Lecture Notes in Computer Science 1619
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Michael T. Goodrich
Catherine C. McGeoch (Eds.)

Algorithm Engineering and Experimentation

International Workshop ALENEX'99
Baltimore, MD, USA, January 15-16, 1999
Selected Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Michael T. Goodrich
John Hopkins University, Department of Computer Science
Baltimore, MD 21218, USA
E-mail: goodrich@cs.jhu.edu

Catherine C. McGeoch
Amherst University, Department of Mathematics and Computer Science
Amherst, MA 01002, USA
E-mail: ccm@cs.amherst.edu

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Algorithmic engineering and experimentation : international workshop ; selected papers / ALENEX '99, Baltimore, MD, USA, January 15 - 16, 1999. Michael T. Goodrich ; Catherine C. McGeoch (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1619)
ISBN 3-540-66227-8

CR Subject Classification (1998): F.2, E.1, I.3.5, G.2

ISSN 0302-9743
ISBN 3-540-66227-8 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN: 10705246 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

The papers in this volume were presented at the 1999 Workshop on Algorithm Engineering and Experimentation (ALENEX99). The workshop took place on January 15–16, 1999, at the Omni Hotel, Baltimore, Maryland. We gratefully acknowledge the support provided by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), located at Rutgers University. Additional support was provided by the Society for Industrial and Applied Mathematics (SIAM) and the ACM Special Interest Group on Algorithms and Computation Theory.
Forty-two papers were submitted in response to the call for papers. Twenty were selected for presentation at the workshop. In addition, there were two invited presentations. The workshop program also included six papers presented as part of the Sixth DIMACS Challenge on search in high-dimensional spaces. Those six papers will be published by the American Mathematical Society as part of the DIMACS Series in Discrete Mathematics and Theoretical Computer Science.

March 1999

Michael Goodrich
Catherine C. McGeoch

Program Committee

Richard Anderson, University of Washington
Roberto Battiti, University of Trento, Italy
Michael Goodrich, Johns Hopkins University
Guiseppe F. Italiano, Università Ca Foscari di Venezia, Italy
David S. Johnson, AT&T Labs
David R. Karger, Massachusetts Institute of Technology
Catherine C. McGeoch, Amherst College
Steven Skiena, SUNY Stony Brook
Roberto Tamassia, Brown University

Table of Contents

Efficient Implementation of the WARM-UP Algorithm for the Construction of Length-Restricted Prefix Codes

Ruy Luiz Milidiú, Artur Alves Pessoa, and Eduardo Sany Laber

Pontifícia Universidade Católica do Rio de Janeiro, Brazil
{milidiu,artur,laber}@inf.puc-rio.br

Abstract. Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ with a corresponding list of positive weights $\{w_1, \ldots, w_n\}$ and a length restriction L, the length-restricted prefix code problem is to find, a prefix code that minimizes $\sum_{i=1}^{n} w_i l_i$, where l_i, the length of the codeword assigned to a_i, cannot be greater than L, for $i = 1, \ldots, n$. In this paper, we present an efficient implementation of the WARM-UP algorithm, an approximative method for this problem. The worst-case time complexity of WARM-UP is $O(n \log n + n \log w_n)$, where w_n is the greatest weight. However, some experiments with a previous implementation of WARM-UP show that it runs in linear time for several practical cases, if the input weights are already sorted. In addition, it often produces optimal codes. The proposed implementation combines two new enhancements to reduce the space usage of WARM-UP and to improve its execution time. As a result, it is about ten times faster than the previous implementation of WARM-UP and overcomes the LRR Package Method, the faster known exact method.

1 Introduction

An important problem in the field of Coding and Information Theory is the Binary Prefix Code Problem. Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ and a corresponding list of positive weights $\{w_1, \ldots, w_n\}$, the problem is to find a prefix code for Σ that minimizes the *weighted length* of a code string, defined to be $\sum_{i=1}^{n} w_i l_i$, where l_i is the length of the codeword assigned to a_i. If the list of weights is already sorted, then this problem can be solved with $O(n)$ complexity for both time and space by one of the efficient implementations of Huffman's Algorithm [7,25]. Katajainen and Moffat presented an implementation of this algorithm that requires $O(1)$ space and the same $O(n)$ time [17]. Let us refer to optimal prefix codes as Huffman codes throughout this paper.

In this paper, we consider the length-restricted variation of this problem, that is, for a fixed $L \geq \lceil \log_2 n \rceil$, we must minimize $\sum_{i=1}^{n} w_i l_i$ constrained to $l_i \leq L$ for $i = 1, \ldots, n$. We also assume that the weights $w_1, \ldots, w_n$ are sorted, with $w_1 \leq \cdots \leq w_n$. It is worth to mention that Zobel and Moffat [26] showed that fast and space-economical algorithms for calculating length-restricted prefix codes can be very useful if a word-based model is used to compress very large text collections. In this case, each distinct word in the source text is treated as one symbol, what often produces very large alphabets. The authors realized that, if the longest codeword generated does not fit into a machine word (usually 32 bits long), then the decoding task slows down. In addition, since the decoding table size is $O(\max_i\{l_i\})$ for most of the efficient decoding methods [15], it could be interesting to restrict the maximum codeword length. Moreover, Milidiú and Laber have proved that the compression loss introduced by the length restriction exponentially decreases as we increase the value of $L - \lceil \log_2 n \rceil$ [11].

Some methods to construct length-restricted Huffman codes can be found in the literature [9,4,1,19,10,11]. The first practical method, called the Package-Merge algorithm, is due to Larmore and Hirschberg [9]. Package-Merge runs in $O(nL)$ time and requires linear space. Currently, the fastest strongly polynomial time algorithm for the problem is due to Schieber [19]. This algorithm runs in $O(n2^{o(\sqrt{\log L \log\log n})})$ time and requires $O(n)$ space.

Experimental results and full implementations for length-restricted code algorithms were reported in [4,22,24,12]. Recently, Turpin and Moffat worked on efficient implementations for length-restricted codes. They realized that the constants involved in the Package-Merge algorithm were prohibitive when dealing with very large alphabets. Hence, they developed some practical implementations for Package-Merge [8,16,22]. The best product of that effort was the LRR Package-Merge (LRR-PM) [24], an hybrid method that combines the advantages of the other implementations. This method was able to generate length-restricted Huffman codes almost as fast as an ordinary generation of unrestricted Huffman codes, requiring a reduced amount of extra memory.

In this paper, we describe an efficient implementation for the WARM-UP algorithm [10,12]. WARM-UP is an approximation algorithm that obtains length-restricted prefix codes. This algorithm is based on the construction of Huffman trees [10]. In order to construct the Huffman trees required by WARM-UP, a lazy technique [13,14] is used. This technique allows one to construct a Huffman tree without overwriting the input list of weights, spending $O(n)$ time if the list of weights is already sorted and using $O(\min(n, H^2))$ additional memory, where H is the height of the produced Huffman tree. In order to improved the execution time of these Huffman tree constructions, we combine this lazy technique with one of the techniques used by LRR-PM. This technique [16] takes advantage of the fact that the number r of distinct weights is usually much smaller than n.

The performance of our final implementation is better than that obtained by LRR-PM since it runs faster and requires a smaller amount of memory. In addition, although the WARM-UP algorithm is approximative, it generated optimal codes in 97% of our experiments.

This paper is organized as follows. In section 2, we describe the new implementation of the WARM-UP algorithm. In section 3, we present experimental results. Finally, in section 4, we summarize our major conclusions.

2 WARM-UP Implementation

The problem of finding a Huffman code for a given list of weights $w_1, \ldots, w_n$ is equivalent to finding a binary tree with minimum weighted external path length $\sum_{i=1}^{n} w_i l_i$ among the set of all binary trees with n leaves. Here, the values l_i for $i = 1, \ldots, n$ are the levels of the leaves of the tree. Huffman's algorithm [7] constructs such trees. The trees constructed by Huffman's algorithm are usually called Huffman trees. Similarly, the problem of finding a Huffman code with restricted maximal length L is equivalent to finding a tree that minimizes the weighted external path length among all binary trees with n leaves and height not greater than L.

Throughout this section, we describe a new implementation of the WARM-UP algorithm. This algorithm constructs a sequence of Huffman trees to obtain a quasi-optimal length-restricted prefix code.

The implementation is very fast in practice and it requires a reduced amount of memory as shown by the results reported in section 3. This happens since this method uses an elaborated construction of Huffman trees [13]. It also implements the runlength approach proposed by Moffat and Turpin [16]. Due to this combination of approachs, we call it the Runlength LazyHuff algorithm. We observe that, although fast, the overall implementation is not simple.

2.1 Efficient Construction of Huffman Trees

For an input list of n sorted symbol weights, Huffman's well-known greedy algorithm generates a list of codewords in $O(n)$ time and $O(n)$ extra space [7,25]. Moffat and Katajainen [17] proposed an in-place algorithm for constructing Huffman codes. Nevertheless, this last method overwrites the list of weights. This feature is not convenient for our purposes, since the WARM-UP algorithm uses information about the list of weights during its execution.

Recently, Milidiú, Pessoa and Laber developed the LazyHuff algorithm [13,14]. LazyHuff obtains the number of leaves at each level of the Huffman tree. With this information, an optimal Huffman-Shannon-Fano prefix code can be easily constructed [2]. This algorithm spends $O(n)$ time if the list of weights is already sorted and requires $O(\min(n, H^2))$ extra space, where H is the height of the resulting Huffman tree. Furthermore, this construction does not overwrite the input list of weights. Hence, it is more adequate for the WARM-UP algorithm.

First, let us briefly describe Huffman's algorithm. It starts with a list S containing the n leaves. In the general step, the algorithm selects the two nodes with smallest weights in the current list of nodes S and removes them from the list. Next, the removed nodes become children of a new internal node, that is inserted in S. To this internal node it is assigned a weight that is equal to the sum of the weights of its children. The general step repeats until there is only one node in S, the root of the Huffman tree. Observe that, in a Huffman tree, to each internal node is also assigned a weight. The optimallity of the Huffman tree is assured by recursively applying the following well-known proposition [3].

Proposition 1. *Let us consider a list of leaves $[x_1, \ldots, x_n]$, with the corresponding weights satisfying $w(x_i) \leq w(x_{i+1})$, for $i = 1, \ldots, n-1$. There is a binary tree that minimizes $\sum_{i=1}^{n} w(x_i)\ell(x_i)$ among all binary trees with leaves $x_1, \ldots, x_n$, where x_1 and x_2 are siblings.*

The LazyHuff algorithm is based on a generalization of proposition 1, that we present bellow.

Proposition 2. *Let us consider a list of nodes $[x_1, \ldots, x_n]$, with the corresponding weights satisfying $w(x_i) \leq w(x_{i+1})$, for $i = 1, \ldots, n-1$. If k is a positive integer such that $w(x_{2k}) \leq w(x_1) + w(x_2)$, then there is a binary tree that minimizes $\sum_{i=1}^{n} w(x_i)\ell(x_i)$ among all binary trees with leaves $x_1, \ldots, x_n$, where x_{2i-1} and x_{2i} are siblings, for $i = 1, \ldots, k$.*

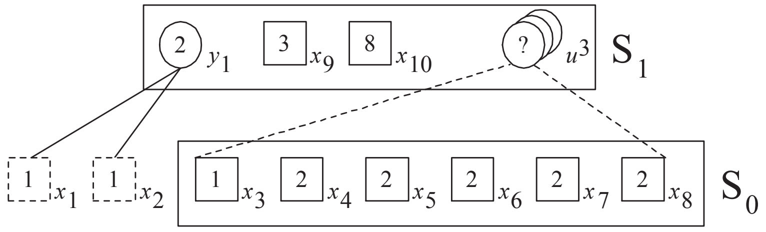


Fig. 1. The lists S_0 and S_1 after the first steps of the LazyHuff algorithm for $W = [1, 1, 1, 2, 2, 2, 2, 2, 3, 8]$

Now, let us describe the first steps performed by LazyHuff for $W = [1, 1, 1, 2, 2, 2, 2, 2, 3, 8]$. Let S_0 be a list of nodes sorted by their weights in a non-decreasing order. Similarly to Huffman's algorithm, we start with $S_0 = [x_1, \ldots, x_{10}]$, where x_i is the leaf with the i-th weight, for $i = 1, \ldots, 10$. Observe that, by proposition 1, there is an optimal binary tree where x_1 and x_2 are siblings. Therefore, these nodes are removed from S_0 and become the children of a new internal node y_1, that is inserted in S_0. The weight of y_1 is given by $w(x_1) + w(x_2) = w_1 + w_2 = 2$. After that, we have $S_0 = [x_3, \ldots, x_8, y_1, x_9, x_{10}]$, since $w(x_8) \leq w(y_1) < w(x_9)$.

Observe that, by proposition 2, there is an optimal binary tree where x_{2i-1} and x_{2i} are siblings, for $i = 1, 2, 3, 4$, that is, these nodes could be replaced by their corresponding parents in S_0. Instead, the LazyHuff algorithm uses a lazy or demand-driven approach. A new list of nodes S_1 is created and all the nodes of S_0 are moved to S_1, except for $x_3, \ldots, x_8$. The new list of nodes also contains three *unknown* internal nodes, that correspond to the parents of $x_3, \ldots, x_8$. Hence, we have $S_0 = [x_3, \ldots, x_8]$ and $S_1 = [y_1, x_9, x_{10}, u^3]$, where u^3 denotes the three unknown internal nodes. Both lists are represented in figure 1. The triple circle with a question mark on top denotes the three unknown internal nodes in S_1. Since the children of y_1 were deleted from the lists, they are represented with dashed lines. These are the steps performed during the LazyHuff first iteration. The following iterations are similar. A new list is always created by applying propositions 1 and 2. In addition, whenever the weight of an unknown internal node from S_i is needed, it is calculated by recursively combining two nodes of S_{i-1}. Moreover, each list can be stored using just a constant amount of memory. Since exactly H lists are created, the additional space required for these lists is $O(H)$.

In order to calculate the number of leaves at each level, the algorithm also maintains a list of integers associated to each internal node. The list associated to a node s stores the number of internal nodes at each level in the subtree rooted by s. When an unknown internal node is revealed, its associated list is obtained by combining the informations stored in the lists of its children. After all iterations are performed, the algorithm uses the list of integers associated to the root to calculate the number of leaves at each level in the tree. These lists are responsible for the $O(\min(n, H^2))$ additional space required by LazyHuff.

2.2 Runlength LazyHuff

In large alphabets, the number r of distinct weights is usually much smaller than n. Moffat and Turpin [16] use this fact to speed up the construction of Huffman codes. As an example, the word-based model used to compress the three-gigabytes TREC collection [6] generates an alphabet with 1,073,971 symbols ($n = 1,073,971$) and with only 11,777 distinct weights ($r = 11,777$). Using a method that explores a runlength representation of the list of weights, the lengths of a Huffman encoding for the TREC collection were produced in about 0.5 seconds, while the Katajainen's in-place method generates these same lengths in 1.4 seconds [16].

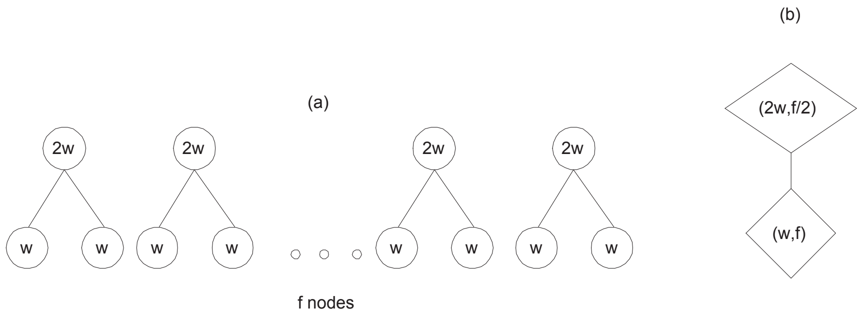


Fig. 2. (a) Melding operation in a traditional construction of Huffman trees. (b) Melding operation in a runlength construction of Huffman trees.

The runlength format of the input list of weights is a set of pairs $\{(f_i, w_{k_i}) | 1 \leq i \leq r\}$, with $w_{k_i} < w_{k_{i+1}}$, where f_i represents the number of alphabet symbols with weight equal to w_{k_i}. The runlength algorithm due to Moffat and Turpin converts the input list of weights into a runlength format. Next, the algorithm follows the same execution of Huffman's algorithm. Nevertheless, it stores all the nodes with the same weight w at an unique special node. This special node also stores a pair (w, f), where f is the node frequency and represents the number of nodes with weight w. Hence, the operation of melding nodes with the same weight is optimized. That happens because the special node that represents the nodes with these weights is replaced by a new special node, what saves several operations. The new node stores the pair $(2w, \lfloor f/2 \rfloor)$. Figure 2.(a) illustrates the melding operation for equal weights in a traditional Huffman algorithm and figure 2.(b) illustrates the same operation in a runlength algorithm. Observe that f is even in figure 2. Let (w', f') be the pair stored at the next available special node. If f is odd, then an additional special node is created with the pair $(w + w', 1)$ and f' is decreased by one. The runlength Huffman's algorithm runs in $O(r \log(n/r))$ time and requires an additional space with the same complexity.

The current implementation of LazyHuff combines the method described in the previous subsection with the runlength method. Nevertheless, it must be observed that the conversion to the runlength format is not performed at the beginning of the execution as in the ordinary runlength method. Runlength LazyHuff obtains each special node as soon as it is required to form a new internal node. As a result, the additional space required by this method is $O(\min(n, H^2))$.

Unfortunately, due to the cost of combining two lists of counters whenever two internal nodes are melded, the time complexity of Runlength LazyHuff is not $O(r \log(n/r))$. Observe that each list of integers can be as long as the height H of the current tree. However, if Runlength LazyHuff is called by WARM-UP, then its execution can be aborted as soon as H overcomes the length-restriction L. In this case, the method runs in $O(\min\{n, Lr \log(n/r)\})$ time and requires $O(L^2)$ additional space.

2.3 The WARM-UP Algorithm

The WARM-UP algorithm proposed by Milidiú and Laber [10] introduces a novel approach to the construction of length-restricted prefix codes. It is based on some fundamental properties of lagrangean relaxation, a well-known technique of widespread use in the solution of combinatorial optimization problems.

Preliminary results with the WARM-UP algorithm were reported in [12]. In that work, WARM-UP was compared against the ROT heuristic [4] and the LazyPM algorithm [8]. WARM-UP was the fastest one and produced an optimal code in all but two cases out of sixty eight. Nevertheless, it spent more memory than LazyPM. In that implementation of WARM-UP, Katajainen's in-place algorithm [17] was used to generate the Huffman trees. The new implementation of WARM-UP introduces the use of the techniques presented in subsections 2.1 and 2.2 for the construction of Huffman trees, being significantly faster and requiring much less additional memory.

Now, we briefly describe the WARM-UP algorithm. Let W denote a list of positive integer weights $\{w_1, \ldots, w_n\}$. For a given real value x, let us define the associated list of weights W^x by $W^x = \{max\{w_1, x\}, \ldots, max\{w_n, x\}\}$. We use T_x to denote any Huffman tree for W^x, and $h(T)$ to denote the height of a tree T. We also use T_x^- to denote a minimum height Huffman tree [20] for W^x. If the original weight of a given leaf is smaller than x, then we say that this leaf is *warmed up* to the value of x in a tree T_x.

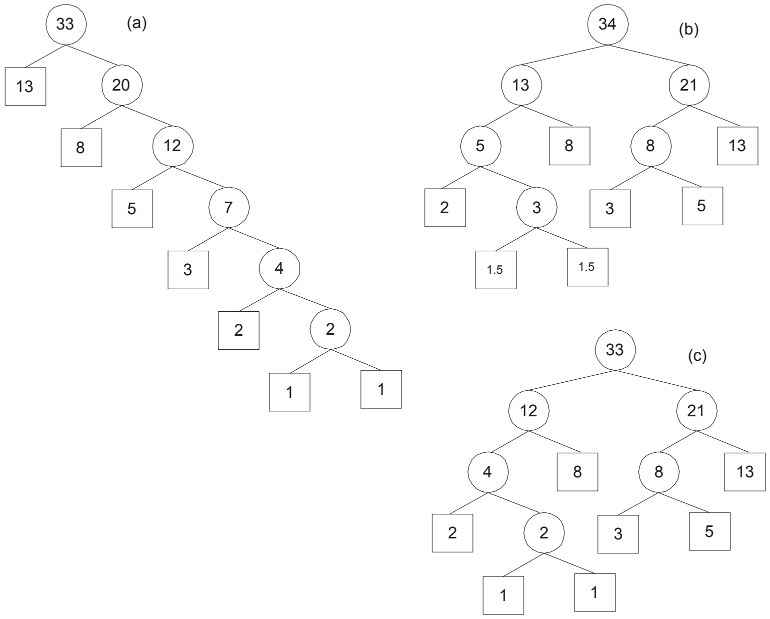


Fig. 3. (a) A Huffman tree for the set of weights $W = \{1, 1, 2, 3, 5, 8, 13\}$ (b) A Tree T_x for the set of weights $W^{1.5}$ (c) An optimal tree for the set of $W = \{1, 1, 2, 3, 5, 8, 13\}$, and height restricted to 4.

Figure 3.(a) shows a Huffman tree for $W = \{1, 1, 2, 3, 5, 8, 13\}$, whereas figure 3.(b) illustrates a $T_{1.5}$ tree for the corresponding set of weights $W^{1.5} = \{1.5, 1.5, 2, 3, 5, 8, 13\}$. Observe that the tree in figure 3.(a) has height equal to 6, and the one in figure 3.(b) has height equal to 4.

The WARM-UP algorithm is strongly based on two results. The first one states that if all the *warmed up* leaves are at level L in T_x, then T_x is an optimal code tree with restricted maximal length L. The second one states that if $x > y$, then $h(T_x) \leq h(T_y)$. Based on these results, the WARM-UP algorithm performs a binary search on the interval (w_1, w_n) looking for the smallest integer x such that $h(T_x) \leq L$. It also stops if a tree T_x is assured to be optimal, that is, when all the *warmed up* leaves are at level L. In figure 4, we present a pseudo-code for the new WARM-UP implementation.

```
Phase 0: Trying a Huffman Tree
    If $h(T^-_{w_1}) \leq L$ then Return($T^-_{w_1}$)

Phase 1: Binary searching the warm-up value
    $inf \leftarrow w_1$;  $sup \leftarrow w_n$
    $x \leftarrow$ GetFirstGuess($L, w_1, \ldots, w_n$)
    Repeat
        $h^- \leftarrow h(T^-_x)$
        If $T^-_x$ Is Optimal Then Return($T^-_x$)
        If $h^- > L$ Then
            If $(sup - x) = 1$ Then Return($T^-_{sup}$)
            $inf \leftarrow x$
        Else
            If $(x - inf) = 1$ Then Return($T^-_x$)
            $sup \leftarrow x$
        End If
        $x \leftarrow \lfloor (inf + sup)/2 \rfloor$
    End Repeat
```

Fig. 4. A pseudo-code for the new WARM-UP implementation

The procedure GetFirstGuess [12] is a heuristic that chooses an initial value for x. This heuristic is based on the following fact: if we relax the integrality constraint on the codeword lengths, then the i-th codeword length of an optimal code would be given by $-\log_2 p_i$, where p_i is defined as $w_i / \sum_{j=1}^{n} w_j$. Hence, we choose x such that $-\log_2(x / \sum_{j=1}^{n} max\{w_j, x\})$ rounds to L.

