*) *insertion* is the operation of inserting a whole set of keys at once, in a single transaction. Similarly, a *bulk deletion* deletes a whole set. In the case of leaf-oriented search trees and keys with a linear order, this set of keys, called the *bulk* (or *batch* or *group*), is sorted before the insertion starts, for efficiency reasons. Sorting improves the performance considerably, because then consecutive keys of the bulk go to locations that are close to each other. In this way, for external memory structures, the number of required disk accesses is small. Similarly, for main memory structures, the number of cache misses is small. A *simple bulk insertion* inserts a set of keys whose search phase ends at the same leaf, called the *position leaf*, and a *simple bulk deletion* removes all keys in an interval.

Algorithms for bulk insertions and deletions into spatial indexes have been presented e.g. in [2]. Bulk insertions for one-dimensional search structures have been considered in [4,6,8,9,10,13,14]. These papers represent different application areas: [13] applies bulk insertions to inverted indexing of document databases, with special emphasis on concurrent searches. In [8] the method of [13] is adjusted to the environment of real-time databases, and in [6] the buffer-tree idea of Lars Arge [2] is used to speed up bulk B-tree insertions into large data warehouses. In [3] a linear time bulk deletion algorithm for B-trees is presented. Bulk insertions and deletions are also needed in differential indexing [14].

M. Penttonen and E. Meineche Schmidt (Eds.): SWAT 2002, LNCS 2368, pp. 439–448, 2002.

Complexity of bulk insertions and deletions, without the corresponding search phases, for different one-dimensional search trees is analyzed in [4,9,10]. In [4,10], the worst case complexities $O(\log n + \log^2 m)$ for simple bulk insertion and $O(\log n + \log m)$ for deletion, were proved for red-black and AVL-trees. Here n is the size of the original tree and m the size of the bulk. In [9] the amortized complexity $O(\log m)$ was proved for simple bulk insertion in the case of (a, b)-trees [11].

In this paper we concentrate on the analysis of the bulk operations when applied to AVL-trees. AVL-trees are interesting, because they are main-memory search trees with small height and they provide simple rebalancing operations by rotations that allow efficient concurrency control. Our new results state that a simple bulk insertion and a simple bulk deletion both have amortized time complexity $O(\log m)$, where m is the size of the bulk. If the bulk operation contains k simple bulks, then the amortized complexity is $O(\Sigma_{i=1}^{k} \log m_i)$, where m_i is the size of the ith simple bulk. This analysis implies a new and simple way to prove the constant amortized complexity for single insert and delete operations.

2 Amortized Complexity of Single Operations

In order to be able to prove the desired amortized complexity results we need new proofs for the case of singleton operations. The results of this section have been previously obtained by [12] (insertion) and [16] (deletion), but we needed new proofs in order to generalize them to bulk operations.

AVL-trees were introduced in [1] in 1962. AVL-trees are binary trees in which nodes either have two children or no children; the former nodes are called internal nodes and the latter leaves. A binary search tree is AVL, if for each internal node u with children v_1 and v_2,

$$|height(v_1) - height(v_2)| \leq 1.$$

We say that internal node u is *strictly balanced* (resp. *non-strictly balanced*) if $height(v_1) = height(v_2)$ (resp. $|height(v_1) - height(v_2)| = 1$).

We consider leaf search trees, in which keys are stored in the leaves and internal nodes contain router information. A single update operation contains the *search phase* that determines the *position leaf* of the operation, the *actual operation*, and finally the *rebalancing* of the tree. The *actual insertion* consists of replacing the position leaf by a subtree with one internal node and two leaves, and the *actual deletion* removes the position leaf and replaces its parent by its sibling. Rebalancing includes not only rotations but resetting the balance information stored in the nodes. In this paper, we assume that node heights are stored as the balance information; thus, node heights need be reset.

In the worst case, a single insertion needs one rotation but $O(\log n)$ increases in the node height, where n denotes the number of keys in the tree, and a single deletion needs $O(\log n)$ rotations and $O(\log n)$ decreases in the node height (see [7]).