In the current implementation, the Huffman trees constructed during the algorithm execution are obtained through the Runlength LazyHuff. In addition, the original implementation of WARM-UP constructs, for each tested value of x, both the minimum and the maximum height Huffman trees. Observe in the pseudo-code of figure 4 that only minimum height Huffman trees are constructed. In our experiments, this simplification does not affect the quality of the resulting code. Finally, it is immediate that this implementation runs in $O(\log w_n \min\{n, Lr\log(n/r)\})$ in the worst case. However, it was much faster in our experiments.

3 Experiments

In this section, we report some experiments concerning two methods to calculate length-restricted prefix codes. The first method is the new implementation of the WARM-UP algorithm, described in section 2. The second method, called the LRR Package-Merge algorithm, is due to Turpin and Moffat [24,23]. It is an hybrid method that combines the advantages of three implementations of the Package-Merge algorithm [9].

3.1 Enviroment

Both algorithms were implemented in C. The source code of LRR-PM was obtained from Turpin [1]. The source code of the current WARM-UP implementation is available by anonymous ftp at the following URL.

```
ftp://ftp.learn.fplf.org.br/pub/codigos_fonte/warm-up.tar.gz
```

The machine used for the experiments has the following characteristics:

Type:	RISC 6000 Station
Model:	42T PowerPC
Manufacturer:	IBM
Internal Memory:	128 Mbytes
Operating System:	AIX version 4.1.4
Clock Rate:	100 ticks per second

Both programs were compiled by gcc [21] at this machine, with the same optimization options. For each algorithm, each input list of weights and each length-restriction, we measured both the average cpu time and the additional amount of memory required by one execution. Each execution time was derived from the total number of machine *clock ticks* during ten repetitions. Therefore, each reported measure may have an error not greater than one thousandth of a second. For the smallest values, one hundred repetitions were used to achieve a better precision.

We also instrumented the code to measure the amount of additional memory used by each algorithm. Each reported value represents the maximum amount of memory simultaneously allocated during an algorithm execution. The memory used by the call stack was not counted. We also ignored the memory required to store the input list of weights.

3.2 Input Data

In table 1, each line corresponds to one of the eight lists of weights used in our experiments, where n is the number of weights in the list, r is its number of distinct weights and p_n is the probability of the most probable symbol. The value of p_n is given by $w_n / \sum_{i=1}^{n} w_i$, where w_n is the greatest weight in the corresponding list.

The construction of the first list of weights is based on Zipf's law [4], a theoretical model of word probabilities in a natural language text. This law states that the i-th most probable word is i times less probable than the most probable one. In order to generate the list of integer weights, we assume that the value of w_i is given by $\lfloor \frac{w_n}{n-i+1} + 0.5 \rfloor$, for $i = 1, \ldots, n-1$, where n is the number of words and $w_n = 3,000,000$. The second list of weights represents an incremental distribution with $60,000$ weights, where the value of the i-th weight

[1] The same source code has been used in the experiments reported in [24]

Table 1. Lists of weights

Source	Id.	n	r	p_n
Zipf's law	ZL	4,000,000	3,463	0.06334
Incremental	IN	60,000	60,000	0.00003
Reuters (words)	RW	92,353	1,210	0.02639
Reuters (non-words)	RN	1,601	194	0.51684
Reuters (spaceless)	RS	93,953	1,273	0.03171
Gutenberg (words)	GW	753,549	3,669	0.04650
Gutenberg (non-words)	GN	27,340	927	0.66194
Gutenberg (spaceless)	GS	780,888	3,902	0.04424

is equal to i. This list is interesting since all its weights are distinct, what is a "worst-case" for the method described in section 2.2.

The other six lists were extracted from two text collections, referred as Reuters and Gutenberg in table 1. For these collections, we used two parsing methods. The first one is the word model adopted by the Huffword compression algorithm [26] and generates two lists of weights. It packs any sequence of alphanumeric ASCII codes into a single word symbol that corresponds to one of the weights in the first list. Moreover, each sequence of non-alphanumeric ASCII codes is considered a non-word symbol, and corresponds to one of the weights in the second list. For example, if the string

"three, fourteen and three again"

is submitted to the parsing method, then "fourteen", "and", "again" and "three" are considered word symbols, corresponding respectively to the weights 1, 1, 1 and 2. In this case, the non-word symbols are ", " and " ", corresponding respectively to the weights 1 and 4.

The second parsing method is a variation of the first one. It was proposed in [18] and generates a single list of weights. This method ignores all occurrences of a single space between two word symbols. During the decoding process, the original message is obtained by inserting a single space between each pair of consecutive word symbols. Since this process must distinguish word symbols from non-word ones, a single prefix code must be used for both words and non-words. The resulting symbols are called *spaceless words*. For the sentence of the previous example, the word symbols "fourteen", "and", "again" and "three" and the non-word symbol ", " are obtained, corresponding respectively to the weights 1, 1, 1, 2 and 1.

In table 1, Reuters is the *Distribution 1.0 of the Reuters-21578* text categorization test collection. This collection, that is freely available to researchers from David D. Lewis' professional home page, has 21578 documents in SGML format distributed in 22 files. The URL of Lewis' home page is the following.

`http://www.research.att.com/~lewis`

Each of the first 21 files contains 1000 documents, while the last one contains 578 documents. Furthermore, Gutenberg is a collection of text files retrieved from the Gutenberg Project. The URL of the Gutenberg Project is the following.

`ftp://sailor.gutenberg.org/pub/gutenberg`

For our purposes, all 637 files with the extension ".txt" were used.

3.3 Execution Time

The execution times (in seconds) of both WARM-UP (W-UP) and LRR-PM are shown in table 2. Their relative performance (the execution times of LRR-PM divided by that of WARM-UP) is also presented in table 3. The measurements that correspond to the Reuters non-word symbols are not reported because their values are too small. Observe that WARM-UP was faster than LRR-PM except for one entry (printed bold) out of forty four!

Now, let us explain some important aspects that distinguish the performance of the WARM-UP algorithm. Although WARM-UP is an approximation algorithm, we observed that optimal codes were generated in all but one case out of forty four: the Gutenberg non-word symbols, with $L = 16$. In seven cases out of forty four, only two Huffman trees were constructed: the unrestricted Huffman tree and the Huffman tree that corresponds to the initial value assigned to x. In all these cases, the value of x calculated by the procedure GetFirstGuess produced an optimal length-restricted prefix code. This fact explains the fast execution time of WARM-UP.

We observe that all the cases where WARM-UP has constructed more than two Huffman trees also correspond to the Gutenberg non-word symbols. In this case, the number of constructions performed for each value of L is shown in the table bellow.

L	Number of constructions
16	18
17-18	2
19-24	3

Recall that both the WARM-UP optimality criterion and the initial value assigned to x are based on the relaxation of the codeword length integrality. This fact suggests that the optimal solutions that correspond to the Gutenberg non-word symbols are far away from their corresponding relaxed solutions. The redundancy of a Huffman code is defined as the difference between its average length and the entropy, the average length of an optimal code with relaxed codeword lengths. Moreover, Gallager [5] proved that the redundancy of Huffman

Table 2. Execution times (seconds)

	RW		RS		ZL		IN	
L	W-UP	LRR-PM	W-UP	LRR-PM	W-UP	LRR-PM	W-UP	LRR-PM
17							0.1671	0.3028
18	0.0209	0.0413	0.0213	0.0387			0.2045	0.5639
19	0.0256	0.0558	0.0258	0.0532			0.2250	0.8479
20	0.0332	0.0701	0.0305	0.0657			0.2418	1.1723
21	0.0346	0.0831	0.0361	0.0793			0.2486	1.4723
22			0.0370	0.0942			0.2530	1.8188
23					0.0884	0.2342	0.2635	2.1103
24					0.1085	0.2946	0.2560	2.4427
25					0.1082	0.3237	0.2683	2.7591
26							0.2882	3.0969
27							0.3427	3.4361
28							0.4440	3.7732
29							0.4877	4.2189

	GW		GN		GS	
L	W-UP	LRR-PM	W-UP	LRR-PM	W-UP	LRR-PM
16			**0.0246**	0.0068		
17			0.0061	0.0132		
18			0.0089	0.0155		
19			0.0190	0.0209		
20			0.0238	0.0297		
21	0.0617	0.1800	0.0276	0.0367	0.0640	0.1759
22	0.0762	0.2278	0.0314	0.0439	0.0752	0.2295
23	0.0929	0.2739	0.0305	0.0555	0.0893	0.2750
24	0.1110	0.3306	0.0298	0.0621	0.1154	0.3193
25	0.1115	0.3581			0.1162	0.3588

Table 3. Relative performance

L	RW	RS	ZL	IN	GW	GN	GS
16						**0.3**	
17				1.8		2.2	
18	2.0	1.8		2.8		1.7	
19	2.2	2.1		3.8		1.1	
20	2.1	2.2		4.8		1.2	
21	2.4	2.2		5.9	2.9	1.3	2.7
22		2.5		7.2	3.0	1.4	3.1
23			2.6	8.0	2.9	1.8	3.1
24			2.7	9.5	3.0	2.1	2.8
25			3.0	10.3	3.2		3.1
26				10.7			
27				10.0			
28				8.5			
29				8.7			

codes is directly related to the probability p_n of the most probable symbol, that is, the greater is the value of p_n, the more redundant can the corresponding Huffman code be. In table 1, observe that the Gutenberg non-word symbols have the greatest value of $p_n = 0.66194$, what confirms our suggestion. Unfortunately, we still do not have a precise theoretical formulation for these observations. However, the only non-optimal code obtained by WARM-UP had an average length only about 6% greater than the optimal one. Although it required eighteen Huffman tree constructions, the execution time of WARM-UP was not too large in this case.

If we do not consider these degenerated cases, then WARM-UP runs in $O(\min\{n, Lr\log(n/r)\})$ time, which is the same complexity as the Runlength LazyHuff. On the other hand, LRR-PM runs in $O(\min\{(L-\log n)n, Lr\log(n/r)\}$ in all cases [24]. Hence, if $r = n$ then WARM-UP runs in $O(n)$ time and LRR-PM runs in $O((L - \log n)n)$. This fact explains the good relative performance of WARM-UP for the Incremental distribution (see table 3). In this case, for $L > 26$, the relative performance of WARM-UP deteriorates due to the first Huffman tree construction. Since the length of the longest Huffman codeword is equal to thirty, the WARM-UP algorithm almost completly constructs the unrestricted Huffman tree before the length-restriction is violated.

For the other lists of weights, the value of r is much smaller than n. In these cases, although the complexity of both algorithms are similar, WARM-UP has two practical advantages. The first one is observed when the length of the longest Huffman codeword is not too close to L (say greater than $L+2$). If that happens, then either the value assigned to x is very large, what decreases the number of distinct weights, or just a small part of the Huffman tree is constructed before the length-restriction is violated. Hence, the performance of WARM-UP is improved. The second advantage is observed in the Runlength LazyHuff algorithm. The L factor observed in the expression of its time complexity is only due to the manipulation of some lists of counters, that is very simple and fast. All the other operations performed by Runlength LazyHuff spend only $O(r\log(n/r))$ time. Hence, we believe that the LRR-PM algorithm is slower because it is based on the $O(Ln)$-time Package Merge algorithm.

3.4 Additional Memory Usage

The total additional memory (in bytes) required by both methods on each execution is reported in table 4. The measurements that correspond to the Reuters non-word symbols are not reported because their values are negligible. Table 4 also shows both the amount of memory required for the input list of weights and the additional memory used by LRR-PM to store this input list after it was converted to the runlength format. Although, WARM-UP also manipulates the weights in the runlength format, the converted list is not stored. Instead, the conversion is done on-line, as each Huffman tree is constructed. Since each construction requires only a single pass through the input list of weights, this conversion does not deteriorate the performance of WARM-UP. On the other hand, LRR-PM needs a pre-processing phase because as much as L passes through the list of weights may be required. As a result, WARM-UP requires only $O(L^2)$ additional memory, while LRR-PM requires $O(r + L^2)$ additional memory.

In table 4, we observe that WARM-UP required much less additional memory than LRR-PM in all cases. LRR-PM always requires about ten times the additional memory of WARM-UP, except for the Incremental distribution where this factor becomes about one hundred. In this case, the additional memory used by LRR-PM to store the converted list represents more than 90% of the total additional memory required. For all other lists of weights, the converted lists were stored at about 50% (or less) of the additional memory required by LRR-PM.

Observe also that, if we ignored the additional memory used to store the converted lists, then LRR-PM would still require about five times the additional memory used by WARM-UP. This fact suggests that, if the input list of weights were already given in the runlength format, then WARM-UP would still be more space-economical than LRR-PM.

4 Conclusions

In this paper, we presented a new implementation of the WARM-UP algorithm. This implementation combines the advantages of two methods for the construction of unrestricted Huffman codes. The first method, that is based on a lazy approach, allows to construct Huffman trees using a very small amount of addition memory. The second method, that is based on the runlength representation of the nodes in a Huffman tree, leads to a very fast implementation of the Huffman's algorithm.

The new implementation was compared to LRR-PM (initials of Lazy Runlength Reverse Package-Merge), the most efficient known exact method. In our experiments, the performance of this new implementation overcomes the results obtained through LRR-PM. In addition, these experiments show that the WARM-UP algorithm produces optimal codes very often, what suggests that it is the more suitable method. It is worth to mention that this observation was confirmed by experiments recently reported by Turpin [23], using the TREC collection [6]. In these experiments, the author obtained similar results.

Finally, LRR-PM [24] combines the advantages of three methods under the Package-Merge paradigm, two of which are similar to that used by our new implementation [8,16]. This fact suggests that these methods can be used as general purpose techniques to improve both the execution time and the space usage of known algorithms for similar problems.

References

1. Aggarwal, Schieber, and Tokuyama. Finding a minimum-weight k-link path in graphs with the concave monge property and applications. *GEOMETRY: Discrete & Computational Geometry*, 12, 1994.
2. J. Brian Connell. A Huffman-Shannon-Fano code. In *Proceedings of the IEEE*, volume 61, pages 1046–1047, July 1973.
3. Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.
4. A. S. Fraenkel and S. T. Klein. Bounding the depth of search trees. *The Computer Journal*, 36(7):668–678, 1993.
5. R.G. Gallager. Variations on a theme of huffman. *IEEE Transaction on Information Theory*, IT-24(6):668–674, November 1978.
6. D. Harman. Overview of the second text retrieval conference (trec-2). *Information Processing Management*, 31(3):271–289, 1995.
7. D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, September 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.
8. Katajainen, Moffat, and Turpin. A fast and space-economical algorithm for length-limited coding. In *ISAAC: 6th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE))*, 1995.
9. Lawrence L. Larmore and Daniel S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473, July 1990.
10. Ruy L. Milidiú and Eduardo S. Laber. Warm-up algorithm: A lagrangean construction of length restricted huffman codes. Technical Report 15, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brasil, January 1996.
11. Ruy L. Milidiú and Eduardo S. Laber. Improved bounds on the inefficiency of length restricted codes. Technical Report 33, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brasil, January 1997.
12. Ruy L. Milidiú, Artur A. Pessoa, and Eduardo S. Laber. Practical use of the warm-up algorithm on length-restricted codes. In Ricardo Baeza-Yates, editor, *Proceedings of the Fourth Latin American Workshop on String Processing*, volume 8 of *International Informatics Series*, pages 80–94, Valparaiso, Chile, November 1997. Carleton University Press.
13. Ruy L. Milidiú, Artur A. Pessoa, and Eduardo S. Laber. A space-economical algorithm for minimum-redundancy coding. Technical Report 02, Departamento de Informática, PUC-RJ, Rio de Janeiro, Brasil, January 1998.
14. Ruy L. Milidiú, Artur A. Pessoa, and Eduardo S. Laber. Two space-economical algorithms for calculating minimum redundancy prefix codes. In *Proceedings of the DCC (to appear)*, March 1999.
15. A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, October 1995.
16. A. Moffat and A. Turpin. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Transactions on Information Theory*, 44(4), July 1998.
17. Alistair Moffat and Jyrki Katajainen. In-place calculation of minimum-redundancy codes. In Selim G. Akl, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, 4th International Workshop*, volume 955 of *Lecture Notes in Computer Science*, pages 393–402, Kingston, Ontario, Canada, 16–18 August 1995. Springer.

18. Edleno S. Moura, Gonzalo Navarro, and Nivio Ziviani. Indexing compressed text. In Ricardo Baeza-Yates, editor, *Proceedings of the Fourth Latin American Workshop on String Processing*, volume 8 of *International Informatics Series*, pages 95–111, Valparaiso, Chile, November 1997. Carleton University Press.
19. Baruch Schieber. Computing a minimum-weight k-link path in graphs with the concave Monge property. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 405–411, San Francisco, California, 22–24 January 1995.
20. Eugene S. Schwartz. An optimum encoding with minimum longest code and total number of digits. *Information and Control*, 7(1):37–44, March 1964.
21. R. M. Stallman. *Using and Porting GNU CC*, 1995. Available at the URL `http://ocean.ucc.ie/utils/manuals/gcc/gcc.ps`.
22. A. Turpin and A. Moffat. Practical length-limited coding for large alphabets. *The Computer Journal*, 38(5):339–347, 1995.
23. Andrew Turpin. *Efficient Prefix Coding*. PhD thesis, University of Melbourne, Departament of Computer Science, July 1998.
24. Andrew Turpin and Alistair Moffat. Efficient implementation of the package-merge paradigm for generating length-limited codes. In Michael E. Houle and Peter Eades, editors, *Proceedings of Conference on Computing: The Australian Theory Symposium*, pages 187–195, Townsville, 29–30 January 1996. Australian Computer Science Communications.
25. J. van Leeuwen. On the construction of Huffman trees. In S. Michaelson and R. Milner, editors, *Third International Colloquium on Automata, Languages and Programming*, pages 382–410, University of Edinburgh, 20–23 July 1976. Edinburgh University Press.
26. Justin Zobel and Alistair Moffat. Adding compression to a full-text retrieval system. *Software—Practice and Experience*, 25(8):891–903, August 1995.

Table 4. Additional memory (bytes)

	RW		RS		ZL		IN	
L	W-UP	LRR-PM	W-UP	LRR-PM	W-UP	LRR-PM	W-UP	LRR-PM
17							2,720	497,640
18	2,424	29,364	2,512	29,868			3,040	499,684
19	2,696	31,520	2,720	32,024			3,328	501,840
20	2,912	33,788	2,928	34,292			3,608	504,108
21	2,920	36,168	3,040	36,672			3,840	506,488
22			3,112	39,164			4,080	508,980
23					3,800	59,288	4,320	511,584
24					3,976	62,004	4,440	514,300
25					3,976	64,832	4,624	517,128
26							4,736	520,068
27							4,816	523,120
28							4,992	526,284
29							5,184	529,560
$*^1$		9,680		10,184		27,704		480,000
$*^2$	369,412		375,812		16,000,000		240,000	

	GW		GN		GS	
L	W-UP	LRR-PM	W-UP	LRR-PM	W-UP	LRR-PM
16			1,472	23,124		
17			1,752	25,056		
18			2,272	27,100		
19			2,516	29,256		
20			2,892	31,524		
21	3,280	55,840	3,008	33,904	3,424	57,704
22	3,488	58,332	3,004	36,396	3,632	60,196
23	3,744	60,936	2,992	39,000	3,736	62,800
24	4,016	63,652	3,068	41,716	4,128	65,516
25	4,016	66,480			4,128	68,344
$*^1$		29,352		7,416		31,216
$*^2$	3,014,196		109,360		3,123,552	

$*^1$ The additional memory (in bytes) used to store the list of weights after it was converted to the runlength format.

$*^2$ The memory (in bytes) required for the input list of weights.

Implementing Weighted *b*-Matching Algorithms: Insights from a Computational Study

Matthias Müller–Hannemann and Alexander Schwartz

Technische Universität Berlin
Department of Mathematics
Straße des 17. Juni 136
D 10623 Berlin, Germany
mhannema@math.tu-berlin.de
http://www.math.tu-berlin.de/~mhannema/
schwartz@math.tu-berlin.de
http://www.math.tu-berlin.de/~schwartz/

Abstract. We present an experimental study of an implementation of weighted perfect *b*-matching based on Pulleyblank's algorithm (1973). Although this problem is well-understood in theory and efficient algorithms are known, only little experience with implementations is available. In this paper several algorithmic variants are compared on synthetic and application problem data of very sparse graphs. This study was motivated by the practical need for an efficient *b*-matching solver for the latter application, namely as a subroutine in our approach to a mesh refinement problem in computer-aided design (CAD).
Linear regression and operation counting is used to analyze code variants. The experiments indicate that a fractional jump-start should be used, a priority queue within the dual update helps, scaling of *b*-values is not necessary, whereas a delayed blossom shrinking heuristic significantly improves running times only for graphs with average degree two. The fastest variant of our implementation appears to be highly superior to a code by Miller & Pekny (1995).

1 Introduction

Given an undirected graph $G = (V, E)$ with edge weights c_e for each edge $e \in E$ and node capacities b_v for each node $v \in V$, the *perfect b-matching problem* is to find a maximum weight integral vector $x \in \mathbb{N}_0^{|E|}$ satisfying $\sum_{e=(v,w)} x_e = b_v$ for all $v \in V$. Weighted *b*-matching is a cornerstone problem in combinatorial optimization. Its theoretical importance is due to the fact that it generalizes both ordinary weighted matching (i.e. matching with all node capacities equal to one, 1-matching) and minimum cost flow problems. All these problems belong to a 'well-solved class of integer linear programs' [EJ70] in the sense that they all can be solved in (strongly) polynomial time. There are excellent surveys on matching theory by Gerards [Ger95] and Pulleyblank [Pul95].

To the best of our knowledge, there is no publicly available code for weighted b-matching problems. As there are many promising variants of b-matching algorithms, but not too much practical experience with them, we decided to develop a general framework which captures most of these variants. This framework enabled us to do a lot of experiments to improve the performance of our code incrementally by exchanging subalgorithms and data structures. Details of our software design are described in [MS98].

Intention. The goal of this paper is not only to determine an empirical asymptotical performance of a b-matching implementation in comparison with theoretical worst case bounds, we also tried to identify bottleneck operations on which one has to work in order to improve the practical performance. We designed a number of experiments to study algorithmic proposals made in previous work on 1-matching for which the "in-practice" behavior for b-matching is either not obvious or contradictory statements have been found in the literature. Linear regression and operation counting [AO96] are used to evaluate the experiments.

Applications. Important applications of weighted b-matching include the *T-join problem*, the *Chinese postman problem*, shortest paths in undirected graphs with negative costs (but no negative cycles), the 2-factor relaxation for the symmetric traveling salesman problem (STSP), and capacitated vehicle routing [Mil95]. For numerous other examples of applications of the special cases minimum cost flow and 1-matching we refer to the book of Ahuja, Magnanti, and Orlin [AMO93].

A somewhat surprising new application of weighted b-matching stems from quadrilateral mesh refinement in computer-aided design [MMW97,MM97]. Given a surface description of some workpiece in three-dimensional space as a collection of polygons, the task to refine the coarse input mesh into an all-quadrilateral mesh can be modeled as a weighted perfect b-matching problem (or, equivalently, as a bidirected flow problem). This class of problem instances is of particular interest because unlike the previous examples, the usually occurring node capacities b_v are quite large (in principle, not even bounded in $\mathcal{O}(|V|)$) and change widely between nodes.

Previous work. Most work on matching problems is based on the pioneering work of Edmonds [Edm65]. "Blossom I" by Edmonds, Johnson, and Lockhart [EJL69] was the first implementation for the *bidirected flow problem* (which is, as mentioned above, equivalent to the b-matching problem).

Pulleyblank [Pul73] worked out the details of a blossom-based algorithm for a mixed version of the perfect and imperfect b-matching in his Ph.D. thesis and gave a PL1 implementation, "Blossom II". His algorithm has a complexity of $\mathcal{O}(|V||E|B)$ with $B = \sum_{v \in V} b_v$, and is therefore only pseudo-polynomial.

Cunningham & Marsh [Mar79] used scaling to obtain the first polynomial bounded algorithm for b-matching. Gabow [Gab83] proposes an efficient reduction technique which avoids increasing the problem size by orders of magnitude.

Together with scaling, this approach leads to an algorithm with a bound of $\mathcal{O}(|E|^2 \log |V| \log B_{max})$ where B_{max} is the largest capacity.

Anstee [Ans87] suggested a staged algorithm. In a first stage, the fractional relaxation of the weighted perfect b-matching is solved via a transformation to a minimum cost flow problem on a bipartite graph, a so-called *Hitchcock transportation problem.* In stage two, the solution of the transportation problem is converted into an integral, but non-perfect b-matching by rounding techniques. In the final stage, Pulleyblank's algorithm is invoked with the intermediate solution from stage two. This staged approach yields a strongly-polynomial algorithm for the weighted perfect b-matching problem if a strongly polynomial minimum cost flow algorithm is invoked to obtain the optimal fractional perfect matching in the first stage. The best strongly polynomial time bound for the (uncapacitated) Hitchcock transportation problem is $\mathcal{O}((|V| \log |V|)(|E| + |V| \log |V|)$ by Orlin's excess scaling algorithm [Orl88], and the second and third stage of Anstee's algorithm require at most $\mathcal{O}(|V|^2|E|)$.

Derigs & Metz [DM86] and Applegate & Cook [AC93] reported on the enormous savings using a fractional "jump-start" for the blossom algorithm in the 1-matching case. Miller & Pekny [MP95] modified Anstee's approach. Roughly speaking, instead of rounding on odd disjoint half integral cycles, their code iteratively looks for alternating paths connecting pairs of such cycles.

Padberg & Rao [PR82] developed a branch & cut approach for weighted b-matching. They showed that violated odd cut constraints can be detected in polynomial time by solving a minimum odd cut problem. Grötschel & Holland [GH85] reported a successful use of cutting planes for 1-matching problems. However, with present LP-solvers the solution time required to solve only the initial LP-relaxation, i.e. the fractional matching problem, is often observed to be in the range of the total run time required for the integral optimal solution by a pure combinatorial approach. Therefore, we did not follow this line of algorithms in our experiments.

With the exception of the paper by Miller & Pekny [MP95] we are not aware of a computational study on weighted b-matching. However, many ideas used for 1-matching can be reused and therefore strongly influenced our own approach. For example, Ball & Derigs [BD83] provide a framework for different implementation alternatives, but focus on how to achieve various asymptotical worst case guarantees. For a recent survey on computer implementations for 1-matching codes, we refer to [CR97]. In particular, the "Blossom IV" code of Cook & Rohe [CR97] seems to be the fastest available code for weighted 1-matching on very large scale instances.

Overview. The rest of the paper is organized as follows. In Section 2 we give a brief review of Pulleyblank's blossom algorithm. It will only be a simplified high-level presentation, but sufficient to discuss our experiments. In particular, the subtle technical details required to implement b-matching algorithms in comparison to 1-matching algorithms are omitted for clarity of the presentation. The experimental environment, classes of test instances and our test-suites are described in Section 3. Afterwards, we report on experiments which compare different algorithmic variants in Section 4, including a comparison of the performance of our code with that of Miller & Pekny [MP95]. Finally, in Section 5, we summarize the main observations and indicate directions of future research.

2 An Outline of the Primal-Dual Blossom Algorithm

In this section, we give a rough outline of the primal-dual algorithm as described by Pulleyblank [Pul73]. The purpose of this sketch is only to give a basis for the discussion of algorithmic variants which are compared in the subsequent sections. A self-contained treatment is given in the Appendix of [MS98].

For an edge set $F \subseteq E$ and a vector $x \in \mathbb{N}_0^{|E|}$, we will often use the implicit summation abbreviation $x(F) := \sum_{e \in F} x_e$. Similarly, we will use $b(W) := \sum_{v \in W} b_v$ for a node set $W \subset V$.

Linear programming formulation. The blossom algorithm is based on a linear programming formulation of the maximum weighted perfect b-matching problem. To describe such a formulation, the *blossom description*, let $\Omega := \{ S \subset V \mid |S| \geq 3 \text{ and } |b(S)| \text{ is odd} \}$ and $q_S := \frac{1}{2}(b(S)-1)$ for all $S \in \Omega$. Furthermore, for each $W \subset V$ let $\delta(W)$ denote the set of edges that meet exactly one node in W, and $\gamma(W)$ the set of edges with both endpoints in W. Then, a maximum weight b-matching solves the linear programming problem

$$\begin{array}{lrl}
 & \multicolumn{2}{l}{\text{maximize } c^T x} \\
 & \multicolumn{2}{l}{\text{subject to}} \\
\text{(P1)} & x(\delta(v)) = b_v & \text{for } v \in V \\
\text{(P2)} & x_e \geq 0 & \text{for } e \in E \\
\text{(P3)} & x(\gamma(S)) \leq q_S & \text{for } S \in \Omega.
\end{array}$$

The dual of this linear programming problem is

$$\begin{array}{lrl}
 & \multicolumn{2}{l}{\text{minimize } y^T b + Y^T q} \\
 & \multicolumn{2}{l}{\text{subject to}} \\
\text{(D1)} & y_u + y_v + Y(\Omega_\gamma(e)) \geq c_e & \text{for } e = (u,v) \in E \\
\text{(D2)} & Y_S \geq 0 & \text{for } S \in \Omega
\end{array}$$

with $\Omega_\gamma(e) := \{ S \in \Omega \mid e \in \gamma(S) \}$.

We define the *reduced costs* as $\bar{c}_e := y_u + y_v + Y(\Omega_\gamma(e)) - c_e$ for all $e \in E$. A b-matching x and a feasible solution (y, Y) of the linear program above are optimal if and only if the following complementary slackness conditions are satisfied:

$$\begin{array}{llcrl}
\text{(CS1)} & x_e > 0 & \Longrightarrow & \bar{c}_e = 0 & \text{for } e \in E \\
\text{(CS2)} & Y_S > 0 & \Longrightarrow & x(\gamma(S)) = q_S & \text{for } S \in \Omega.
\end{array}$$

A primal-dual algorithm. The primal-dual approach starts with some not necessarily perfect b-matching x and a feasible dual solution (y, Y) which satisfy together the complementary slackness conditions (CS1) and (CS2). Even more, the b-matching x satisfies (P2) and (P3). Such a starting solution is easy to find, in fact, $x \equiv 0$, $y_v := \frac{1}{2}\max\{c_e | e = (v,w) \in E\}$ for all $v \in V$ and $Y \equiv 0$ is a feasible choice.

The basic idea is now to keep all satisfied conditions as invariants throughout the algorithm and to work iteratively towards primal feasibility. The latter means that one looks for possibilities to augment the current matching. To maintain the complementary slackness condition (CS1) the search is restricted to the graph induced by edges of zero reduced costs with respect to the current dual solution, the so-called *equality subgraph* $G^=$. In a *primal step* of the algorithm, one looks for a maximum cardinality b-matching within $G^=$.

We grow a forest F which consists of trees rooted at nodes with a *deficit*, i.e. with $x(\delta(v)) < b_v$. Within each tree $T \in F$ the nodes are labeled *even* and *odd* according to the parity of the number of edges in the unique simple path to the root r (the root r itself is even). In addition, every even edge of a path from the root r to some node $v \in T$ must be matched, i.e. $x_e > 0$. *Candidate edges* to grow the forest are edges where one endpoint is labeled even and the other is either unlabeled or labeled even.

Augmentations are possible if there is a path of odd length between two deficit nodes on which we can alternatively add and subtract some δ from the current matching x without violating primal feasibility. Observe that we can augment if there is an edge between two nodes of different trees of F with the label even. In some cases, an augmentation is also possible if we have such an edge between even nodes of the same tree, but not always. It is the latter case which is responsible for complications. If no augmentation is possible and there is no further edge available to grow the forest, the forest is called *Hungarian forest.*

Edmonds' key insight was the observation that by *shrinking* of certain subgraphs (the *blossoms*) one can ensure that the tree growing procedure detects a way to augment the current b-matching, if the matching is not maximum. The reverse operation to shrinking is *expanding.* Hence, we are always working with a so-called *surface graph* which we obtain after a series of shrinking and expanding steps.

The main difference to 1-matching lies in the more complicated structure of the *blossoms* which we have to shrink into *pseudo-nodes.* Roughly speaking, when blossoms in the 1-matching case are merely formed by odd circuits C for which $x(\gamma(C)) = q_C$, a blossom B in the b-matching case contains such an odd circuit C but also the connected components of matched edges incident to nodes of C (*petals*). The additional complication is that C must be the only circuit of the blossom. (See Figure 1 for a sample blossom.)

If the primal step finishes with a maximum cardinality matching which is perfect, we are done and the algorithm terminates the primal-dual loop. Otherwise, we start a dual update step. Roughly speaking, its purpose is to alter the current dual solution such that new candidate edges are created to enter the current forest F. Depending on the label of a node and whether it is an original node or a pseudo-node we add or subtract some ε (but leave unlabeled nodes

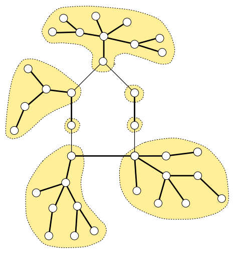


Fig. 1. A sample blossom with an odd circuit of length seven. Each shaded region corresponds to a petal.

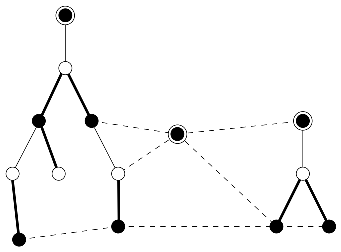


Fig. 2. Example of an augmenting forest consisting of three trees. Even (odd) nodes are filled (non-filled), root nodes equipped with an extra circle, non-forest edges are dashed, matched (unmatched) edges are drawn with thick (thin) lines.

unchanged to maintain (CS1)) which is chosen as the maximum value such that the reduced cost of all edges in the forest F remain unchanged (i.e. they remain in $G^=$), all other edges of the original G have non-negative reduced costs (D1), and the dual variables associated to pseudo-nodes remain non-negative (D2). If the dual variable associated to an odd pseudo-node becomes zero after a dual update, the pseudo-node will be expanded. This guarantees that no augmenting paths will be missed.

After finishing the primal-dual loop, all remaining pseudo-nodes are expanded, and the algorithm terminates.

3 Experimental Setup

In this section, we describe our test-suites, the computational environment, and how we compare code variants.

Graph classes and test-suites. Code variants are compared across real-world data from the mesh refinement application and artificial data on Euclidean nearest-neighbor graphs.

This study concentrates on extremely sparse graphs only, as our application data has this property. Moreover, the method of choice for dense problems is a two-stage approach which solves the problem on such a sparse graph substitute first, and makes pricing and repair steps afterwards [DM86,AC93].

1. **Real-world data: mesh refinement** As mentioned in the introduction, a successful approach to CAD mesh refinement into quadrilateral finite element meshes requires the solution of minimum cost perfect b-matching problems as a subroutine. We selected all available instances with non-trivial size (more than 3000 nodes) for our test-suite. These instances are extremely sparse,

the average vertex degree is about 2.5, the node capacities change between 1 and 1,091 with an average value of 2.6.

2. **Synthetic data: Euclidean nearest-neighbor graphs** We produce an artificial b-matching problem instance as follows, parameterized by the number of nodes n, the density $d = m/n$, and the node capacities b_{max}. We generate n nodes with random two-dimensional Euclidean coordinates and a random integral node capacity b_v within the interval $\{b_{min}, \ldots, b_{max}\}$ where $b_{min} = 1$. To create the edge set, we connect each node v to the d nearest neighbors with respect to the Euclidean distance and take the negative distance as edge cost (recall that the objective is to maximize). Afterwards, the edge set is filled up (to meet exactly the required density) by adding edges from edge-disjoint spanning trees with edge cost induced by the Euclidean distance as above.
   In order to guarantee the existence of at least one perfect b-matching, we afterwards add a node v_B with node potential $b_{v_B} = \sum_{v \in V \setminus \{v_B\}} b_v$ and connect this node to each other node (with a sufficiently large negative cost). Two further nodes with node potential b_{v_B} are added in order to form a triangle with v_B (the three triangle edges have cost zero). Note that this construction increases the degree of every original node by one, but we present the original values of n and d in this paper.

We constructed three test-suites for the latter graph class:

$\boxed{\text{E1}}$ varying the number of nodes n from 2^{10} to 2^{15}, for fixed density values $2, 4, \ldots, 16$ and $b_{max} = 10$,

$\boxed{\text{E2}}$ varying the density d from 2 to 16, for fixed node sizes $n = 2^{10}, \ldots, 2^{15}$ and $b_{max} = 10$,

$\boxed{\text{E3}}$ varying the maximum node capacity b_{max} from 2^0 to 2^{16}, for fixed $n = 2^{14}$ and $d = 2, 10, 16$.

All real-world problem instances and our generator for Euclidean nearest-neighbor graphs will be made available on the internet.

Computational environment and measurements. All experiments were performed on the same SUN UltraSPARC2 workstation with 256 MByte of RAM and a 200 MHz processor running under Solaris 2.6. We took care to set up all experiments in a range where memory limitations do not spoil the measured CPU-times through effects like paging. Our code was written in C++ and compiled under *egcs*[1], version 1.1.1., with the -O3 option. Times reported are user times in seconds obtained by `getrusage()`. The time measured is only the amount of time taken to compute the optimal b-matching. The time needed to read in the graph, to output the optimal b-matching, and to check the correctness was excluded for several reasons. The input/output part excluded from the time measurements is exactly the same for all algorithmic variants of our own code. Thus, to highlight differences between variants we measure only the

[1] Cygnus Solutions

parts where these variants differ. For the comparison with the executable of Miller & Pekny it is even necessary to exclude the input/output procedures as they may spoil a fair comparison due to different input/output formats. In the experiments with synthetic data, each data point represents an average over ten different instances. Comparisons across variants and codes are performed on the same ten inputs. The chosen number of ten samples was a compromise between practicability and the need of a reasonable basis for statistical investigations as the observed variance was relatively large.

Comparing different variants. Our implementation was carefully designed with respect to a flexible substitution of subalgorithms and data structures (for software engineering aspects see [MS98]). Among the many possible variants, it is only feasible to test a carefully selected subset of combinations. (Suppose, we consider only six parameters for a possible change which can be set at two or more levels, then a complete factorial design would give more than 64 variants.)

Through a series of initial experiments between certain variants we determined a *reference implementation* which reflects the "best combination of parameters on average" we could find. The reference implementation uses the parameters (JS), (DU1), (CS4), and (HB2) which will be described in detail below. In the experiments of the following section, we compare our reference implementation to an alternative implementation where a single parameter of interest is changed and all others are kept constant.

For fixed density d and fixed range of the node capacities b, it seems reasonable to assume that the running time $T(n)$ grows with $T(n) = c \cdot n^{\alpha}$ for some constants c and α. We conducted linear regressions based on this assumption to estimate the two unknown constants. (The regression was done on the raw data and not on the mean values shown in our plots). Whenever we report results of such a regression analysis, we checked that the regression fit is well and that the residuals from the fit do not show a suspicious pattern.

4 Experimental Comparison of Algorithmic Variants

Experiment 1: Fractional jump-start. In our first experiment we are interested in a quantitative analysis of the effect of the so-called "jump-start". The jump-start runs in several phases. Recall from the Introduction that it first solves the fractional matching problem, then rounds the half-integral solution, and finally starts the integral algorithm with this rounded solution.

Theoretical work by Anstee [Ans87] and computational studies reported by Derigs & Metz [DM86], and Applegate & Cook [AC93] for 1-matching, and by Miller & Pekny [MP95] as well as Avgetidis [Avg96] for b-matching clearly indicated that a significant speed-up can be expected by this technique.

In order to solve the fractional b-matching problem, we adapted the integral algorithm by simply switching the primal data type from `integer` to `double`. Each time a blossom is detected, we augment with a half-integral value and therefore introduce half-integral values on some cycle. If such a cycle is found

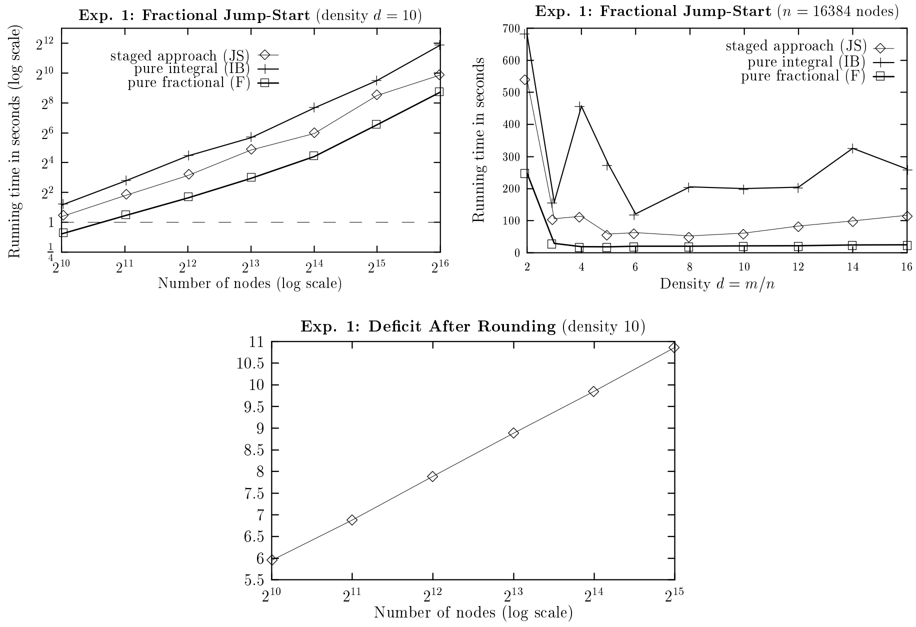


Fig. 3. The pure integral blossom algorithm (IB) vs. the staged approach with fractional jump-start (JS) vs. the fractional part only (F): running times for different number of nodes but fixed density $d = 10$ (left, top) and for different densities but fixed number of vertices $n = 16384$ (right, top). The third figure shows the remaining deficit after the fractional phase.

in a grow operation, an augmentation rounds all cycle values to integers. Note that shrinking is not necessary in the fractional algorithm.

We tested the following variants:

(IB) the pure blossom algorithm with an empty start matching,
(JS) the staged approach with a fractional matching algorithm in the first stage, and
(F) the fractional matching algorithm only.

Evaluation. We refer to Fig. 3 for the results of this experiment. Linear regression for density 10 yields $IB(n) = 3.262 \cdot 10^{-5} n^{1.597}$ as a best fit for the purely integral method with an adjusted $\bar{R}^2$-value (the square of the multiple correlation coefficient) of 0.945, whereas the jump-start code needs $JS(n) = 3.006 \cdot 10^{-5} n^{1.523}$ (with an adjusted $\bar{R}^2$-value of 0.960). The deficit after rounding grows linearly in the number of nodes.

Experiment 2: Dual update: Use of priority queues. Recall that the dual update consists of two main parts. First, the ϵ-value has to be determined by which the node potentials (y, Y) can be changed without losing dual feasibility

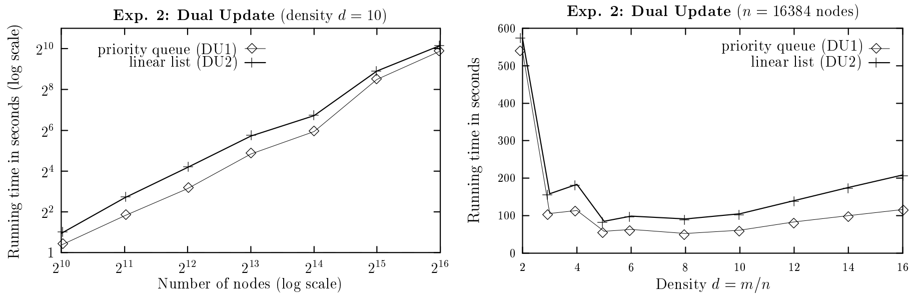


Fig. 4. Use of priority queue (d-heap) (DU1) vs. linear scan (DU2): running times for different number of nodes but fixed density $d = 10$ (left) and for different densities but fixed number of vertices $n = 16384$ (right).

or violating complementary slackness conditions of type (CS1). Second, for all labeled nodes, the node potentials are changed by ϵ (addition/subtraction is used depending on the individual node labels). The second part is a simple iteration over the nodes of the current forest and gives not much freedom for the implementation. The crucial part is the first one, where the ϵ-value is determined as the minimum determined for three sets: (1) the minimum with respect to reduced costs over all edges with one endpoint labeled even and the other being unlabeled, (2) half the minimum with respect to reduced costs over all edges between pairs of even-labeled nodes, (3) half the minimum with respect to dual potentials over all odd pseudo-nodes.