Let $u_0, u_1, \ldots, u_k$ be a path from the root u_0 to a leaf u_k of an AVL-tree, and assume that u_k is the position leaf of a new insertion. Moreover, let i be minimal such that all $u_i, \ldots, u_k$ are strictly balanced. Then, after the insertion the heights of all nodes $u_i, \ldots, u_k$ must be increased. Finally, if u_i is not the root and the sibling of u_i was lower than u_i, one single or double rotation is done at u_{i-1}, after which the tree is again an AVL-tree (see [7]).

In order to show that insertion has amortized constant rebalancing complexity it is enough to count the height increase operations because the other tasks (the actual insertion and the possible final rotation) have constant worst case complexity. We apply the potential function technique by Tarjan [15]. For each internal node u the potential $\Phi_I(u)$ is defined as follows:

$$\Phi_I(u) = \begin{cases} 1, \text{ if } u \text{ is strictly balanced and at least one child of } u \\ \quad \text{is an internal node,} \\ 0, \text{ otherwise.} \end{cases}$$

(We could have defined $\Phi_I(u) = 1$, if u is strictly balanced, and $\Phi_I(u) = 0$, otherwise, but the above definition makes it simpler to prove the desired amortized complexity result for bulk insertion.) The potential $\Phi_I(T)$ of a tree T is defined as the sum of the potentials of its internal nodes. Here T is an AVL-tree or a tree that is not yet AVL but is being processed into an AVL-tree by rebalancing after an insertion into an AVL-tree.

What we have to show is that the number of node height increases is less than or equal to cn, where c is a constant and n is the number of insertions performed thus far:

Lemma 1. Assume that insertions are applied into an initially empty AVL-tree. Each actual insertion will increase the potential of the tree by at most 1. Each rotation included in the rebalancing phase of an insertion will increase the potential of the tree by at most two. Each height increase operation when applied to a node u, such that at least one child of u is an internal node, will decrease the potential of the tree by 1.

Proof. Performing the actual insertion, i.e., replacing the position leaf by a subtree of three nodes, can increase the whole potential by at most 1. The root of this subtree is strictly balanced, but by definition its root has potential 0. Second, if the potential of an internal node u in the path from the root to the position leaf is increased from 0 to 1, then this cannot have happened for any node above u, because the height of u is not changed. Thus u must be the only such node, and the potential is increased by at most one.

Clearly, a single or double rotation may increase the number of strictly balanced nodes by at most 2. Thus any insertion, excluding the height increase operations, increases the potential of the tree by at most 3.

If the height of a node is increased, this is because the node was strictly balanced and the height of exactly one of its children has been increased. Thus the height increase means that the node is no more strictly balanced, and the potential of the node is decreased by one, except when both of its children were leaves. □

Lemma 1 implies that the total number of height increase operations is $O(n)$, where n is the number of insertions performed. Thus we have:

Theorem 1. Assume that insertions are performed into an initially empty AVL-tree. The amortized rebalancing complexity of each insertion is constant. □

As noted by [12], mixed insertions and deletions do not have amortized constant complexity. This is seen by considering an AVL-tree with all nodes strictly balanced. Then inserting a new key will cause $\Omega(\log n)$ height increases, and deleting this new key or its sibling will cause $\Omega(\log n)$ height decreases. By repeating alternating insertions and deletions we obtain a sequence of n insertions and deletions that require $\Omega(n \log n)$ balance changes.

However, for pure deletions the constant amortized complexity can be obtained [16]. We now give a simple proof of this fact. For each internal node u the potential $\Phi_D(u)$ is defined as follows:

$$\Phi_D(u) = \begin{cases} 1, \text{ if } u \text{ is non-strictly balanced,} \\ 0, \text{ otherwise.} \end{cases}$$

The potential $\Phi_D(T)$ of a tree T is defined as the sum of the potentials of its internal nodes. Here T is an AVL-tree or a tree that is not yet AVL, but is being processed into an AVL-tree by rebalancing after a deletion from an AVL-tree.

In the case of deletion we have to count the height decrease operations and the rotations.

Lemma 2. Assume that deletions are applied into an AVL-tree with n leaves. Each actual deletion will increase the potential of the tree by at most 1. Each last rotation included in the rebalancing process will increase the potential by at most 1. Each height decrease operation will decrease the potential of the tree by at least 1, and each rotation before the last rotation in the rebalancing process decreases the potential by at least 1.

Proof. Assume that the potential of an internal node u is increased by one because of deletion. This means that the height of one of the children of u is decreased by one and u has become non-strictly balanced. Then for no node above u the potential can have increased because the height of u has not changed, and thus u is the only such node.