An important idea proposed by Ball & Derigs [BD83] is to partition the edges according to the labels of their endpoints and to keep and update this partition in a container data structure. We looked at two alternatives for containers, doubly linked lists and priority queues (more precisely d-heaps). The list implementation allows $\mathcal{O}(1)$ insert and delete operations but needs linear time to determine the minimum, whereas the d-heap implementation supports $\mathcal{O}(1)$-minimum operations but takes logarithmic time for insert/delete operations.

Miller & Pekny [MP95] reported on considerable savings by the use of d-heaps in the fractional phase. On the other hand it was interesting to note that the fast Blossom IV code of Cook & Rohe [CR97] for 1-matching does not use priority queues. This was our motivation to compare the two variants (DU1) which uses a d-heap with $d = 10$ and (DU2) which uses a linear list.

Evaluation. Fig. 4 demonstrates that the d-heap implementation is significantly faster within the range of our experiments, but the advantage seems to decrease for larger instances. The regression analysis yields $DU1(n) = 3.006 \cdot 10^{-5} n^{1.523}$ as a best fit for (DU1) (adjusted $\bar{R}^2$-value is 0.960) and $DU2(n) = 6.611 \cdot 10^{-5} n^{1.495}$ (adjusted $\bar{R}^2$-value of 0.970) for (DU2). The exponent of the estimated asymptotic running time for the d-heap variant appears to be slightly larger than for the simple list implementation, only the constant factors are much in favour for the d-heap. Hence, this is a typical example that extrapolation beyond the range

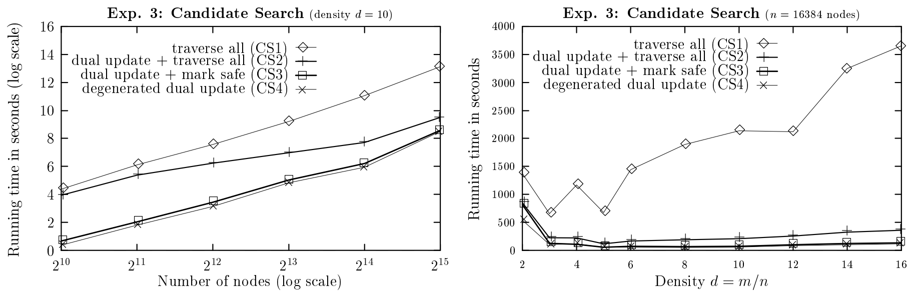


Fig. 5. The four candidate search strategies: Running times for different numbers of nodes but fixed density $d = 10$ (left) and for different densities but a fixed number of nodes $n = 16384$ (right).

of the experiments is not feasible. Looking at the chart for different densities we also observe that the improvement due to the use of the priority queue grows with the density.

Experiment 3: Candidate search. The candidate search for edges in the tree growing part of the algorithm (primal part) allows several strategies on which we elaborate in the next experiment. We implemented and tested four variants:

(CS1) The direct approach to find candidate edges for the growing of the current forest F is to traverse all trees of F and to examine iteratively all edges adjacent to even nodes.

(CS2) The purpose of the dual update is to change the node potentials such that new candidate edges become available. Hence, one can keep track of an initial list of edges which become new candidates already in the dual update. If, after some primal steps, this list becomes empty, a traversal as in (CS1) finds all other candidate edges.

(CS3) The previous idea can be refined with a simple trick: A heuristic to reduce the number of edges to be examined is to mark nodes as safe after the examination of their adjacency. As long as the label of a node does not change, the node remains safe and can be ignored for the further candidate search.

(CS4) A further variation of (CS2) is to allow *degenerated updates* with $\epsilon = 0$. This strategy (proposed by Ball & Derigs [BD83]) simply invokes a new dual update if the candidate list becomes empty.

Evaluation. As Fig. 5 shows, using the dual update as in (CS2), (CS3), and (CS4) allows a significant improvement for the candidate search. Linear regression yields $CS1(n) = 4.120 \cdot 10^{-4} n^{1.531}$ (adjusted $\bar{R}^2$-value is 0.994) for variant (CS1), whereas our estimation for the fastest variant (CS4) gives a more than 10 times smaller constant factor and a slightly better asymptotic, namely

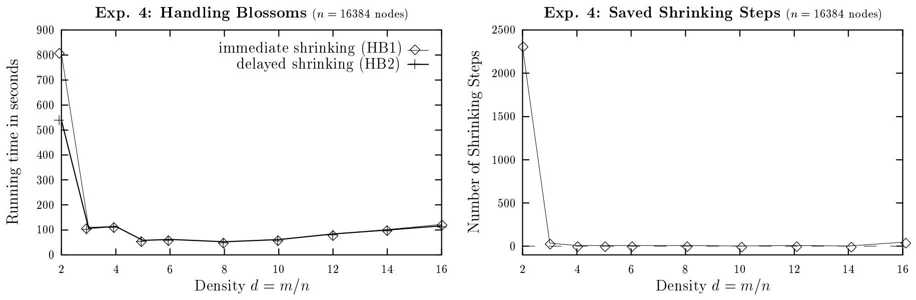


Fig. 6. Immediate shrinking of blossoms (HB1) vs. delayed shrinking (HB2): running times (left) and number of saved shrinking steps (right) for different densities.

$CS4(n) = 3.006 \cdot 10^{-5} n^{1.523}$ (with an adjusted $\bar{R}^2$-value of 0.960). Moreover, Variant (CS4) which uses the degenerated dual update is only slightly faster than all other variants, but is faster for each single instance we tested.

Experiment 4: Handling of blossoms. Applegate & Cook [AC93] proposed to incorporate a heuristic which delays the shrinking of blossoms if a blossom forming edge is found until no other candidate edge for tree growing is available. The rational behind this idea is to cut down the total number of blossom shrinking/expanding operations, as this number is conjectured to be crucial for the overall running time. We implemented and tested the following variants:

(HB1) shrink each detected blossom immediately, or

(HB2) try to avoid shrinking of blossoms, i.e. delay the shrinking until no other candidate edge is left.

Fig. 6 reports the running times for the two variants on graphs with $2^{14} = 16384$ vertices. (The corresponding charts for other graph sizes look similar.)

Evaluation. Delayed blossom shrinking decreases the total running time significantly for graphs with a very small density $d \leq 2$, but only slightly in general. This is in accordance with the number of saved blossom shrinking steps. Additionally, a closer look to the statistics of the test-suites shows that the simple heuristic of avoiding shrinking does neither guarantee that the total number of shrinking steps is minimal, nor the maximum or average nesting level is minimal, nor the number of shrunken nodes is minimal.

Experiment 5: Large node capacities. As mentioned in the introduction, Pulleyblank's version of the primal-dual algorithm is only pseudo-polynomial, but can be turned into a (strongly) polynomial algorithm if the fractional phase is solved in (strongly) polynomial time. Hence, worst case analysis would suggest to use scaling, and indeed, Miller & Pekny [MP95] claim that for problems with large *b*-values, scaling should be implemented in the fractional phase.

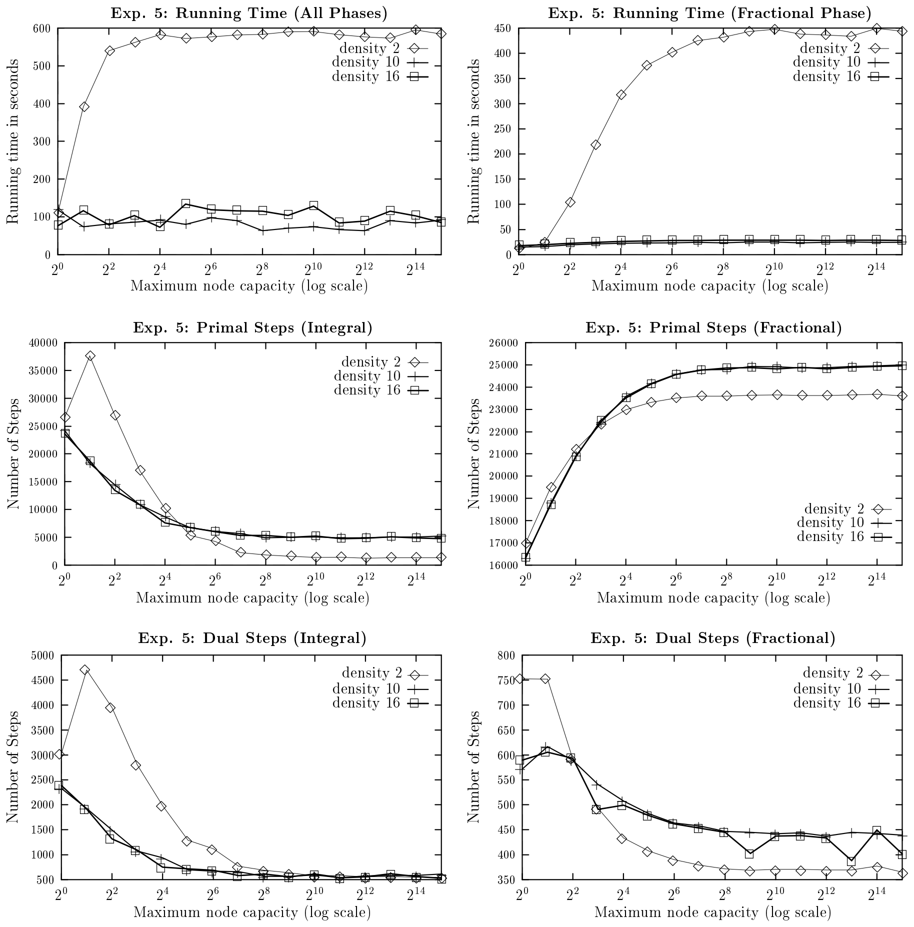


Fig. 7. The impact of the node capacities on the running time for graphs of fixed size $n = 16384$ and a fixed densities of $d = 2, 10, 16$. The charts show the running time for the staged algorithm (all phases) and the fractional phase, as well as the number of primal and dual steps in the integral and fractional phase, respectively.

However, earlier experiments by Avgetidis and Müller-Hannemann [Avg96] on cardinality b-matching showed a slow-down in actual performance if scaling was incorporated. Therefore, we decided not to use scaling in our implementation. Clearly, such a decision has to be justified, and the experiment on test-suite $\boxed{\text{E3}}$ was designed to test the conjecture that scaling is not necessary.

Evaluation. Fig. 7 shows that an increase of node capacities has almost no effect on the time to solve problems of test-suite $\boxed{\text{E3}}$, neither in the staged approach with a combined fractional and integral phase, nor in the fractional phase alone. There is again an anomaly for density $d = 2$ where a jump in running time oc-

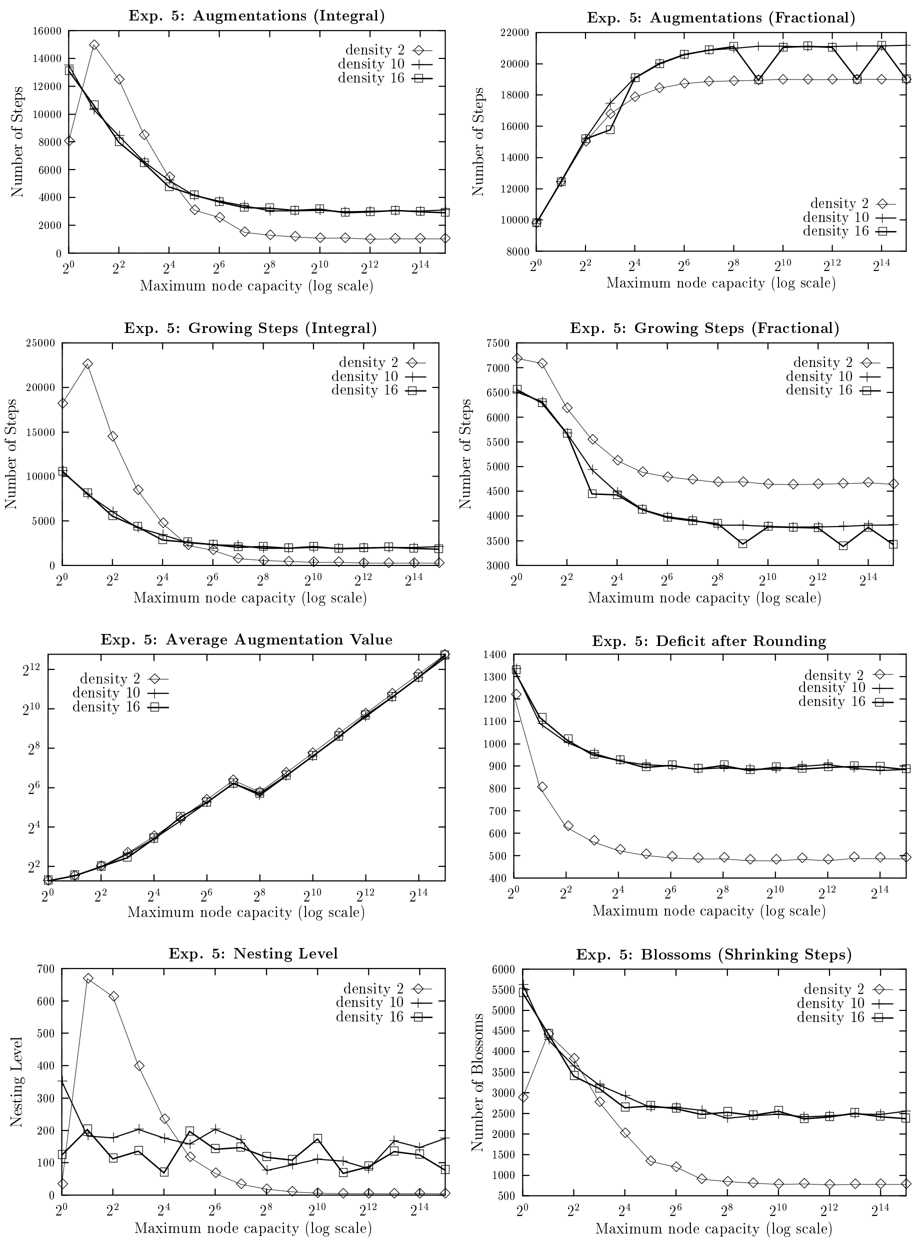


Fig. 8. Additional evaluations for the influence of the node capacities: The charts show the number of growing and augmentation steps in the integral and fractional phase, the average augmentation value, the remaining deficit after rounding at the beginning of the integral phase, and finally the average nesting level and the number of blossom shrinking steps.

curs from $b \equiv 1$ to $b > 1$, but the growth in running time vanishes for sufficiently large b-values. To understand these surprising observations, we performed a detailed analysis of the different algorithmic steps in both phases, shown in Figs. 7 and 8. First, we observe that the number of primal steps grows significantly only for small values of b and then stabilizes to an almost constant number in the fractional part, whereas the integral part shows the inverse trend. Second, the number of dual updates decreases slightly for both phases, once more with an exceptional behavior for density $d = 2$ and very small b-values. The primal steps have been further analyzed with respect to the major components, namely tree growing steps, augmentations, and blossom shrinking steps. It turns out that neither the number of tree growing nor the blossom shrinking operations increase with b. Apart from very small node capacities, the number of augmentations increases only very slightly in the fractional part, and even seems to decrease in the integral part.

There are two more crucial observations: One is that the deficit after rounding is almost independent from the node capacities. Hence, the amount of "remaining work" for the integral phase is nearly constant. The other is that the average augmentation value increases roughly with the same speed as the node capacities. The latter explains why the number of augmentations does not increase (as suspected).

Experiment 6: Comparison with Miller & Pekny. Miller & Pekny kindly provided us with an executable of their b-matching code which is specialized to solve geometric problem instances. Because of its limited interface we could test this code only on problems generated by the built-in problem generator. The class of nearest-neighbor instances provided by this generator is slightly different from ours as described in Section 3. In this experiment, sparse graphs are formed by taking for each node, the k nearest neighbors in each of the four quadrants of a coordinate system centered on the node. We performed experiments with k chosen as 3, 5, and 10 and node capacities in the interval $\{1, \ldots, 10\}$. The number of nodes varied from $n = 1024$ to $n = 8192$, and are increased in steps of 512.

Evaluation. Fig. 9 clearly demonstrates that our code outperforms the code of Miller & Pekny. Based on this test-suite, linear regression yields $T(n) = 8.190 \cdot 10^{-5} n^{1.327}$ as a best fit for our code with an adjusted $\bar{R}^2$-value of 0.981, whereas the Miller & Pekny code needs $MP(n) = 2.838 \cdot 10^{-7} n^{2.293}$ with an adjusted $\bar{R}^2$-value of 0.910. Figs. 10 and 11 show the fitted curves form our regression analysis together with the raw data.

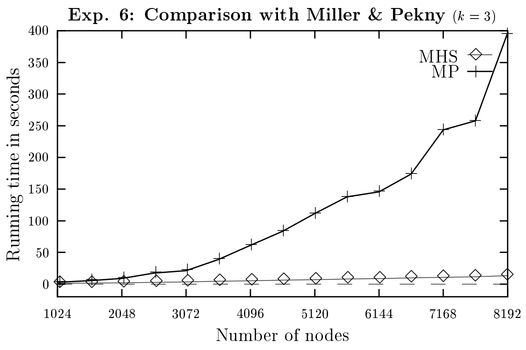


Fig. 9. A comparison of the code from Miller & Pekny (MP) with our *b*-matcher (MHS) on a test-suite of graphs with an Euclidean distance function, and $k = 3$ (density $d \approx 10$).

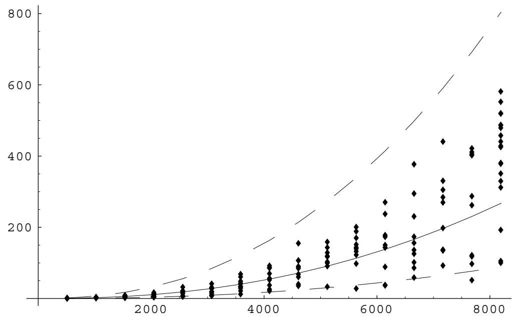


Fig. 10. Miller & Pekny: The raw data of Experiment 6 (CPU times in seconds), the fitted curve from the linear regression, and the 95% confidence intervals for the predicted values.

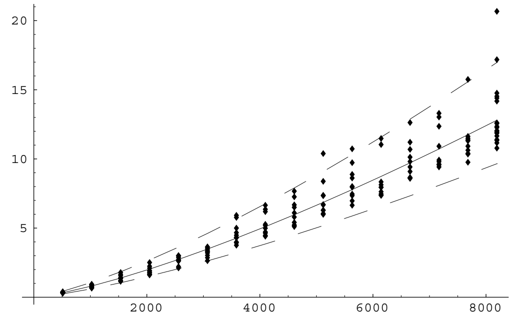


Fig. 11. Our implementation: The raw data of Experiment 6 (CPU times in seconds), the fitted curve from the linear regression, and the 95% confidence intervals for the predicted values.

Experiment 7: Mesh refinement. We tested all implementation variants used for the previous experiments also for our real-world instances from the mesh refinement application.

In the following table, the second column displays the number of nodes n, followed by the nesting level N and the number of blossoms B observed in our reference implementation (REF) in columns three and four. Our reference implementation uses the variants (JS), (DU1), (CS4) and (HB2). The remaining columns show the running time in seconds for all other implementation variants.

Sample	n	N	B	REF	F	IB	DU2	CS1	CS2	CS3	HB1
Cool5	4002	6	47	0.69	0.61	1.36	0.69	0.88	0.79	0.73	0.70
Cool10	3529	5	52	0.54	0.45	1.05	0.58	0.51	0.63	0.56	0.55
Cool15	3293	5	29	0.48	0.39	0.92	0.48	0.48	0.61	0.52	0.49
Benz15	3989	15	94	0.91	0.67	1.66	0.92	1.41	1.17	0.93	0.95
Benz25	3779	7	87	0.82	0.58	1.54	0.88	1.12	1.15	0.81	0.79
Benz35	3711	1	60	0.74	0.56	1.37	0.77	0.83	0.90	0.72	0.70
Wing10	5273	9	114	1.82	1.40	3.35	1.86	3.14	2.14	1.81	1.83
Wing15	5061	30	88	1.32	1.03	2.75	1.36	2.37	1.66	1.39	1.31
Wing20	4931	10	61	1.29	1.03	2.43	1.21	1.93	1.43	1.21	1.23
Pump5	6467	9	89	2.01	1.76	4.15	2.05	4.58	2.44	2.12	1.98
Pump10	6227	11	72	1.68	1.51	3.33	1.72	3.08	1.95	1.70	1.72
Pump15	6051	20	102	1.53	1.31	3.00	1.59	2.54	1.81	1.58	1.54
Shaft3	10176	1	70	4.25	4.13	8.37	4.00	8.97	4.44	4.05	4.11
Shaft5	9632	1	39	3.44	3.38	7.43	3.42	7.63	3.82	3.51	3.40
Shaft7	9016	1	122	2.91	2.75	6.25	2.96	2.92	3.18	2.95	3.02
Bend10	21459	91	415	16.95	15.23	33.79	17.16	18.15	19.83	16.75	17.28
Bend15	20367	5	131	13.90	13.29	29.78	14.32	7.97	14.36	13.51	14.10
Bend20	19969	16	137	13.49	12.76	28.70	13.88	8.18	14.43	13.44	14.22
Sum	**146932**	—	**1809**	**68.77**	**62.84**	**141.23**	**69.85**	**76.69**	**76.74**	**68.29**	**69.92**

Evaluation. The mesh refinement problems form a class of structured real-world instances with a high variance of the nesting level N and the number of blossoms B. Nevertheless, we are able to solve each instance in less than 17 seconds.

Our measurements for the mesh refinement test-suite confirm all of our previous results for the artificial test-suites. The reference implementation (REF) is slightly faster than the variants (DU2), (CS3), and (HB1) but clearly faster than all others.

5 Conclusion

This paper presents an experimental analysis of several variants of Pulleyblank's primal-dual b-matching algorithm. The choice of experiments was guided by a set of hypotheses based on previous computational experiments with 1-matching problems (Derigs & Metz [DM86], Applegate & Cook [AC93]) and with b-matching problems (Miller & Pekny [MP95]). The overall best variant, our reference implementation, not only solves all of our real-world problems from the mesh refinement test-suite in less than 17 seconds, it clearly outperforms the code of Miller & Pekny by a factor of about $\mathcal{O}(n^{0.9})$ on Euclidean nearest neighbor graphs.

On the synthetic instances, the asymptotic performance estimated by linear regression from our measurements is significantly lower than the worst-case predictions as discussed in the introduction. Most surprising for us was the marginal influence of the b-values on the overall running time. On the Euclidean nearest neighbor graph test-suites, our algorithms seems to behave like a strongly polynomial one. Operation counts and the observer technique enabled us to explain these observations.

It was also somewhat unexpected how much more difficult problems on the lowest density level turned out to be. However, this corresponds to a much higher number of blossom shrinking operations and a larger nesting level. Blossom shrinking is expensive but unfortunately a simple heuristic to reduce the number of shrinking steps is only successful for extremely sparse graphs which turned out to be the most difficult problem instances.

Bottleneck analysis as well as profiler results show that the most important part for an efficient implementation is the dual update in connection with the candidate search, where about 30% of the running time is spent. Attacking these bottlenecks should be most promising for further improvements.

Future work should try to develop new challenging classes of problem instances, and we would also like to test our algorithms on additional real world data. In addition, we will enlarge our analysis of representative operation counts following the lines of Ahuja & Orlin [AO96].

Acknowledgments. Both authors were partially supported by the special program "Efficient Algorithms for Discrete Problems and Their Applications" of the Deutsche Forschungsgemeinschaft (DFG) under grants Mo 446/2-2 and Mo 446/2-3. The authors wish to thank Donald Miller, Joseph Pekny, and his student Paul Bunch for providing us with an executable of their b-matching code.

References

AC93. D. Applegate and W. Cook, *Solving large-scale matching problems*, Network Flows and Matching, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (D. S. Johnson and C. C. McGeoch, eds.), vol. 12, 1993, pp. 557–576.

AMO93. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows*, Prentice Hall, 1993.

Ans87. R. P. Anstee, *A polynomial algorithm for b-matching: An alternative approach*, Information Processing Letters **24** (1987), 153–157.

AO96. R. K. Ahuja and J. B. Orlin, *Use of representative operation counts in computational testing of algorithms*, INFORMS Journal on Computing (1996), 318–330.

Avg96. I. Avgetidis, *Implementation und Vergleich von Lösungsverfahren für das maximale, ungewichtete b-Matching Problem*, Diploma thesis, Technische Universität Berlin, 1996.

BD83. M. O. Ball and U. Derigs, *An analysis of alternative strategies for implementing matching algorithms*, Networks **13** (1983), 517–549.

CR97. W. Cook and A. Rohe, *Computing minimum-weight perfect matchings*, Tech. Report 97863, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, 1997.

DM86. U. Derigs and A. Metz, *On the use of optimal fractional matchings for solving the (integer) matching problem*, Computing **36** (1986), 263–270.

Edm65. J. Edmonds, *Paths, trees, and flowers*, Can. J. Math. **17** (1965), 449–467.

EJ70. J. Edmonds and E. L. Johnson, *Matching: A well-solved class of integer linear programs*, Combinatorial Structures and their Applications, Calgary International Conference, Gordon and Breach (1970), 89–92.

EJL69. J. Edmonds, E. L. Johnson, and S. C. Lockhart, *Blossom I: a computer code for the matching problem*, unpublished report, IBM T. J. Watson Research Center, Yorktown Heights, New York, 1969.

Gab83. H. N. Gabow, *An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems*, Proceedings of the 15th Annual ACM Symposium on the Theory of Computing (1983), 448–456.

Ger95. A. M. H. Gerards, *Matching*, Handbooks in Operations Research and Management Science, vol. 7, North-Holland, 1995, pp. 135–224.

GH85. M. Grötschel and O. Holland, *Solving matching problems with linear programming*, Math. Prog. **33** (1985), 243–259.

Mar79. A. B. Marsh III, *Matching algorithms*, Ph.D. thesis, The John Hopkins University, Baltimore, 1979.

Mil95. D. L. Miller, *A matching based exact algorithm for capacitated vehicle routing problems*, ORSA J. of Computing (1995), 1–9.

MM97. R. H. Möhring and M. Müller-Hannemann, *Complexity and modeling aspects of mesh refinement into quadrilaterals*, Proceedings of the 8th Annual International Symposium on Algorithms and Computation, ISAAC'97, Singapore, Lecture Notes in Computer Science **1350**, Springer-Verlag, 1997, pp. 263–273.

MMW97. R. H. Möhring, M. Müller-Hannemann, and K. Weihe, *Mesh refinement via bidirected flows: Modeling, complexity, and computational results*, Journal of the ACM **44** (1997), 395–426.

MP95. D. L. Miller and J. F. Pekny, *A staged primal-dual algorithm for perfect b-matching with edge capacities*, ORSA J. of Computing **7** (1995), 298–320.

MS98. M. Müller-Hannemann and A. Schwartz, *Implementing weighted b-matching algorithms: Towards a flexible software design*, Technical report No. 591/1998, Fachbereich Mathematik, Technische Universität Berlin, 1998, an extended abstract appeared in Proceedings of 2nd Workshop on Algorithm Engineering, K. Mehlhorn (Ed.), 1998, pages 86–97, http://www.mpi-sb.mpg.de/~wae98/PROCEEDINGS/.

Orl88. J. B. Orlin, *A faster strongly polynomial minimum cost flow algorithm*, Proceedings of the 20th Annual ACM Symposium on Theory of Computing (1988), 377–387.

PR82. M. Padberg and M. R. Rao, *Odd minimum cut-sets and b-matchings*, Math. Oper. Res. **7** (1982), 67–80.

Pul73. W. R. Pulleyblank, *Faces of matching polyhedra*, Ph.D. thesis, Faculty of Mathematics, University of Waterloo, 1973.

Pul95. W. R. Pulleyblank, *Matchings and extensions*, Handbook of Combinatorics, vol. 1, North-Holland, 1995, pp. 179–232.

Designing Practical Efficient Algorithms for Symmetric Multiprocessors[*] (Extended Abstract)

David R. Helman and Joseph JáJá

Institute for Advanced Computer Studies &
Department of Electrical Engineering,
University of Maryland, College Park, MD 20742
{helman, joseph}@umiacs.umd.edu
www.umiacs.umd.edu/~{helman, joseph}

Abstract. Symmetric multiprocessors (SMPs) dominate the high-end server market and are currently the primary candidate for constructing large scale multiprocessor systems. Yet, the design of efficient parallel algorithms for this platform currently poses several challenges. In this paper, we present a computational model for designing efficient algorithms for symmetric multiprocessors. We then use this model to create efficient solutions to two widely different types of problems - linked list prefix computations and generalized sorting. Our novel algorithm for prefix computations builds upon the sparse ruling set approach of Reid-Miller and Blelloch. Besides being somewhat simpler and requiring nearly half the number of memory accesses, we can bound our complexity *with high probability* instead of merely *on average*. Our algorithm for generalized sorting is a modification of our algorithm for sorting by regular sampling on distributed memory architectures. The algorithm is a stable sort which appears to be asymptotically faster than any of the published algorithms for SMPs. Both of our algorithms were implemented in C using POSIX threads and run on four symmetric multiprocessors - the IBM SP-2 (High Node), the HP-Convex Exemplar (S-Class), the DEC AlphaServer, and the Silicon Graphics Power Challenge. We ran our code for each algorithm using a variety of benchmarks which we identified to examine the dependence of our algorithm on memory access patterns. In spite of the fact that the processors must compete for access to main memory, both algorithms still yielded scalable performance up to 16 processors, which was the largest platform available to us. For some problems, our prefix computation algorithm actually matched or exceeded the performance of the standard sequential solution using only a single thread. Similarly, our generalized sorting algorithm always beat the performance of sequential merge sort by at least an order of magnitude, even with a single thread.

[*] Supported in part by NSF grant No. CCR-9627210 and NSF HPCC/GCAG grant No. BIR-9318183. It also utilized the NCSA HP-Convex Exemplar SPP-2000 and the NCSA SGI/CRAY POWER CHALLENGEarray with the support of the National Computational Science Alliance under grant No. ASC970038N.

1 Introduction

Symmetric multiprocessors (SMPs) dominate the high-end server market and are currently the primary candidate for constructing large scale multiprocessor systems. Yet, the design of efficient parallel algorithms for this platform currently poses several challenges. The reason for this is that the rapid progress in microprocessor speed has left main memory access as the primary limitation to SMP performance. Since memory is the bottleneck, simply increasing the number of processors will not necessarily yield better performance. Indeed, memory bus limitations typically limit the size of SMPs to 16 processors. This has at least two implications for the algorithm designer. First, since there are relatively few processors available on an SMP, any parallel algorithm must be competitive with its sequential counterpart with as little as one processor in order to be relevant. Second, for the parallel algorithm to scale with the number of processors, it must be designed with careful attention to minimizing the number and type of main memory accesses.

In this paper, we present a computational model for designing efficient algorithms for symmetric multiprocessors. We then use this model to create efficient solutions to two widely different types of problems - linked list prefix computations and generalized sorting. Both problems are memory intensive, but in different ways. Whereas generalized sorting algorithms typically require a large number of memory accesses, they are usually to contiguous memory locations. By contrast, prefix computation algorithms typically require a more modest quantity of memory accesses, but they are are usually to non-contiguous memory locations.

Our novel algorithm for prefix computations builds upon the sparse ruling set approach of Reid-Miller and Blelloch [13]. Unlike the original algorithm, we choose the ruling set in such a way as to avoid the need for conflict resolution. Besides making the algorithm simpler, this change allows us to achieve a stronger bound on the complexity. Whereas Reid-Miller and Blelloch claim an *expected* complexity of $O\left(\frac{n}{p}\right)$ for $n >> p$, we claim a complexity *with high probability* of $O\left(\frac{n}{p}\right)$ for $n > p^2 \ln n$. Additionally, our algorithm incurs approximately half the memory costs of their algorithm, which we believe to be the smallest of any parallel algorithm we are aware of. Our algorithm for generalized sorting is a modification of our algorithm for sorting by regular sampling on distributed memory architectures [10]. The algorithm is a stable sort which appears to be asymptotically faster than any of the published algorithms we are aware of.

Both of our algorithms were implemented in C using POSIX threads and run on four symmetric multiprocessors - the IBM SP-2 (High Node), the HP-Convex Exemplar (S-Class), the DEC AlphaServer, and the Silicon Graphics Power Challenge. We ran our code for each algorithm using a variety of benchmarks which we identified to examine the dependence of our algorithm on memory access patterns. In spite of the fact that the processors must compete for access to main memory, both algorithms still yielded scalable performance up to 16 processors, which was the largest platform available to us. For some problems, our prefix computation algorithm actually matched or exceeded the performance of the standard sequential solution using only a single thread. Similarly, our generalized sorting algorithm always beat the performance of sequential merge sort by at least an order of magnitude, which from our experience is the best sequential sorting algorithm on these platforms.

The organization of our paper is as follows. **Section 2** presents our computational model for analyzing algorithms on symmetric multiprocessors. **Section 3** describes our prefix computation algorithm for this platform and its experimental performance. Similarly, **Section 4** describes our generalized sorting algorithm for this platform and its experimental performance.

2 A Computational Model for Symmetric Multiprocessors

For our purposes, the cost of an algorithm needs to include a measure that reflects the number and type of memory accesses. Given that we are dealing with a multi-level memory hierarchy, it is instructive to start with a brief overview of a number of models that have been proposed to capture the performance of multilevel hierarchical memories.

Many of the models in the literature are specifically limited to two-level memories. Aggarwal and Vitter [3] first proposed a simple model for main memory and disks which recognized the importance of spatial locality. In their uniprocessor model, a constant number of possibly non-contiguous blocks, each consisting of B contiguous records, can be transferred between primary and secondary memory in a single I/O. Vitter and Shriver [17] then proposed the more realistic D-disk model, in which secondary storage is managed by D physically distinct disk drives. In this model, D blocks can be transferred in a single I/O, but only if no two blocks are from the same disk. For both of these models, the cost of accessing data on disk was substantially higher than internal computation, and, hence, the sole measure of performance used is the number of parallel I/Os.

Alternatively, there are a number of models which allow for any arbitrary number of memory levels. Focusing on the fact that access to different levels of memory are achieved at differing costs, Aggarwal et al. [1] introduced the Hierarchical Memory Model (HMM), in which access to location x requires time $f(x)$, where $f(x)$ is any monotonic nondecreasing function. Taking note of the fact that the latency of memory access makes it economical to fetch a block of data, Aggarwal, Chandra, and Snir [2] extended this model to the Hierarchical Memory with Block Transfer Model (BT). In this model, accessing t consecutive locations beginning with location x requires time $f(x) + t$.

These models both assume that while the buses which connect the various levels of memory might be simultaneously active, this only occurs in order to cooperate on a single transfer. Partly in response to this limitation, Alpern et al. [4] proposed the Uniform Memory Hierarchy Model (UMH). In this model, the l^{th} memory level consists of $\alpha\rho^l$ blocks, each of size ρ^l, and a block of data can be transfered between level $l+1$ and level l in time $\rho^l/b(l)$, where $b(l)$ is the bandwidth. The authors of the UMH model stress that their model is an attempt to suggest what should be possible in order to obtain maximum performance. Certainly, the ability to specify the simultaneous, independent behavior of each bus would maximize computer performance, but as the authors acknowledge this is beyond the capability of current high-level programming languages. Hence, the UMH model seems unnecessarily complicated to describe the behavior of existing symmetric multiprocessors.

All the models mentioned so far focus on the relative cost of accessing different levels of memory. On the other hand, a number of shared memory models have focused instead on the contention caused by multiple processors competing to access main memory. Blelloch et al. [6] proposed the (d,x)-BSP model, an extension to the Bulk Synchronous Parallel model, in which main memory is partitioned amongst px banks. In this model, the time required for execution is modeled by five variables, which together describe the amount of time required for computation, the maximum number of memory requests made by a processor, and the maximum number of requests handled by a bank. The difficulty with this model is that the contention it describes depends on specific implementation details such as the memory map, which may be entirely beyond the control of the algorithm designer. A more general version of this model was suggested by Gibbons et al. [7]. Known as the Queue-Read Queue-Write (QRQW) PRAM model, it decomposes an algorithm into a series of synchronous steps. The time required for a given step is simply the maximum of the time required by any processor for computation, the number of memory accesses made by any processor, and the maximum number of requests made to any particular memory location. By focusing only on those requests which go to the same location, the QRQW model avoids implementation details such as the memory map, which makes it more appropriate as a high-level model. On the other hand, references which go to the same bank of memory but not to the same location can be just as disruptive to performance, and so ignoring details of the memory architecture can seriously limit the usefulness of the model.

In our SMP model, we acknowledge the dominant expense of memory access. Indeed, it has been widely observed that the rapid progress in microprocessor speed has left main memory access as the primary limitation to SMP performance. The problem can be minimized by insisting where possible on a pattern of contiguous data access. This exploits the contents of each cache line and takes full advantage of the pre-fetching of subsequent cache lines. However, since it does not always seem possible to direct the pattern of memory access, our complexity model needs to include an explicit measure of the number of non-contiguous main memory accesses required by an algorithm. Additionally, we recognize that efficient algorithm design requires the efficient decomposition of the problem amongst the available processors, and, hence, we also include the cost of computation in our complexity.

More precisely, we measure the overall complexity of an algorithm by the triplet $\langle M_A, M_E, T_C \rangle$, where M_A is the maximum number of accesses made by any processor to main memory, M_E is the maximum amount of data exchanged by any processor with main memory, and T_C is an upper bound on the local computational complexity of any of the processors. Note that M_A is simply a measure of the number of non-contiguous main memory accesses, where each such access may involve an arbitrary sized contiguous block of data. While we report T_C using the customary asymptotic notation, we report M_A and M_E as *approximations* of the actual values. By approximations, we mean that if C_A or C_V is described by the expression $\left(c_k x^k + c_{(k-1)} x^{(k-1)} + ... + c_0 x^0\right)$, then we report it using the approximation $\left(c_k x^k + o\left(x^k\right)\right)$. We distinguish between memory access cost and computational cost in this fashion because of the dominant expense of memory access on this architecture. With so few processors available, this coefficient is usually crucial in determining whether or not a parallel algorithm can be a viable replacement to the sequential alternative. On the other hand, despite the importance of these memory costs, we report only the highest order term, since otherwise the expression can easily become unwieldy.

In practice, it is often possible to focus on either M_A or M_E when examining the cost of algorithmic alternatives. For example, we observed when comparing prefix computation algorithms that the number of contiguous and non-contiguous memory accesses were always of the same asymptotic order, and therefore we only report M_A, which describes only the much more expensive non-contiguous accesses. Subsequent experimental analysis of the step-by-step costs has validated this simplification. On the other hand, algorithms for generalized sorting are usually all based on the idea of repeatedly merging sorted sequences, which are accessed in a contiguous fashion. Moreover, since our model is concerned only with the cost of main memory access, once values are stored in cache they may be accessed in any pattern at no cost. As a consequence, the number of non-contiguous memory accesses are always much less than the number of contiguous memory accesses, and in this situation we only report M_E, which includes the much more numerous contiguous memory accesses. Again, subsequent experimental analysis of the step-by-step costs has validated this simplification.

3 Prefix Computations

Consider the problem of performing a prefix computation on a linked list of n elements stored in arbitrary order in an array X. For each element X_i, we are given $X_i.succ$, the array index of its successor, and $X_i.data$, its input value for the prefix computation. Then, for any binary associative operator $\otimes$, the prefix computation is defined as:

$$X_i.prefix = \begin{cases} X_i.data & \text{if } X_i \text{ is the head of the list.} \\ X_i.data \otimes X_{(pre)}.prefix & \text{otherwise.} \end{cases} , \quad (1)$$

where *pre* is the index of the predecessor of X_i. The last element in the list is distinguished by a negative index in its successor field, and nothing is known about the location of the first element.

Any of the known parallel prefix algorithms in the literature can be considered for implementation on an SMP. However, to be competitive, a parallel algorithm must contend with the extreme simplicity of the obvious sequential solution. A prefix computation can be performed by a single processor with two passes through the list, the first to identify the head of the list and the second to compute the prefix values. The pseudocode for this obvious sequential algorithm is as follows:

- **(1):** Visit each list element X_i in order of ascending array index. If X_i is not the terminal element, then label its successor with index $X_i.succ$ as having a predecessor.
- **(2):** Find the one element not labeled as having a predecessor by visiting each list element X_i in order of ascending array index - this unlabeled element is the head of the list.
- **(3):** Beginning at the head, traverse the elements in the list by following the successor pointers. For each element traversed with index i and predecessor *pre*, set *List[i].prefix_data* = *List[i].prefix_data* $\otimes$ *List[pre].prefix_data.*

To compute the complexity, note that Step (1) requires at most n non-contiguous accesses to label the successors. Step (2) involves a single non-contiguous memory access to a block of n contiguous elements. Step (3) requires at most n non-contiguous memory accesses to update the successor of each element. Hence, this algorithm requires approximately $2n$ non-contiguous memory accesses and runs in in $O(n)$ computation time.

According to our model, however, the obvious algorithm is not necessarily the best sequential algorithm. The non-contiguous memory accesses of Step (1) can be replaced by a single contiguous memory access by observing that the index of the successor of each element is a unique value between 0 and $n-1$ (with the exception of the tail, which by convention has been set to a negative value). Since only the head of the list does not have a predecessor, it follows that together the successor indices comprise the set $\{0, 1, .., h-1, h+1, h+2, .., n-1\}$, where h is the index of the head. Since the sum of the complete set $\{0, 1, .., n-1\}$ is given by $\frac{1}{2}n(n-1)$, it easy to see that the identity of the head can be found by simply subtracting the sum of the successor indices from $\frac{1}{2}n(n-1)$. The importance of this lies in the fact that the sum of the successor indices can be found by visiting the list elements in order of ascending array index, which according to our model requires only a single non-contiguous memory access. The pseudocode for this improved sequential algorithm is as follows:

- **(1):** Compute the sum Z of the successor indices by visiting each list element X_i in order of ascending array index. The index of head of the list is $h = \left(\frac{1}{2}n(n-1) - Z\right)$.
- **(2):** Beginning at the head, traverse the elements in the list by following the successor pointers. For each element traversed with index i and predecessor *pre*, set *List[i]._prefix_data* = *List[i].prefix_data* $\otimes$ *List[pre].prefix_data.*

Since this modified algorithm requires no more than approximately n non-contiguous memory accesses while running in $O(n)$ computation time, it is optimal according to our model.

The first fast parallel algorithm for prefix computations was probably the list ranking algorithm of Wyllie [18], which requires at least $n \log n$ non-contiguous accesses. Other parallel algorithms which improved upon this result include those of Vishkin [16] ($5n$ non-contiguous accesses), Anderson and Miller [5] ($4n$ non-contiguous accesses), and Reid-Miller and Blelloch [13] ($2n$ non-contiguous accesses - see [8] for details of this analysis). Clearly, however, none of these match the memory requirement of our optimal sequential algorithm.

3.1 A New Algorithm for Prefix Computations

The basic idea behind our prefix computation algorithm is to first identify the head of the list using the same procedure as in our optimal sequential algorithm. We then partition the input list into s sublists by randomly choosing exactly one *splitter* from each memory block of $\frac{n}{(s-1)}$ elements, where s is $\Omega(p \log n)$ (the list head is also designated as a splitter). Corresponding to each of these sublists is a record in an array called *Sublists*. We then traverse each of these sublists, making a note at each list element of the index of its sublist and the prefix value of that element within the sublist. The results of these sublist traversals are also used to create a linked list of the records in *Sublists*, where the input value of each node is simply the sublist prefix value of the last element in the previous sublist. We then determine the prefix values of the records in the *Sublists* array by sequentially traversing this list from its head. Finally, for each element in the input list, we apply the prefix operation between its current prefix input value (which is its sublist prefix value) and the prefix value of the corresponding *Sublists* record to obtain the desired result.

The pseudo-code of our algorithm is as follows, in which the input consists of an array of n records called *List*. Each record consists of two fields, *successor* and *prefix_data*, where *successor* gives the integer index of the successor of that element and *prefix_data* initially holds the input value for the prefix operation. The output of the algorithm is simply the *List* array with the properly computed prefix value in the *prefix_data* field. Note that as mentioned above we also make use of an intermediate array of records called *Sublists*. Each *Sublists* record consists of the four fields *head*, *scratch*, *prefix_data*, and *successor*, whose purpose is detailed in the pseudo-code.