It is straightforward to see that a rotation performed at a node v will increase the potential by one, if the rotation yields a tree with the same height as v had before the rotation, and that otherwise, i.e., when the rotation is height decreasing, the potential will decrease by at least one. Thus the last rotation can increase the potential of the tree by at most one.

Then consider a height decrease operation. The height of a node is decreased, if the height of its higher child is decreased. Thus the node has become strictly balanced, and the potential has decreased by one.

It is straightforward to see that a rotation performed at a node v is the last rotation, if it is not height decreasing, i.e., the result of the rotation is as high as v. Thus, all rotations before the last one are height decreasing, and each of them decreases the potential of the tree by at least one. □

Theorem 2. Assume that n deletions are applied into an AVL-tree with n keys. Then altogether cn rebalancing operations, where c is a constant, will be performed in conjunction with these deletions. In other words, the amortized rebalancing complexity of each deletion is constant.

Proof. Initially the potential of the tree is less than n. The upper bound of the number of last rotations is n, and the upper bound of the number of all other rebalancing operations is the initial potential plus the total possible increase of the potential, which is $2n$ by Lemma 2. Thus the total number of rebalancing operations is bounded by a constant. □

3 Bulk Insertions into AVL-Trees

Let T' be an AVL-tree with n keys, and assume that m distinct sorted keys with the same position leaf are inserted into T'. Assume further that an AVL-tree S, called an *update tree*, has been constructed from these keys together with the key in the common position leaf. A *simple bulk insertion* contains the *actual bulk insertion*, in which S is substituted for the position leaf, and *rebalancing* the resulting tree, denoted T, i.e., transforming T into an AVL-tree.

The idea of bulk insertion is that changing the structure of S is avoided as long as possible, such that S is gradually moved towards the root by applying rotations. This means that no rotation is allowed in the parent p of the root of S, as long as a rotation is possible above p.

In this way we can obtain a time bound $O(\log m)$ for *merging* S into T, i.e., for rebalancing T up to the point where there is no balance conflict in the grandparent of the root of S. The merging consists of steps that perform rotations. The first step has $T_0 = T$ as input and the input T_i of each following step is the output of the previous step.

By the *level* of a subtree B we mean the level of the root of B plus the height of B. Let $u_1, \ldots, u_s$ be the path from the root of S to the root of T_i, and denote by $B_1, \ldots, B_{s-1}$ the subtrees of T_i rooted at the siblings of $u_1, \ldots, u_{s-1}$. We will show that at each intermediate stage of the merging process T_i is in balance except at nodes $u_2, \ldots, u_s$, and, moreover, for the levels of B_i holds:

$$level(B_j) - \max\{level(B_k) | 1 \leq k < j\} \leq 1 \tag{1}$$

for $j = 3, \ldots, s-1$, and

$$level(B_1) - level(B_2) \leq 1, level(B_2) - level(B_j) \leq 2 \tag{2}$$

for $j = 3, \ldots, s-1$. Initially, these conditions clearly hold because tree T' was in balance.

Tree T_{i+1} is constructed from T_i, $i \geq 0$, as follows. First set the heights of the parent p and the grandparent g of the root of S in T_i according to the heights of their children. If there is no height difference of more than one between the child nodes of g, or g is the root of T_i, then $T_{i+1} = T_i$ and the process terminates. Otherwise perform a rotation at the parent node g' of node g, such that the

update tree S will be moved one step closer to the root. There are two cases to consider depending on how the path $u_1, \ldots, u_s$ starts.

Case 1. g is the left (right) child of g', p is the left (right) child of g, and the root of S is either the left or right child of p. In this case a single rotation to the right (left) at g' is performed. If before the rotation the level of B_3 was one or two larger than the level of B_2, at most three additional rotations are necessary at the root of the new subtree containing B_2 and B_3. It is straightforward to check that the conditions (1) and (2) now hold for the resulting tree T_{i+1}.