- **(1):** Processor P_i $(0 \leq i \leq p-1)$ visits the list elements with array indices $\frac{in}{p}$ through $\left(\frac{(i+1)n}{p} - 1\right)$ in order of increasing index and computes the sum of the successor indices. Note that in doing this a negative valued successor index is ignored since by convention it denotes the terminal list element - this negative successor index is however replaced by the value $(-s)$ for future convenience. Additionally, as each element of *List* is read, the value in the successor field is preserved by copying it to an identically indexed location in the array *Succ*. The resulting sum of the successor indices is stored in location i of the array Z.
- **(2):** Processor P_0 computes the sum T of the p values in the array Z. The index of the head of the list is then $h = \left(\frac{1}{2}n(n-1) - T\right)$.

- **(3):** For $j = \frac{is}{p}$ up to $\left(\frac{(i+1)s}{p} - 1\right)$, processor P_i randomly chooses a location x from the block of list elements with indices $\left((j-1)\frac{n}{(s-1)}\right)$ through $\left(j\frac{n}{(s-1)} - 1\right)$ as a splitter which defines the head of a sublist in *List* (processor P_0 chooses the head of the list as its first splitter). This is recorded by setting *Sublists[j].head* to x. Additionally, the value of *List[x].successor* is copied to *Sublists[j].scratch*, after which *List[x].successor* is replaced with the value $(-j)$ to denote both the beginning of a new sublist and the index of the record in *Sublists* which corresponds to its sublist.
- **(4):** For $j = \frac{is}{p}$ up to $\left(\frac{(i+1)s}{p} - 1\right)$, processor P_i traverses the elements in the sublist which begins with *Sublists[j].head* and ends at the next element which has been chosen as a splitter (as evidenced by a negative value in the *successor* field). For each element traversed with index x and predecessor *pre* (excluding the first element in the sublist), we set *List[x].successor = -j* to record the index of the record in *Sublists* which corresponds to that sublist. Additionally, we record the prefix value of that element within its sublist by setting *List[x].prefix_data = List[x].prefix_data* $\otimes$ *List[pre].prefix_data*. Finally, if x is also the last element in the sublist (but not the last element in the list) and k is the index of the record in *Sublists* which corresponds to the successor of x, then we also set *Sublists[j].successor = k* and *Sublists[k].prefix_data = List[x].prefix_data*. Finally, the *prefix_data* field of *Sublists[0]*, which corresponds to the sublist at the head of the list is set to the prefix operator identity.
- **(5):** Beginning at the head, processor P_0 traverses the records in the array *Sublists* by following the successor pointers from the head at *Sublists[0]*. For each record traversed with index j and predecessor *pre*, we compute the prefix value by setting *Sublists[j].prefix_data = Sublists[j].prefix_data* $\otimes$ *Sublists[pre].prefix_data*.
- **(6):** Processor P_i visits the list elements with array indices $\frac{in}{p}$ through $\left(\frac{(i+1)n}{p} - 1\right)$ in order of increasing index and completes the prefix computation for each list element x by setting *List[x].prefix_data = List[x].prefix_data* $\otimes$ *Sublists[-(List[x].successor)].prefix_data*. Additionally, as each element of *List* is read, the value in the successor field is replaced with the identically indexed element in the array *Succ*. Note that is reasonable to assume that the entire array of s records which comprise *Sublists* can fit into cache.

We can establish the complexity of this algorithm with high probability - that is with probability $\geq (1 - n^{-\epsilon})$ for some positive constant ϵ. But before doing this, we need the results of the following Lemma, whose proof has been omitted for brevity [8].

Lemma 1. *The number of list elements traversed by any processor in Step (4) is at most* $\alpha\frac{n}{p}$ with high probability, *for any* $\alpha(s) \geq 2.62$ *(read* $\alpha(s)$ *as "the function* α *of* s*"),* $s \geq (p \ln n + 1)$*, and* $n > p^2 \ln n$.

With this result, the analysis of our algorithm is as follows. In Step (1), each processor moves through a contiguous portion of the list array to compute the sum of the indices in the *successor* field and to preserve these indices by copying them to the array *Succ*. When this task is completed, the sum is written to the array Z. Since this is done in order of increasing array index, it requires only three non-contiguous memory accesses to exchange approximately $\frac{2n}{p}$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time. In Step (2), processor P_0 computes the sum of the p entries in the array Z. Since this is done in order of increasing array index, this step requires only a single non-contiguous memory accesses to exchange p elements with main memory and $O(p)$ computation time. In Step (3), each processor randomly chooses $\frac{s}{p}$ splitters to be the heads of sublists. For each of these sublists, it copies the index of the corresponding record in the *Sublists* array into the successor field of the splitter. While the *Sublists* array is traversed in order of increasing array index, the corresponding splitters may lie in mutually non-contiguous locations and so the whole process may require $\frac{s}{p}$ non-contiguous memory accesses to exchange $\frac{2s}{p}$ elements with main memory and $\frac{s}{p}$ computation time. In Step (4), each processor traverses the sublist associated with each of its $\frac{s}{p}$ splitters, which together contain at most $\alpha(s)\frac{n}{p}$ elements *with high probability.* As each sublist is completed, the prefix value of the last element in the subarray is written to the record in the *Sublists* array which corresponds to the succeeding sublist. Since we can reasonably assume that $(s << n)$ and can therefore ignore the cost of writing to the *Sublists* array, this step requires approximately $\alpha(s)\frac{n}{p}$ non-contiguous memory accesses to exchange approximately $\alpha(s)\frac{n}{p}$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time *with high probability* . However, it is important to note that an $\frac{s}{n}$-biased binomial process requires *on average* $\frac{n}{s}$ events before encountering the first success and so *on average* each processor traverses about $\frac{n}{p}$ list elements (which is what we observe experimentally in the next section). In Step (5), processor P_0 traverses the the linked list of s records in the *Sublists* array established in Step (4) to compute their prefix values, which requires s non-contiguous memory accesses to exchange s elements with main memory and $O\left(s\right)$ computation time. Finally, in Step (6), each processor completes the prefix values for a contiguous chunk of the input list by first looking up the prefix value of the record in *Sublists* which maps to the head of its sublist. Since we make the reasonable assumption that the entire array of s records which comprise *Sublists* will fit into the cache, which is the case for all three platforms considered in this paper and the choices for n, accessing the prefix values in the *Sublists* array will only require s non-contiguous memory accesses (non-contiguous because we are assuming they are accessed in the order of request). As the computation of the prefix value for an element is completed, the correct value is restored to its *successor* field from the array *Succ.* Hence, this step will require approximately $(s+1)$ non-contiguous memory accesses to exchange approximately $\frac{2n}{p}$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time. Thus, *with high probability*, the overall complexity of our prefix computation algorithm is given by

$$T(n,p) = \langle M_A(n,p); M_E(n,p); T_C(n,p)\rangle \tag{2}$$
$$= \left\langle \alpha(s)\frac{n}{p}; \left((\alpha(s)+4)\frac{n}{p}\right); O\left(\frac{n}{p}\right)\right\rangle \tag{3}$$

for $\alpha(s) \geq 2.62$, $s \geq (p \ln n + 1)$, $n >> s$, and $n > p^2 \ln n$. Noting that the relatively expensive M_A non-contiguous memory accesses comprise a substantial proportion of the M_E total elements exchanged with memory, and recalling that *on average* each processor traverses only about $\frac{n}{p}$ elements in Step (4), we would expect that in practice the complexity of our algorithm can be characterized by

$$T(n,p) = \langle M_A(n,p); T_C(n,p)\rangle \tag{4}$$
$$= \left\langle \frac{n}{p}; O\left(\frac{n}{p}\right)\right\rangle, \tag{5}$$

Notice that our algorithm's requirement of approximately n non-contiguous memory accesses is nearly half the cost of Reid-Miller and Blelloch and compares very closely with the requirements of the optimal sequential algorithm.

3.2 Performance Evaluation

Both our parallel algorithm and the optimal sequential algorithm were implemented in C using POSIX threads and run on an IBM SP-2 (High Node), an HP-Convex Exemplar (S-Class), a DEC AlphaServer 2100A system, and an SGI Power Challenge. To evaluate these algorithms, we examined the prefix operation of of floating point addition on three different benchmarks, which were selected to compare the impact of various memory access patterns. These benchmarks are the **Random [R]**, in which each successor is randomly chosen, the **Stride [S]**, in which each successor is (wherever possible) some stride S away, and the **Ordered [O]**, in which which element is paced in the array according to its rank. See [8] for a more detailed description and justification of these benchmarks.

The graphs in Fig. 1 compare the performance of our optimal parallel prefix computation algorithm with that of our optimal sequential algorithm. Notice first that our parallel algorithm almost always outperforms the optimal sequential algorithm with only one or two threads. The only exception is the [O] benchmark, where the successor of an element is always the next location in memory. Notice also that for a given algorithm, the [O] benchmark is almost always solved more quickly than the [S] benchmark, which in turn is always solved more quickly than the [R] benchmark. A step by step breakdown of the execution time for the HP-Convex Exemplar in Table 1 verifies that these differences are entirely due to the time required for the sublist traversal in Step (4). This agrees well with our theoretical expectations, since in the [R] benchmark, the location of the successor is randomly chosen, so almost every step in the traversal involves accessing a non-contiguous location in memory. By contrast, in the [O] benchmark, the memory location of the successor is always the successive location in memory, which in all likelihood is already present in cache. Finally, the [S] benchmark is designed so that where possible the successor is always a constant stride away. Since for our work this stride is chosen to be 1001, we might expect that each successive memory access would be to a non-contiguous memory location, in which case the [S] benchmark shouldn't perform any better than the [R] benchmark. However, cache modeling reveals that as the the number of samples increases, the number of cache misses decreases. Hence, for large value of s, the majority of requests are already present in cache, which explains the superior performance of the [S] benchmark. Finally, notice that, in Table 1, the n noncontiguous memory required by the [R] benchmark in Step (4) consume on average almost five time as much time as the $4n$ contiguous memory accesses of Steps (1) and (6). Taken as a whole, these results strongly support the emphasis we place on minimizing the number of non-contiguous memory accesses in this problem.

Table 1. Comparison of the time (in seconds) required as a function of the benchmark for each step of computing the prefix sums of 4M list elements on an HP-Convex Exemplar, for a variety of threads.

Step:	**Number of Threads & Benchmark**														
	[1]			**[2]**			**[4]**			**[8]**			**[16]**		
	[R]	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**
(1)-(3):	0.59	0.87	0.66	0.34	0.40	0.34	0.18	0.21	0.18	0.10	0.12	0.10	0.08	0.08	0.08
(4):	6.69	1.86	2.33	3.40	1.08	1.17	1.75	0.57	0.59	0.96	0.31	0.30	0.74	0.22	0.18
(5):	0.01	0.12	0.01	0.01	0.04	0.01	0.01	0.05	0.01	0.01	0.06	0.01	0.01	0.02	0.01
(6):	0.69	0.75	0.69	0.37	0.38	0.35	0.21	0.20	0.19	0.11	0.12	0.11	0.09	0.12	0.08
Total:	7.97	3.60	3.68	4.12	1.91	1.87	2.14	1.03	0.97	1.19	0.60	0.52	0.92	0.41	0.35

The graphs in Figs. 2 and 3 examine the scalability of our prefix computation algorithm as a function of the number of threads. Results are shown for a variety of problem sizes on both the HP-Convex Exemplar and the the IBM SP-2 using the [R] benchmark. Bearing in mind that these graphs are log-log plots, they show that for large enough inputs, the execution time decreases as we increase the number of threads p, which is the expectation of our model. For smaller inputs and larger numbers of threads, this inverse relationship between the execution time and the number of threads deteriorates. In this case, such performance is quite reasonable if we consider the fact that for small problem sizes the size of the cache approaches that of the problem. This introduces a number of issues which are beyond the intended scope of our computational model.

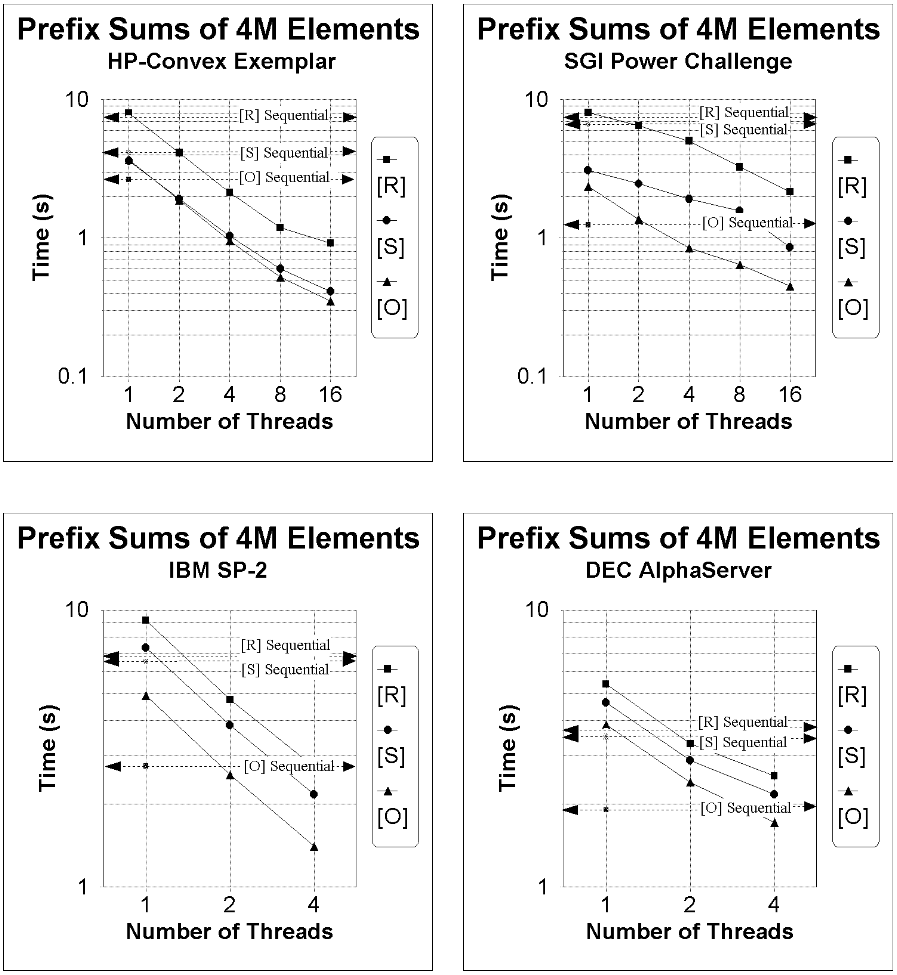


Fig. 1. Comparison between the performance of our parallel algorithm and our optimal sequential algorithm on four different platforms using three different benchmarks.

4 Generalized Sorting

Consider the problem of sorting n elements equally distributed amongst p processors, where we assume without loss of generality that p divides n evenly. Any of the algorithms that have been proposed in the literature for sorting on hierarchical memory models can be considered for possible implementation on an SMP. However, without modifications, most are unnecessarily complex or inefficient for a relatively simple platform such as ours. A notable exception is the algorithm of Varman et al. [15]. Yet another approach is an adaptation of our sorting by regular sampling algorithm [10,9], which we originally developed for distributed memory machines.

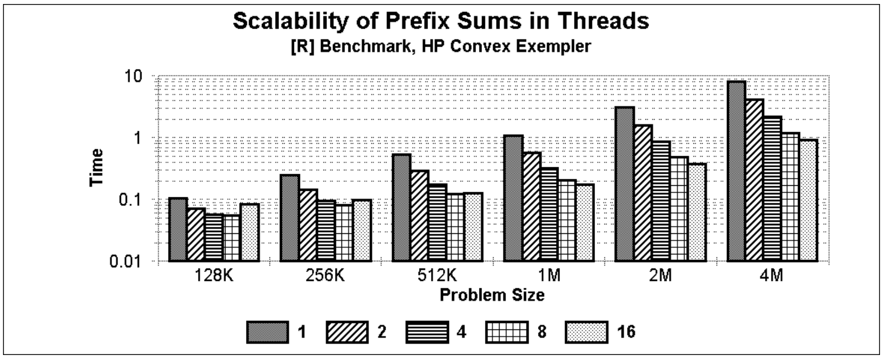


Fig. 2. Scalability of our prefix computation algorithm on the HP-Convex Exemplar with respect to the number of threads, for differing problem sizes.

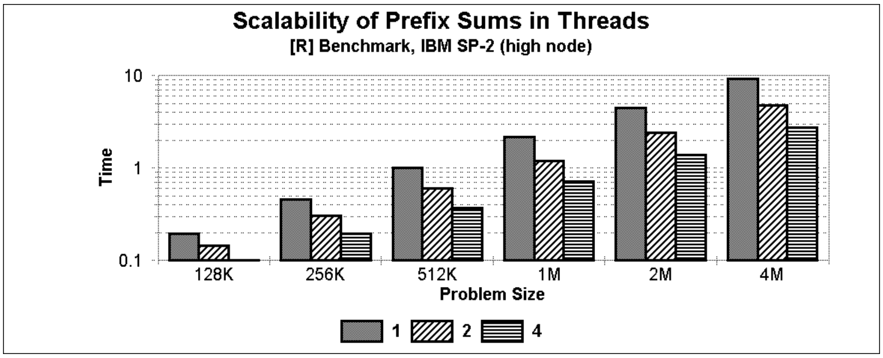


Fig. 3. Scalability of our prefix computation algorithm on the IBM SP-2 with respect to the number of threads, for differing problem sizes.

4.1 A New Algorithm for Generalized Sorting

The idea behind sorting by regular sampling is to first partition the n input elements into p memory-contiguous blocks and then sort each of these blocks using an appropriate sequential algorithm. Then, a set of $p-1$ *splitters* is found to partition each of these p sorted sequences into p subsequences indexed from 0 up to $(p-1)$, such that every element in the i^{th} group is less than or equal to each of the elements in the $(i+1)^{th}$ group, for $(0 \leq i \leq p-2)$. Then the task of sorting merging the p subsequences with a particular index can be turned over to the correspondingly indexed processor, after which the n elements will be arranged in sorted order. One way to choose the splitters is by regularly sampling the input elements - hence the name Sorting by Regular Sampling. As modified for an SMP, this algorithm is similar to the parallel sorting by regular sampling (PSRS) algorithm of Shi and Schaeffer [14]. However, unlike their algorithm, our algorithm accommodates the presence of duplicate values without the overhead of tagging each element.

However, while our algorithm will efficiently partition the work amongst the available processors, it will not be sufficient to minimize main memory accesses unless we also carefully specify how the sequential tasks are to be performed. Specifically, straightforward binary merge sort or quick sort will require $\log \frac{n}{p}$ memory accesses for each element to be sorted. Thus, a more efficient sequential sorting algorithm running on a single processor can be expected to outperform a parallel algorithm running on the relatively few processors available with an SMP, unless the sequential steps of the parallel algorithm are properly optimized. Knuth [11] describes a better approach for the analogous situation of external sorting. First, each processor partitions its $\frac{n}{p}$ elements to be sorted into blocks of size $\frac{C}{2}$, where C is the size of the cache, and then sorts each of these blocks using merge sort. This alone eliminates $\log\left(\frac{C}{4}\right)$ memory accesses for each element. Next, the sorted blocks are merged z at a time using a tournament of losers, which further reduces the memory accesses by a factor of $\log z$. To be efficient, the parameter z must be set less than $\frac{C}{L}$, where L is the cache line size, so that the cache can hold the entire tournament tree plus a cache line from each of the z blocks being merged. Otherwise, as our experimental evidence demonstrates, the memory performance will rapidly deteriorate. Note that this approach to sequential sorting was simultaneously utilized by LeMarca and Ladner [12], though without noting this important limitation on the size of z.

The pseudocode for our algorithm is as follows:

```
– (1) Each processor $P_i$ $(0 \le i \le p-1)$ sorts the subsequence of the $n$ input
  elements with indices $\left(\frac{in}{p}\right)$ through $\left(\frac{(i+1)n}{p} - 1\right)$ as follows:
    • (A) Sort each block of $m$ input elements using sequential merge sort,
      where $m \le \frac{C}{2}$.
    • (B) For $j = 0$ up to $\left(\frac{\log(n/pm)}{\log(z)} - 1\right)$, merge the sorted blocks of size
      $\left(mz^j\right)$ using $z$-way merge, where $z < \frac{C}{L}$.
– (2) Each processor $P_i$ selects each $\left(\frac{in}{p} + (j+1)\frac{n}{ps}\right)^{th}$ element as a sample,
  for $(0 \le j \le s-1)$ and a given value of $s$ $\left(p \le s \le \frac{n}{p^2}\right)$.
– (3) Processor $P_{(p-1)}$ merges the $p$ sorted subsequences of samples and then
  selects each $((k+1)s)^{th}$ sample as Splitter[$k$], for $(0 \le k \le p-2)$. By
  default, the $p^{th}$ splitter is the largest value allowed by the data type used.
  Additionally, binary search is used to compute for the set of samples with
  indices 0 through $((k+1)s - 1)$ the number of samples Est[$k$] which share
  the same value as Splitter[$k$].
– Step (4): Each processor $P_k$ uses binary search to define an index $b_{(i,k)}$ for
  each of the $p$ sorted input sequences created in Step (1). If we define $T_{(i,k)}$
  as a subsequence containing the first $b_{(i,k)}$ elements in the $i^{th}$ sorted input
  sequence, then the set of $p$ subsequences $\{T_{(0,k)}, T_{(1,k)}, ..., T_{((p-1),k)}\}$ will
  contain all those values in the input set which are strictly less than Splitter[$k$]
```

and *at most* $\left(\text{Est}[k] \times \frac{n}{ps}\right)$ elements with the same value as Splitter[k]. The term *at most* is used because there may not actually be this number of elements with the same value as Splitter[k].

– **Step (5):** Each processor P_k merges those subsequences of the sorted input sequences which lie between indices $b_{(i,(k-1))}$ and $b_{(i,k)}$ using p-way merge.

Before establishing the complexity of this algorithm, we need the results of the following lemma, whose proof has been omitted for brevity [10]:

Lemma 2. *At the completion of the partitioning in Step (4), no more than* $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ *elements will be associated with any splitter, for* $\left(p \leq s \leq \frac{n}{p^2}\right)$ *and* $n \geq ps$.

With this result, the analysis of our algorithm is as follows. In Step (1A), each processor moves through a contiguous portion of the input array to sort it in blocks of size m using sequential merge sort. If we assume that $\left(m \leq \frac{C}{2}\right)$, this will require only a single non-contiguous memory accesses to exchange $\frac{2n}{p}$ elements with main memory and $O\left(\frac{n}{p}\log m\right)$ computation time. Step (1B) involves $\frac{\log(n/pm)}{\log(z)}$ rounds of z-way merge. Since round j will begin with $\frac{n}{pmz^j}$ blocks of size mz^j, this will require at most $\frac{2nz}{pm(z-1)}$ non-contiguous memory accesses to exchange $\frac{2n\log(n/pm)}{p\log(z)}$ elements with main memory memory and $O\left(\frac{n}{p}log\left(\frac{n}{pm}\right)\right)$ computation time. The selection of s noncontiguous samples by each processor in Step (2) requires s non-contiguous memory accesses to exchange $2s$ elements with main memory and $O(s)$ computation time. Step (3) involves a p-way merge of blocks of size s followed by p binary searches on segments of size s. Hence, it requires approximately $p\log(s)$ non-contiguous memory accesses to exchange approximately $2sp$ elements with main memory and $O(sp\log p)$ computation time. Step (4) involves p binary searches by each processor on segments of size $\frac{n}{p}$ and hence requires approximately $p\log\left(\frac{n}{p}\right)$ non-contiguous memory accesses to exchange approximately $p\log\left(\frac{n}{p}\right)$ elements with main memory and $O\left(p\log\left(\frac{n}{p}\right)\right)$ computation time. Step (5) involves a p-way merge of p sorted sequences whose combined length from Lemma (2) is at most $\left(\frac{n}{p} + \frac{n}{s} - p\right)$. This requires approximately p non-contiguous memory accesses to exchange approximately $2\left(\frac{n}{p} + \frac{n}{s}\right)$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time. Hence, the overall complexity of our sorting algorithm is given by

$$\begin{aligned} T(n,p) &= \langle M_A(n,p); M_E(n,p); T_C(n,p)\rangle \\ &= \left\langle \left(\frac{nz}{pm(z-1)} + s + p\log\left(\frac{n}{p}\right)\right); \right. \\ &\qquad \left. \left(\left(2\frac{\log(n/pm)}{\log(z)} + 2\right)\frac{n}{p} + 2\frac{n}{s}\right); O\left(\frac{n}{p}\log n\right)\right\rangle \end{aligned} \tag{6}$$

for $\left(p \leq s \leq \frac{n}{p^2}\right)$, $n \geq ps$, $m \leq \frac{C}{2}$, and $z \leq \frac{C}{L}$. Since the analysis suggests that the parameters m and z should be as large as possible subject to the stated constraints while selecting s so that $\left(p << s << \frac{n}{p}\right)$, we would expect that in practice the complexity of our algorithm could be characterized as

$$T(n,p) = \langle M_E(n,p); T_C(n,p)\rangle \tag{7}$$
$$= \left\langle \left(4 + 2\frac{\log(n/pm)}{\log(z)}\right)\frac{n}{p}; O\left(\frac{n}{p}\log n\right)\right\rangle . \tag{8}$$

4.2 Performance Evaluation

Our sorting algorithm was implemented in C using POSIX threads and run on an IBM SP-2 (High Node), an HP-Convex Exemplar (S-Class), a DEC AlphaServer 2100A system, and an SGI Power Challenge. We ran our code using six widely different double precision floating point benchmarks which were selected to test the dependence of our algorithm on the input distribution. A detailed description and justification of these benchmarks is presented in [9]. The results in Table 2 verify that as expected performance does not significantly depend on the input distribution. Because of this independence, the remainder of this section will only discuss performance on the single benchmark [U], in which the input data forms a uniform random distribution.

Table 2. Sorting doubles (in seconds) using 4 threads on a DEC AlphaServer 2100A.

Input	**Benchmark**					
Size	[U]	[G]	[Z]	**[WR]**	**[DD]**	**[RD]**
512K	0.397	0.394	0.320	0.421	0.337	0.348
1M	0.868	0.856	0.741	0.844	0.724	0.710
2M	1.64	1.72	1.39	1.73	1.40	1.51
4M	3.50	3.47	3.00	3.52	3.01	2.98

Table 3 displays the times required to sort 4M *doubles* (i.e. double precision floating point values) on the HP-Convex Exemplar using a single thread as a function of m and z. Notice first that performance suffers dramatically when the block size reaches 1MB (128K eight byte double precision numbers), which is the limit of the single level cache on the Exemplar. This is expected, since sorting a block in Step (1A) now requires that data be repeatedly swapped to main memory. Consider also the data for a given block size - say 1K. The execution time drops as we move from $z = 2$ to $z = 16$. This is reasonable since we require 12 rounds of 2-way merge, 6 rounds of 4-way merge, 4 rounds of 8-way merge, and only 3 rounds of 16-way merge, and each round of z-way merge is obviously another round where all the input elements must be brought in from main memory. Moving from $z = 16$ to $z = 32$ has little effect on the execution time since it does nothing to reduce the memory requirements, but moving to $z = 64$ saves a round of memory access and, hence, the execution time is further reduced. However, the most dramatic illustration of the importance of minimizing secondary memory access can be found by comparing the optimal sorting time of 15.86 seconds for $m = 2K$ and $z = 2048$ with the time of 39.25 seconds required to sort using only binary merge sort. Reducing memory access by a combination of block sorting and z-way merging improved the performance by 60%. Clearly, such results strongly support the attention that we place in this algorithm on the number of contiguous memory accesses.

Table 3. Time (in seconds) required on the HP-Convex Exemplar to sort 4M doubles using a single thread as a function of M and z.

Block	**Denomination of z-Way Merge**											
Size	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
1K	30.43	21.49	19.22	17.66	17.87	16.20	16.74	16.91	16.87	16.82	16.73	16.29
2K	29.14	21.65	19.15	18.02	17.95	16.63	16.91	16.98	16.79	18.29	15.86	
4K	27.96	20.55	19.62	18.29	16.65	17.00	17.14	17.06	16.86	15.91		
8K	27.59	21.55	19.18	19.27	17.90	18.07	18.04	17.84	17.08			
16K	26.69	20.73	19.84	18.21	18.50	18.53	18.43	17.83				
32K	27.77	23.14	22.14	20.81	20.88	20.90	20.41					
64K	29.98	25.46	24.26	24.51	24.51	23.06						
128K	37.54	34.19	33.26	33.36	31.84							
256K	39.85	36.51	36.74	35.37								
512K	39.78	37.81	36.54									
1M	39.53	37.62										
2M	39.25											
4M	38.86 - (No z-way merge is necessary for this block size)											

A slightly more complicated picture of the role m and z emerges from Table 4, which displays the times required to sort 4M doubles on the IBM SP-2 using a single thread as a function of m and z. Again, performance suffers dramatically when the block size reaches 1MB (128K eight byte double precision numbers), which is the limit of the secondary cache on this platform. But consider the data for a given block size - say 256. The execution time drops as we move from $z = 2$ to $z = 128$. This is reasonable since we require 14 rounds of 2-way merge, 7 rounds of 4-way merge, 5 rounds of 8-way merge, 4 rounds of 16-way, 3 rounds of 32-way merge and 64-way merge, and only 2 rounds of 128-way merge, and each round of z-way merge is obviously another round where all the input elements must be brought in from main memory. We would then expect that moving from $z = 128$ to $z = 16384$ would have little effect on the execution time since it does nothing to reduce the memory requirements, but this turns out not to be the case. The explanation lies in recalling that, unlike the Exemplar, the SP-2 has both a primary and a secondary cache. An efficient implementation of the z-way merge in Step (1B) would fill this 16 KB 4-way set associative primary cache with the entire tree of losers (z 12 byte records) plus a cache line (32 bytes) from each of the z sequences being merged. For $z = 256$, this primary cache is essentially filled, and cache misses to secondary cache become an issue. Finally, note the difference between the optimal sorting time of 10.24 seconds for $m = 256$ and $z = 128$ with the time of 28.29 seconds required to sort using only binary merge sort. Here, reducing memory access by a combination of block sorting and z-way merging improved the performance by 64%. Again, such results strongly support the attention that we place in this algorithm on the number of contiguous memory accesses.

Table 4. Time (in seconds) required on the IBM SP-2 to sort 4M doubles using a single thread as a function of M and z.

Block Size	**Denomination of z-Way Merge**														
	2	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**
128	23.72	15.98	13.40	12.13	11.17	11.34	11.55	10.34	10.85	12.06	13.83	15.43	17.18	20.81	20.03
256	22.71	14.86	13.45	12.32	11.20	11.51	10.24	10.59	11.26	12.27	13.71	15.37	18.37	17.12	
512	21.70	15.07	13.66	12.51	11.39	11.71	10.47	11.02	11.58	12.27	13.71	15.33	15.34		
1K	20.74	14.10	12.68	11.43	11.68	10.50	10.82	11.13	11.37	12.19	13.64	13.60			
2K	20.05	14.62	13.00	11.78	12.13	11.00	11.31	11.42	11.67	12.44	12.23				
4K	20.45	14.93	14.49	13.38	12.31	12.54	12.64	12.88	13.04	12.08					
8K	19.84	15.58	14.00	13.91	12.84	13.10	13.26	13.40	12.68						
16K	19.51	15.18	14.58	13.46	13.61	13.82	13.94	12.65							
32K	19.70	16.62	15.90	14.71	15.18	15.22	13.99								
64K	21.13	17.68	17.17	17.26	17.79	16.66									
128K	24.23	21.81	20.73	21.04	19.89										
256K	26.45	24.42	24.62	23.55											
512K	27.95	27.00	26.00												
1M	28.10	27.26													
2M	28.16														
4M	28.29 - (No z-way merge is necessary for this block size)														

The graphs in Figs. 4 and 5 examines the scalability of our sorting algorithm as a function of the number of threads, for a variety of problem sizes. Bearing in mind that these graphs are log-log plots, they show that for large enough inputs, the execution time decreases as we increase the number of threads p, which is the expectation of our model. For smaller inputs on the HP-Convex Exemplar, this inverse relationship between the execution time and the number of threads deteriorates when we move to 16 threads. This explanation for this problem may lie in the fact that when we moved to 16 threads on this platform, the data suddenly became very erratic, perhaps because some threads now had to compete with operating system processes for access to one of the 16 processors.

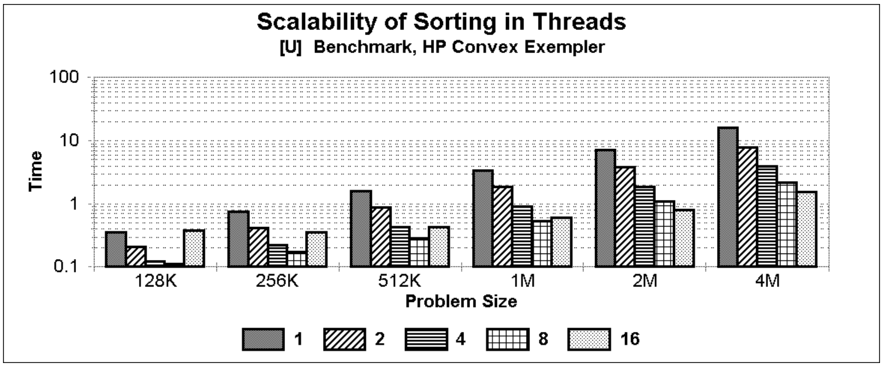


Fig. 4. Scalability of our generalized sorting algorithm on the HP-Convex Exemplar with respect to the number of threads, for differing problem sizes.

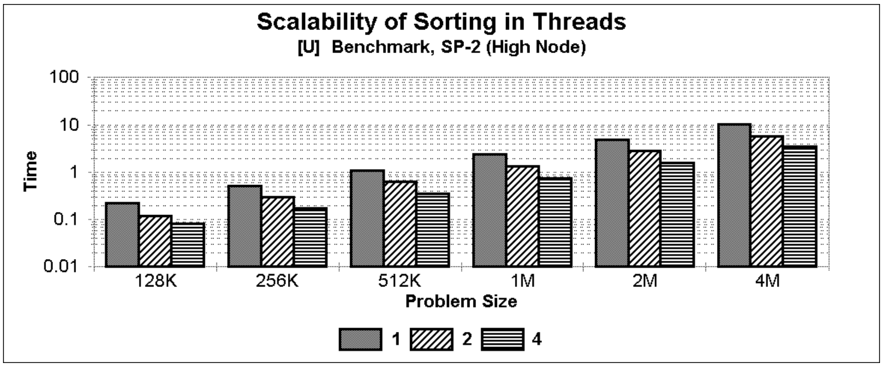


Fig. 5. Scalability of our generalized sorting algorithm on the IBM SP-2 with respect to the number of threads, for differing problem sizes.

References

1. A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A Model for Heirarchical Memory. In *Proceedings of the 19th Annual ACM Symposium of Theory of Computing*, pages 305–314, May 1987.
2. A. Aggarwal, A. Chandra, and M. Snir. Heirarchical Memory with Block Transfer. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, October 1987.
3. A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31:1116–1127, 1988.
4. B. Alpern, L. Carter, E. Feig, and T. Selker. The Uniform Memory Hierarchy Model of Compuatation. *Algorithmica*, 12:72–109, 1994.
5. R. Anderson and G. Miller. Deterministic Parallel List Ranking. In *Proceedings Third Aegean Workshop on Computing, AWOC 88*, pages 81–90, Corfu, Greece, June/July 1988. Springer-Verlag.
6. G.E. Blelloch, P.B. Gibbons, Y. Matias, and M. Zagha. Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):943–958, 1997.
7. P.B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM Journal on Computing*, 1997. To appear.
8. D.R. Helman and J. JáJá. Prefix Computations on Symmetric Multiprocessors. Technical Report CS-TR-3915 and UMIACS-TR-98-38, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1998.
9. D.R. Helman and J. JáJá. Sorting on Clusters of SMPs. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, Florida, April 1998.
10. D.R. Helman, J. JáJá, and D.A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. *ACM Journal of Experimental Algorithmics*, 3(4):1–24, 1998.
11. D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.
12. A. LaMarca and R. Ladner. The Influence of Caches on the Performance of Sorting. In *roceedings of the Eighth Annual Symposium on Discrete Algorithms*, pages 370–377, January 1997.
13. M. Reid-Miller. List Ranking and List Scan on the Cray C90. *Journal of the Computer and System Sciences*, 53:344–356, 1996.
14. H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.
15. P. Varman, B. Iyer, and S. scheufler. A Multiprocessor Algorithm for Merging Multiple Sorted Lists. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 22–26.
16. Uzi Vishkin. Randomized Speed-Ups in Parallel Computation. In *Proceedings of the Sixteenth ACM Symposium on Theory of Computing*, pages 230–239, Washington, D.C., 1984.
17. J. Vitter and E. Shriver. Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, 12:110–147, 1994.
18. J.C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, Ithica, NY, 1979.

Circular Drawings of Biconnected Graphs[⋆]

Janet M. Six and Ioannis G. Tollis

CAD & Visualization Lab
Department of Computer Science
The University of Texas at Dallas
P.O. Box 830688, EC 31
Richardson, TX 75083-0688
{janet,tollis}@utdallas.edu

Abstract. Many applications which would benefit from an accompanying circular graph drawing include tools which manipulate telecommunication, computer, and social networks. Previous research has produced solutions which are visually complex with respect to the number of crossings. In this paper we focus our attention on developing better and more efficient circular drawing algorithms. In particular we present an $O(m^2)$ algorithm which lays out a biconnected graph onto a single embedding circle. Furthermore, we can guarantee that if a zero crossing circular embedding exists for an input graph, then our algorithm will find it. Also, the results of extensive experiments conducted over a set of 10,328 biconnected graphs and show our technique to perform significantly better than the current technology.

1 Introduction

Graphs are used to represent many kinds of information structures: computer, telecommunication, and social networks, entity-relationship diagrams, data flow charts, resource allocation maps, and much more. *Graph Drawing* researchers develop techniques which embed graphs onto a two or three-dimensional surface where nodes are represented by circles or polygons and edges by polygonal chains of line segments. The input to a graph drawing algorithm is a graph, $G = (V, E)$, where V is the set of n nodes and E is the set of m edges, while the output is a set of layout coordinates for each graph element. See [2,3] for a comprehensive annotated bibliography and introduction to the area.

The visualizations produced by graph drawing algorithms can be very powerful. These drawings free the user from having to create a mental image of the information structure thereby aiding the user in the design and analysis of that structure. Much work has been done in graph drawing and there now exist many techniques to produce nice visualizations of information structures. These visualizations can take form in one of many standards: e.g., circular, force-directed, hierarchical, or orthogonal [2,3].

⋆ Research supported in part by NIST, Advanced Technology Program grant number 70NANB5H1162 and by the Texas Advanced Research Program under grant number 009741-040.

A *circular graph drawing* (see Figure 1 for an example) is an embedding of a graph with the following characteristics:

- The graph is partitioned into clusters
- The nodes of each cluster are placed onto the circumference of an individual embedding circle
- Each edge is drawn with a straight line

There are many applications which would be strengthened by an accompanying circular graph drawing. An important application of circular graph drawing is to visualize various types of networks. For example, our drawing technique could be added to tools which manipulate telecommunication, computer, and social networks to show clustered views of those information structures. Emphasizing natural group structures within the topology of the network is vital to pin-point strengths and weaknesses within that design. It is essential that the number of crossings within each cluster remain low and that nodes have good proximity to their neighbors. Researchers have produced several circular drawing techniques [1,7,9,15], some of which have been integrated into commercial tools. However, the resulting drawings are visually complex with respect to the number of crossings. This paper is focused on finding better and more efficient circular drawing techniques.

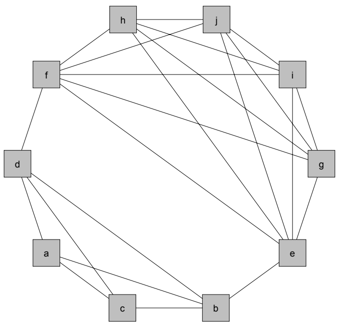


Fig. 1. A circular drawing as produced by our algorithm.

The remainder of this paper is organized as follows: Section 2 discusses previous work in this area. In Section 3 we present an $O(m^2)$ time complexity algorithm for the circular layout of biconnected graphs. Interesting properties of outerplanar graphs and circular drawings are also presented in Section 3. In Section 4 we discuss the implementation of the algorithm of Section 3 and experimental results. Conclusions are presented in Section 5.

2 Previous Work

2.1 Complexity of the Circular Graph Drawing Problem

Intuitively the Circular Graph Drawing problem is very hard since algorithms attempt to place the nodes of each cluster onto an embedding circle such that the number of edge crossings is minimum. The general problem of placing nodes such that the number of edge crossings is minimum is the well known NP-Complete CROSSING NUMBER problem. However, the more restricted problem of finding a minimum crossing embedding such that all the nodes are placed onto the circumference of a circle and all edges are represented with straight lines is also NP-Complete as proved in [10]. The authors show the NP-Completeness of the Circular Graph Drawing problem by giving a polynomial transformation from the NP-Complete MODIFIED OPTIMAL LINEAR ARRANGEMENT problem.

2.2 Previous Circular Drawing Techniques

Kar, Madden, and Gilbert present in [7] a technique and tool to aid decision making involving complex issues within network management. Recognizing that a clustered view of a network can be quite helpful to its design and maintenance, the authors build a system which first partitions the network into clusters, places the clusters onto the main embedding circle and then sets the coordinates of individual nodes. Finally a heuristic approach is used to minimize the number of crossings. As discussed in [5], an advanced version of this technique has been implemented as part of Tom Sawyer Software's successful Graph Layout Toolkit (GLT).

Tollis and Xia introduced several linear time algorithms for the visualization of survivable telecommunication networks in [15]. Given the ring covers of a network, these algorithms create circular drawings such that the survivability of the network is very visible. Techniques were presented for outside(inside) drawings such that the rings are placed outside(inside) a root circle. An additional linear time algorithm produces drawings which are a combination of outside and inside drawings. This type of flexibility in a tool allows each network designer to choose the best technique given the exact application.

Citing a need for graph abstraction and reduction of today's large information structures, Brandenburg describes an approach to draw a path (or cycle) of cliques in [1]. This $O(n^3)$ algorithm creates a two-level abstraction of the given graph giving the ability to project a clique on each node of the abstracted graph.

Circular drawing techniques are not limited to telecommunication and computer network applications by any means. InFlow [9] is a tool to visualize human networks and produces diagrams and statistical summaries to pinpoint the strengths and weaknesses within an organization. The usually unvisualized concepts of self-organization, emergent structures, knowledge exchange, and network dynamics are captured by the drawings of InFlow. Resource bottlenecks, unexpected work flows, and gaps within the organization are clearly shown in these circular drawings.

3 The Algorithm

In this section, we present an algorithm for obtaining a circular drawing of any biconnected graph such that all the nodes are placed onto the circumference of a single embedding circle. The experimental study presented in Section 4 shows our algorithm to be a significant improvement over the current state of the art.

In order to find a circular drawing with a lower number of crossings than previous techniques, we have developed an algorithm which tends to place edges toward the outside of the embedding circle. This characteristic means that there are not many edges in the middle of the drawing to be crossed and also that nodes are placed near their neighbors. In fact, this algorithm tries to maximize the number of edges appearing on the circumference of the embedding circle. This is achieved by selectively removing some edges and then building a DFS-based node ordering of the resulting graph. The number of crossings is then further reduced by locally optimizing the placement of each node.

Before describing the algorithm, we introduce some definitions. A *pair edge* is incident to two nodes which share at least one neighbor, see Figure 2. Nodes v and w are *paired* by u. And u is said to *establish* the pair edge vw. A *triangulation* edge is a new pair edge which is placed into the graph.

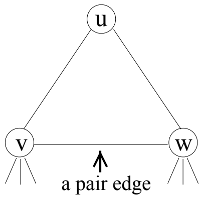


Fig. 2. Example of a pair edge.

Define a *wave front node* to be adjacent to the last node processed, see Figure 3. A *wave center node* is adjacent to some node which has already been processed.