Case 2. g is the left (right) child of g', p is the right (left) child of g, and the root of S is either the left or right child of p; see Figure 1. In this case a double

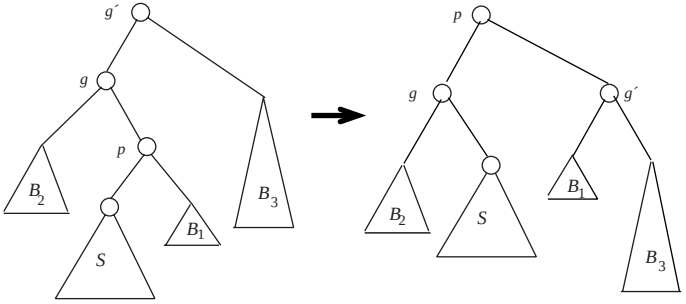


Fig. 1. Case 2: A double rotation.

rotation at g' is performed. Notice here that the level of B_3 can be $O(k)$ larger than the level of B_1, where k denotes the number of steps after the previous application of Case 2. Thus at most $O(k)$ additional rotations can be needed to achieve balance at the subtree containing B_1 and B_3. It is again straightforward to verify that conditions (1) and (2) hold for the resulting tree T_{i+1}.

There is an i such that $T_{i+1} = T_i$, because each step moves S towards the root. Moreover, it is easy to see that the imbalance at the grandfather of the root of S is at least one smaller in T_{i+1} than in T_i. Thus the number of steps needed is $O(\mathit{height}(S)) = O(\log m)$.

After completing the merging process, i.e., after having found that $T_{i+1} = T_i$, it is possible that the parent of the root of S is not in balance. This imbalance can easily be resolved in time $O(\log m)$, see [10]. The resulting subtree S' might have become one lower, but the parent of the root of S' remains in balance. By condition (2) the level of B_j, $2 < j \leq s-1$, can have been 2 less than the level of B_2. Thus one rotation is possibly needed in the path up to the root of T_i. In addition to this, node heights may need be increased in the whole path up to the root.

We have:

Theorem 3. Let T' be an AVL-tree, and let T be the tree obtained from T' by replacing one of its leaves by an update tree S. The number of rotations

needed to rebalance T is $O(\log m)$, where m is the size of S. The time needed to rebalance T, excluding the height increase operations, is $O(\log m)$. The time needed to perform the height increase operations is $O(\log n)$, where n is the size of T'. □

We are not only interested in the worst case complexity of simple bulk insertions, but merely in their amortized complexity.

Using the potential function Φ_I as defined for single insertions we show that the amortized complexity of simple bulk insertions is $O(\log m)$, where m is the size of the bulk. The potential Φ_I is defined for all trees that may appear at any intermediate stage of merging or of final rebalancing.

Lemma 3. Assume that single insertions and simple bulk insertions are applied to an initially empty AVL-tree. Each actual (single) insertion will increase the potential of the tree by at most 1. Each rotation included in the rebalancing phase of a single insertion will increase the potential of the tree by at most two.

Each actual bulk insertion, the merging process, and the remaining rotation altogether will increase the potential by $O(\log m)$, where m is the size of the bulk.

Each height increase operation applied to a node u, such that at least one child of u is an internal node, will decrease the potential of the tree by 1.

Proof. For single insertions this is Lemma 1. For simple bulk insertions, first notice that an update tree can be constructed such that no internal node except those whose both children are leaves is strictly balanced. Thus, the update tree of a simple bulk insertion can be assumed to contain zero potential. In the same way as for a single insertion, hanging an update tree in place of a leaf will increase the potential only by at most 1. By Theorem 3 the number of performed rotations in a simple bulk insertion is $O(\log m)$, and thus the possible increase in the potential is $O(\log m)$. The stated potential decrease is concluded as in the proof of Lemma 1. □

Lemma 3 together with Theorem 3 implies:

Theorem 4. Assume that single insertions and simple bulk insertions are performed into an initially empty AVL-tree. The amortized rebalancing complexity of each single insertion is constant, and the amortized complexity of each simple bulk insertion is $O(\log m)$, where m is the size of the bulk. The bulk insertion composed of several simple ones of sizes $m_1, \ldots, m_k$ has amortized complexity $O(\Sigma_{i=1}^{k} \log m_i)$. □

4 Bulk Deletion

We assume that we are given an AVL-tree T, an interval $[L, R]$ of keys that have to be deleted from T, and at most two leaves l_L and l_R that contain the smallest key $\geq L$ and the largest key $\leq R$, respectively. We also assume that l_L and l_R have been obtained by searching in T for L and R, respectively. Because of these two searches we know the lowest common ancestor, denoted c, of L and R, and also for L (resp. R) the node, denoted d_L (resp. d_R), which is the node with the largest (resp. smallest) router value smaller (resp. larger) than the router of c

in the path from c to l_L (resp. l_R). The nodes d_L and d_R are called the *left* and *right deletion roots*, respectively.

The *actual simple bulk deletion* of the keys in $[L, R]$ is performed by traversing from l_L to d_L, and from l_R to d_R, cf. Figure 2. Let $v_1 v_2 \ldots v_k$ be the path from l_L to d_L. (The path from l_R to d_R is handled correspondingly.) First delete v_1 from tree T. Assume then that the process has advanced up to node v_i, $1 < i \leq k$. If the leftmost child of v_i has been deleted, then also delete v_i. If only the rightmost child of v_i has been deleted, then replace v_i by its undeleted child. Continue the process with $i = i + 1$ until $i = k$.

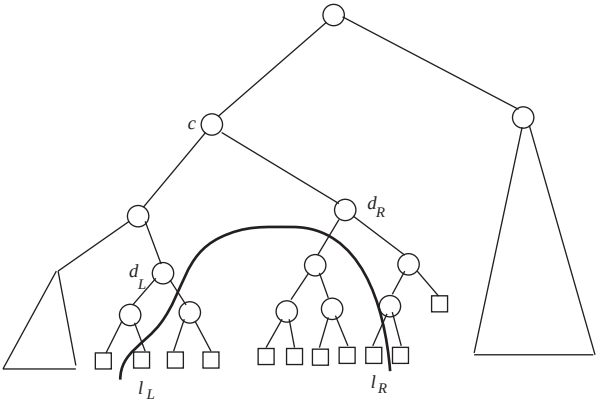


Fig. 2. Bulk deletion: the removed part lies inside the bold line.

After the actual deletion has been finished, rebalancing may be needed at d_L and d_R to complete the process of simple bulk deletion. These rebalancing tasks can be done in time $O(\log m)$, where m is the number of keys in the interval $[L, R]$. This time bound comes from the observation that the height of d_L (and of d_R) is bounded by $O(\log m)$, and thus the height difference of the children of d_L (and of d_R) is $O(\log m)$. The rebalancing time is then implied by Lemma 2 in [10]. After having put d_L and d_R in balance it is possible that the lowest common ancestor requires rebalancing. Again, the height difference of the children of c is $O(\log m)$ and thus rebalancing of c takes time $O(\log m)$.

All above rebalancing tasks can have made c lower, which in turn implies that there can be a balance conflict at the parent of c. But rebalancing the parent of c (in time $O(\log m)$) can make it only as low as the sibling of c was before. Thus we have now come to the point from which, possibly up to the root, the needed rebalancing is the same as for a single deletion. That is, we may need $O(\log n)$ height decrease operations and rotations on the way from the parent of c to the root of the whole tree.

We have:

Theorem 5. Given an AVL-tree T with n leaves and an interval $[L, R]$ containing m keys, the algorithm for bulk deleting the keys in $[L, R]$ as described

above (simple bulk deletion) will produce a new AVL-tree T' such that T' contains exactly those keys of T that are not in $[L, R]$. The algorithm has time complexity $O(\log m + \log n)$. □

Using the potential function Φ_D as defined for single deletions we show that the amortized complexity of simple bulk deletion is $O(\log m)$, where m is the size of the bulk. The potential Φ_D is defined for all trees that may appear at any intermediate stage of actual deletion or rebalancing thereafter.

Lemma 4. Assume that single deletions and simple bulk deletions are applied to an AVL-tree with n leaves. Each actual single deletion will increase the potential of the tree by at most 1. Each actual simple bulk deletion and the rebalancing up to the lowest common ancestor of the end points of the bulk will increase the potential by $O(\log m)$, where m is the size of the bulk. Each last rotation included in the rebalancing process of single deletion or simple bulk deletion will increase the potential by at most 1.

Each height decrease operation will decrease the potential of the tree by at least 1. Each rotation before the last rotation and, in the case of simple bulk deletion, above the lowest common ancestor will decrease the potential by at least 1.

Proof. For a bulk deletion, only $O(\log m)$ rotations are performed below or at c. Thus, the potential increase can altogether be $O(\log m)$. In all other respects the proof parallels the proof of Lemma 2. □

Theorem 6. Assume that k_1 single deletions and k_2 simple bulk deletions with bulk sizes $m_1, m_2, \ldots, m_{k_2}$, such that $k_1 + \Sigma_{i=1}^{k_2} m_i = n$, are applied to an AVL-tree with n keys. Then altogether $c_1 k_1 + c_2 \Sigma_{i=1}^{k_2} \log m_i$ rebalancing operations, where c_1 and c_2 are constants, will be performed in conjunction with these deletions. In other words, the amortized complexity of single deletion is constant, and the amortized complexity of simple bulk deletion is $O(\log m)$, where m is the size of the bulk.

Proof. Initially the potential of the tree is $\leq n - 1$. The upper bound of the number of last rotations is $k_1 + k_2$, and the upper bound of the number of all other rebalancing operations is the initial potential plus the total possible increase of the potential, which is $2k_1 + c_2 \Sigma_{i=1}^{k_2} \log m_i$ by Lemma 4. □

5 Conclusion

We have studied the problem of bulk insertions and deletions, i.e., inserting or deleting a large number keys at the same time. In particular, we considered the case when the underlying search structure is an AVL-tree. We studied the key question of merging a simple bulk, i.e., a "small" tree, with a "large" tree, when all keys of the bulk fall in the same leaf position in the large tree. We proved the amortized complexity $O(\log m)$ of such a bulk insertion, where m is the size of the bulk. Notice that such a result cannot be obtained by simple splitting the large tree in the right place and joining these parts with the small tree, because splitting requires time proportional to the height of the tree.

References

1. G.M.Adel'son-Vel'skii and Landis. An algorithm for the organisation of information. *Dokl. Akad. Nauk SSSR* **146** (1962), 263–266 (in Russian); English Translation in *Soviet. Math.* **3**, 1259–1262.
2. L.Arge, K.H.Hinrichs, J.Vahrenhold, and J.S.Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica* **33** (2002), 104–128.
3. A.Gärtner, A.Kemper, D.Kossmann, B.Zeller. Efficient bulk deletes in relational databases. In: *Proceedings of the 17th International Conference on Data Engineering*. IEEE Computer Society, 2001, pp. 183–192.
4. S.Hanke and E.Soisalon-Soininen. Group updates for red-black trees. In: *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, Lecture Notes in Computer Science 1767. Springer-Verlag, 2000, pp. 253–262.
5. S.Huddleston and K.Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica* **17** (1982), 157–184.
6. C.Jermaine, A.Datta, and E.Omiecinski. A novel index supporting high volume data warehouse insertion. In: *Proceedings of the 25th International Conference on Very Large Databases*. Morgan Kaufmann Publishers, 1999, pp. 235–246.
7. D.E.Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching, Second Edition*. Addison-Wesley, Reading, Mass., 1998.
8. T.-W.Kuo, C-H.Wei, and K.-Y.Lam. Real-time data access control on B-tree index structures. In: *Proceedings of the 15th International Conference on Data Engineering*. IEEE Computer Society, 1999, pp. 458–467.
9. K.S.Larsen. Relaxed multi-way trees with group updates. In: *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM Press, 2001, pp. 93–101.
10. L.Malmi and E.Soisalon-Soininen. Group updates for relaxed height-balanced trees. In: *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM Press, 1999, pp. 358–367.
11. K.Mehlhorn. *Data Structures and Algorithms, Vol. 1: Sorting and Searching,* Springer-Verlag, 1986.
12. K.Mehlhorn and A.Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal on Computing* **15**:1 (1986), 22–33.
13. K.Pollari-Malmi, E.Soisalon-Soininen, and T.Ylönen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering* **8** (1996), 975–984.
14. K.Pollari-Malmi, J.Ruuth, and E.Soisalon-Soininen. Concurrency control in B-trees with differential indices. In: *Proceedings of the International Database Engineering and Applications Symposium*. IEEE Computer Society, 2000, pp. 287–295.
15. R.E.Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* **6** (1985), 306–318.
16. A.K.Tsakalidis. Rebalancing operations for deletions in AVL-trees. *RAIRO Inform. Theorique* **19**:4 (1985), 323–329.