3.1 Phase 1 - Node Ordering

In order to selectively remove some edges, Algorithm CIRCULAR - Phase 1 visits the nodes in a wave-like fashion. It first starts at a lowest degree node and continues to visit wave front and wave center nodes if they have lowest degree. The flow of the node traversal is similar to that of a local breadth-first search (BFS). If none of the current wave front or wave center nodes are of lowest degree then some lowest degree node is chosen. The wave-like node traversal begins again from this newly chosen node and will continue from this node and the previous wave front and wave center nodes. The process continues until all but the last three nodes are processed. The sequence in which the nodes

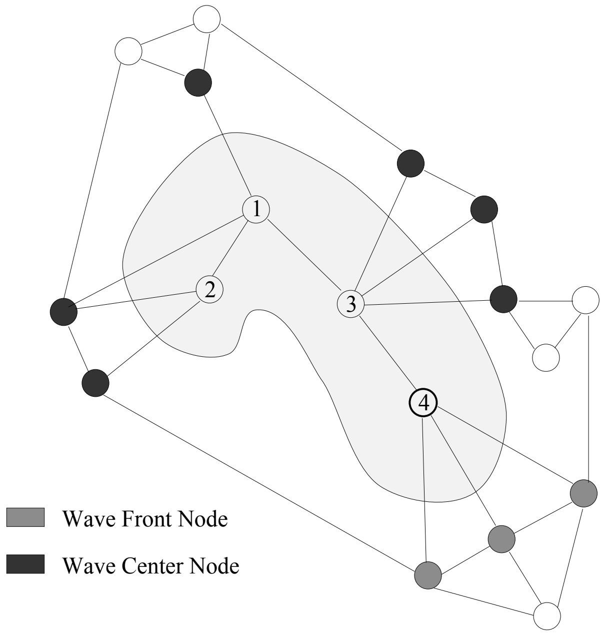


Fig. 3. Examples of wave front and wave center nodes. The shaded region includes those nodes which have already been processed. The node labeled **4** is the most recently processed.

are visited is like that of the execution of several breadth-first searches run in parallel on a single processor machine. With the additional requirements that each BFS is performed on the same graph and yet initialized at unique nodes. Hence the algorithm processes nodes from one section of the graph and can then go to another section of the graph.

Each time a node is visited, a list of pair edges is built. If a sufficient number of pair edges do not exist within the graph, triangulation edges are inserted into the structure. With the ensuing removal of that node, it is inherently represented by those newly found pair edges: an absorption of a node and its incident edges. It is this selective absorption that causes the behavior of edge placement towards the perimeter of the embedding circle.

Subsequently our edge removal, CIRCULAR - Phase 1 proceeds to build an ordering of the reduced graph. A traditional depth-first search (DFS) is performed and then the nodes in a longest path of the DFS tree are placed around the embedding circle. Finally, the remaining nodes are nicely merged into the ordering.

Algorithm 31 CIRCULAR - Phase 1
Input: *A biconnected graph, $G = (V, E)$*
Output: *An embedding such that each node of G lies on the perimeter of a single embedding circle*

```
1.  Bucket Sort the nodes by ascending degree into a table, T
2.  Set counter to 1
3.  While counter $< n - 3$
4.          Choose currentNode to be some lowest degree node with the following
            priority:
            a wave front node, a wave center node, some lowest degree node
5.          Find and place all pair edges which are established by currentNode
             into removalList
6.          If the number of pair edges is less than degree(currentNode) - 1 then
7.                  insert triangulation edges between pairs of currentNode's
                    neighbors such that each neighbor node is incident to at
                    least one of the new removalList edges and there are
                    degree(currentNode) - 1 pair edges
8.          Update the location of currentNode's neighbors in T
9.          Remove currentNode from G
10.         Increment counter by 1
11. Restore G to its original topology
12. Remove the edges in removalList from G
13. Perform a traditional DFS on G
14. Place the longest path of the resulting DFS tree on the embedding circle
15. If there are any nodes which have not been placed then
16.         place the remaining nodes into the embedding order with the
            following priority:
            between two neighbors, next to one neighbor, next to zero neighbors
```

The time complexity of CIRCULAR - Phase 1 is $O(n * m)$ where n is the number of nodes and m is the number of edges. Steps 1, 3, 4, 6, 9, and 10 require $O(n)$ time over the entire execution of the algorithm. Likewise, Steps 7, 8, 11, 12, 13, 14, and 16 require $O(m)$ time. Step 8 requires $\Sigma(degree(v_i))$ repetitions during each iteration which is $\Theta(m)$ time. Likewise, Step 16 is takes $O(m)$ time since the algorithm reviews at most $degree(Node)$ positions for a possible placement. Finding the pair edges in Step 5 requires $O(n * m)$ time. However if the degree of nodes is bounded by some constant, this step requires $O(m)$ time thereby making this phase linear. It is important to state a bound on the number of triangulation edges which can be added to G. Since at each iteration of Step 6, at most $O(degree(currentNode))$ triangulation edges are added and this set is repeated $n - 3$ times the total number of edges added is $O(m)$.

The first phase of this algorithm produces a circular ordering of the nodes. If the given biconnected graph is outerplanar then this phase will find a circular embedding such that no two edges cross. In fact, this phase has been inspired by the technique for recognizing outerplanar graphs presented in [11]. A graph, G, is *outerplanar* if and only if G can be drawn on the plane such that all nodes lie on the boundary of a single face and no two edges cross.

By the definition of outerplanar graphs we know that there exists a plane circular drawing for any outerplanar graph. Also, by that same definition we know that a graph which is not outerplanar does not admit a plane circular drawing. In fact, the set of biconnected graphs which may be drawn in a circular fashion without any crossings is exactly the set of biconnected outerplanar graphs. The requirement of placing all nodes on some embedding circle is equivalent to placing all nodes on a single face (say the external face) of some embedding. So if a graph can be drawn in a circular fashion without edge crossings, it must be outerplanar. Furthermore, if a zero-crossing embedding exists for a biconnected graph, G, then that embedding can be found by CIRCULAR - Phase 1.

Theorem 1. *Algorithm CIRCULAR - Phase 1 produces a circular drawing without crossings of any n-vertex outerplanar graph in $O(n)$ time.*

Theorem 2. *There exists only one relative ordering of the nodes in an outerplanar graph, G, such that the embedding of G with the nodes in that order around the embedding circle is plane.*

3.2 Phase 2 - Further Crossing Reduction

There are significant issues with crossing reduction. The brute force approach to finding the global minimum number of crossings is to try all relevant permutations of the nodes ($(n-1)!$ permutations are pertinent since we are working with a circular as opposed to a linear order) and choose the one which has the least number of crossings. This approach finds the global minimum number of crossings, but takes exponential time, and hence is not practical. We present a heuristic which takes advantage of certain properties of the graph and finds a good local minimum. The initial configuration of the nodes produces a low number of crossings, which is then further reduced to some local minimum with a monotonic crossing reduction technique. This phase visits each node and queries whether or not crossings can be reduced by the movement of that node next to one of its neighbors.

Algorithm 32 CIRCULAR - Phase 2
Input: *The embedding of G produced by CIRCULAR - Phase 1*
Output: *An embedding of G with fewer or equal number of crossings*

```
1. currentCrossings = current number of crossings in the embedding
2. For i = 1 to 10
3.         For each node, currentNode, in G
```

```
4.                  List_1 = embedding circle positions which lie between two of
                    currentNode's neighbors
5.                  If List_1 is empty
6.                          List_2 = embedding circle positions which lie next to
                            one of currentNode's neighbors
7.                          PositionList = List_2
8.                  else
9.                          PositionList = List_1
10.                 For each location in PositionList
11.                         place currentNode at this location
12.                         newCrossings = the new number of crossings
13.                         If newCrossings < currentCrossings then
14.                                 currentCrossings = newCrossings
15.                         Else
16.                                 Place currentNode back into its previous
                                    position
17.         If no improvement was made during this iteration, STOP
```

The time complexity of CIRCULAR - Phase 2 is $O(m^2)$. This order is dominated by the required time for counting the number of crossings (steps 1 and 12). It is vitally important to the time efficiency of this phase that the number of crossings be counted in a sensible fashion.

Lemma 1. *An $O(m + \chi)$ time complexity algorithm exists to count the total number of edge crossings in a single circle embedding where m is the number of edges and χ is the number of crossings.*

Consider the edges e_i and e_j of Figure 4. The edge e_i can cross e_j only if one endpoint, v, but not both, of e_j appear between the two endpoints, u and w, of e_i. In this case, e_j is called an *open* edge with respect to the arc uvw. See Figure 4.

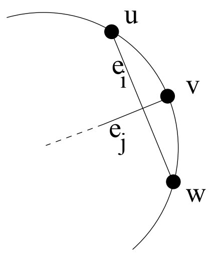


Fig. 4. Illustration of an open edge with respect to the arc uvw.

So, if we order the edges as they are encountered around the embedding circle and visit their endpoints in that order, we can determine the total number of edge crossings. Each time we encounter the second endpoint of an edge, we count the number of open edges with a higher order number. Step 1 of Algorithm 3.2 can be accomplished with the following algorithm. Although our problem is one dimensional, this technique has some similarities to the line segment intersection algorithm presented in [12].

Algorithm 33 CountAllCrossings
Input: *A single circle embedding of G*
Output: *The number of edge crossings within the input embedding*

```
1. Order the edges as they are encountered around the circle in a clockwise order
2. For each edge endpoint, $p_i$, of edge $e_i$, do
3.         If $p_i$ is the first endpoint of edge $e_i$
4.                 append $e_i$ to openEdgeList
5.         Else
6.                 count the number of open edges with higher order
7.                 remove $e_i$ from openEdgeList
```

Step 1 takes $O(m)$ time. Over the course of the entire algorithm, Steps 4 and 6 require constant time. Likewise, Step 7 requires $\sum_{i=1}^{2m} \chi_i = O(\chi)$ time, where χ_i is the number of edge crossings caused by the edge e_i and χ is the total number of edge crossings in the embedding. Therefore, Algorithm CountAllCrossings has $O(m + \chi)$ time complexity.

Since we start with the overall number of crossings and move one node at a time, it is only necessary to count the number of crossings caused by the incident edges of the current node, v. During each iteration of the crossing reduction, the number of crossings in the entire drawing is equal to the following formula:

$$New\ Number\ of\ Crossings = Old\ Number\ of\ Crossings - v_x + v'_x$$

where, v_x = Number of crossings caused by v in the old location,
and v'_x = Number of crossings caused by v in the new location.

Because we already know the old number of crossings, finding the new number of crossings is dominated by the time to find v_x and v'_x.

Lemma 2. *An $O(m)$ time complexity algorithm exists to count the number of edge crossings gained or lost by moving a node, v, within a single circle embedding.*

Following the same principles as discussed with the previous theorem, this algorithm counts the number of edge crossings gained or lost by moving node v within a single circle embedding. Any change in the edge crossings will occur with edges that have an endpoint in the arc between the old and new positions of v. See Figure 5. The pertinent edges are those which have one endpoint within the aforementioned arc. These pertinent edges are visited in order from the old towards the new position of v. A counter, *ctr*, holds the number of open edges in the secant (not including the open edges incident to v). Each time that an endpoint of an edge incident to v is encountered, the number of crossings is increased by the value in *ctr*. At the conclusion of this process, the number of crossings caused by v in the old position is known. This process is repeated again in the opposite direction after moving v in order to find the number of crossings caused by v in its new position.

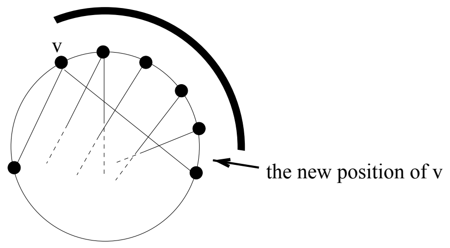


Fig. 5. The arc created by moving node v to the position denoted with the arrow. The pertinent edges of the arc are shown.

Algorithm 34 CountSingleNodeCrossings
Input: *A single circle embedding of G, a node, v, and a new position for v*
Output: *The number of edge crossings caused by v*

```
1.  $ctr = 0$
2.  $numberOfCrossings = 0$
3.  Order the pertinent edge endpoints
4.  For each edge endpoint, $p_i$, of edge $e_i$ do
5.          If $e_i$ is incident to $v$
6.                  If $p_i$ is an endpoint
7.                          increment the numberOfCrossings by ctr
8.          Else
9.                  If $p_i$ is an endpoint
10.                         decrement ctr by 1
11.                 Else
12.                         increment ctr by 1
13. $OldNumberSingleNodeCrossings = numberOfCrossings$
14. $ctr = 0$
15. $numberOfCrossings = 0$
16. Move $v$ to its new position
17. Repeat the above process in the opposite direction
18. $NewNumberSingleNodeCrossings = numberOfCrossings$
```

Algorithm CountSingleNodeCrossings clearly has $O(m)$ time complexity. However, if Phase 2 is swapping the placement of two nodes which are next to each other, then CountSingleNodeCrossings only takes $O(n)$ time. See [13] for more details.

The time complexity of CIRCULAR - Phase 2 is as follows: Step 1 requires $O(m + \chi)$ time to find the total number of crossings. Steps 3, 5, 7, 8, 9, and 17 require $O(n)$ time. Steps 4, 6, 10, 11, 13, 14, 15, and 16 require $O(m)$ time. Step 12 takes $O(m^2)$ time. Therefore, Phase 2 has time complexity $O(m^2)$. A large set of experiments have shown that the loop of Step 2 needs to be iterated at most 9 times. In fact, the vast majority of drawings converge within the first two iterations.

4 Implementation and Experiments

We have implemented our technique for building circular drawings of biconnected graphs in C++ (GNU C++ version 2.7.2.1) on a SPARC 5 running SunOS 4.1.3. The code runs on top of the Tom Sawyer Software Graph Layout Toolkit (GLT) version 2.3.1. See [13] for further implementation details.

4.1 Implementation of CIRCULAR - Phase 1

During Step 4, the algorithm chooses a node of lowest degree with the following priority: a wave front node, a wave center node, or some lowest degree node. An efficient way to execute this is to initially sort the nodes by degree and keep nodes in a table of lists which reflect those categories. A bucket sort is initially used to place each node into its respective category. In order to keep the table updated, when a node, v, is processed, we simply reinsert each neighbor of v into the front of its respective degree list during each iteration. This way the nodes are retrieved in the desired priority: neighbor, previous neighbor and lowest degree node. See Figure 6.

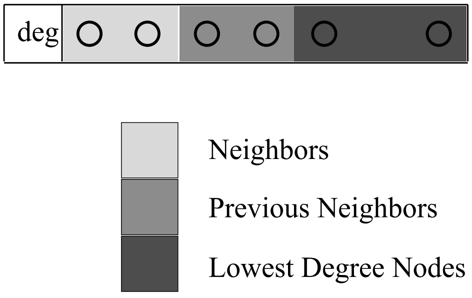


Fig. 6. The construction of each degree list within the node table

During Step 5 if currentNode has degree two, then the pair edge is the edge induced by currentNode. A triangulation edge is nominally added to the graph if the induced edge does not already exist. Remember that triangulation edges are always removed from the graph at the end of Phase 1.

For higher degree nodes, we include the edges which exist between any two adjacent nodes. Our technique adds triangulation edges incident to the nodes adjacent to a higher degree removal node such that the total number of removalList edges is equal to degree(removalNode) - 1. We are careful not to unnecessarily increase the degrees of any node in the current graph. Triangulation edges are added with the following priority: first make each neighbor of currentNode incident to at least one removalList edge. Then if the number of new removalList edges is still not degree - 1, then add the minimum number of edges between nodes of low degree that are not already adjacent. See Figure 7.

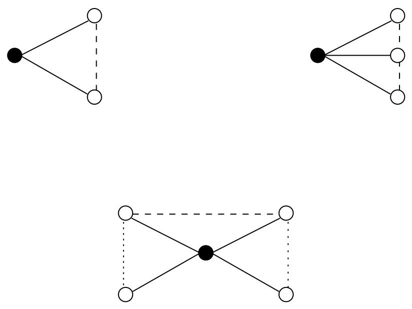


Fig. 7. Different scenarios for the addition of adding triangulation edges

The top left drawing shows the example of a degree two currentNode. The edge induced by currentNode or the triangulation edge added if one does not exist, is the single edge added to removalList. This edge is shown with a dashed line. The top right drawing shows a similar example for a degree three current node. The bottom picture shows an example of an important scenario. This currentNode (with black fill) has degree four and there are no pre-existing edges among the neighbors. First we make each neighbor incident to one removalList edge. These are the two vertical, dotted lines. One more triangulation edge must be added so that the number of new removalList edges is equal to $4 - 1 = 3$. Phase 1 looks at the degrees of the neighbor nodes and places an edge between two lowest degree nodes which are not already adjacent. This is shown with the horizontal, dashed line.

During Step 11, the algorithm performs a depth-first search. Given a graph, a traditional DFS will find a tree. Since we want to place the nodes of the given graph around a single circle, we need not find a tree, but some linear order of the given nodes. Phase 1 removes some edges from the graph in order to get a spanning subgraph. In order to have a high connectivity rate between neighbors on the embedding circle, we perform a modified DFS on the reduced graph. First we conduct a traditional DFS recording the distance of the current node from its most distant descendant in the resulting DFS tree. See Figure 8. Then we place the nodes from the longest path within that DFS tree onto the embedding circle. See Figure 9. The longest path is determined by first finding the node in the DFS tree which has the two most distant descendants. The rest of the longest path is found by following the paths to those most distant descendants until a leaf node is reached. Finally we merge in the remaining DFS tree branches which reach the other nodes. This step places nodes with the following priority: between two neighbors, next to one neighbor and next to zero neighbors.

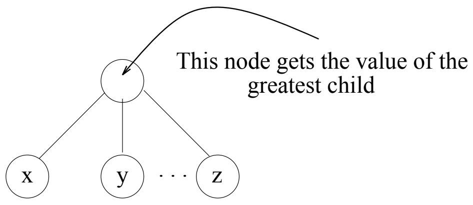


Fig. 8. The assignment of values to each node during the modified DFS

4.2 Implementation of CIRCULAR - Phase 2

If the input graph is outerplanar, the order of nodes around the embedding circle will always be the one which results in a plane drawing. But if there are crossings then it may be possible to further reduce the number of crossings by reinserting nodes into a better position on the embedding circle.

As noted in the time complexity analysis of Phase 2, the order is dominated by the time required for counting the number of crossings. Therefore it is vitally important to the time efficiency of this phase that the number of crossings be counted in an efficient manner.

In order to lower the average time cost of counting crossings in the drawing, we ignore all edges which lie on the perimeter of the embedding circle. These edges cannot possibly cause crossings. Also, in the step which determines the number of crossings caused by a single node, either the clockwise or counter-clockwise direction is first chosen dependent on which has the shorter arc.

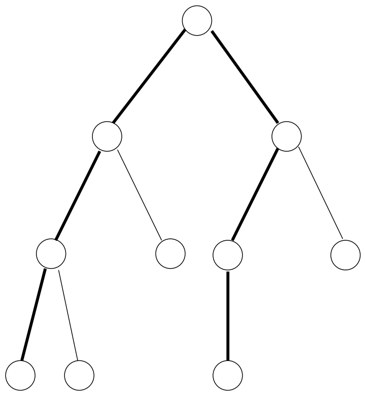


Fig. 9. A DFS tree with the edges of the longest path designated by thick lines. The longest path does not necessarily go through the root of the DFS tree as it does in this example

4.3 Experimental Study

The set of input graphs for our experiments included 10,328 biconnected components of the Rome graphs [4] which between 10 and 80 nodes. The number of edge crossings is measured for both phases of CIRCULAR and also for the GLT. As shown in the plot of Figure 10, our technique produces significantly fewer crossings on average than the technique implemented in the GLT. Even the drawings of CIRCULAR - Phase 1 have significantly fewer crossings. And as the plot shows, CIRCULAR - Phase 2 effectively reduces the number of edge crossings even further. The percentage improvement between CIRCULAR and GLT averages is a very good 33%. In fact, this percent improvement is never less than 20%. Sample drawings as produced by both GLT and CIRCULAR are shown in Figures 11 and 12.

This technique can be extended to layout trees and other non-biconnected components onto a single embedding circle. Also we are working on an algorithm which embeds not necessarily biconnected components onto separate embedding circles while placing the superstructure onto a larger, all-encompassing embedding circle.

5 Conclusions

Visualizations of networks which show the inherent strengths and weaknesses of structures with clustered views would be advantageous additions to many design tools. Some techniques for circular graph drawing have been previously presented, but the resulting embeddings are visually complicated by the number of crossings.

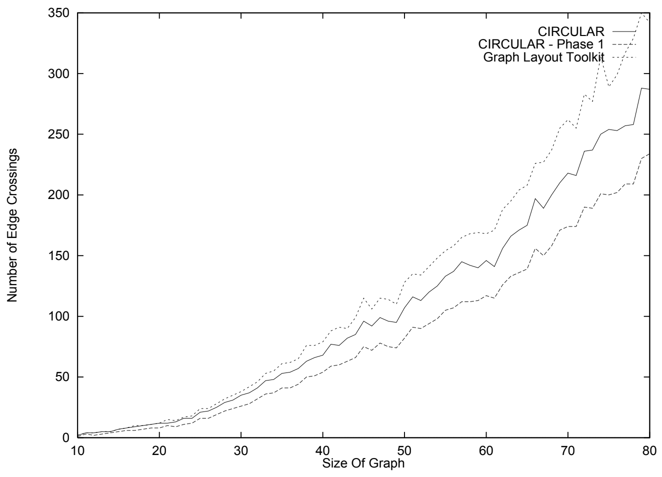


Fig. 10. This plot shows the average number of edge crossings produced by CIRCULAR and the Graph Layout Toolkit over 10,543 biconnected components of the Rome graphs.

We have introduced an $O(m^2)$ algorithm for drawing circular visualizations of biconnected graphs onto a single embedding circle. Not only is this technique efficient, it also produces a plane drawing of the biconnected graph if such exists. Extensive experiments show that our technique significantly outperforms the current state of technology.

References

1. F. Brandenburg, Graph Clustering 1: Cycles of Cliques, *Proc. GD '97, Rome, Italy, Lecture Notes in Computer Science 1353, Springer-Verlag*, pp. 158-168.
2. G. Di Battista, P. Eades, R. Tamassia and I. Tollis, Algorithms for Drawing Graphs: An Annotated Bibliography, *Computational Geometry: Theory and Applications*, 4(5), 1994, pp. 235-282. Also available at `http://www.utdallas.edu/~tollis`.
3. G. Di Battista, P. Eades, R. Tamassia and I. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice-Hall, Englewood Cliffs, NJ, 1999.
4. G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, F. Vargiu and L. Vismara, An Experimental Comparison of Four Graph Drawing Algorithms, *Computational Geometry: Theory and Applications*, 7(5-6), pp.303-25. Also available at `http://www.cs.brown.edu/people/rt`.
5. U. Doğrusöz, B. Madden and P. Madden, Circular Layout in the Graph Layout Toolkit, *Proc. GD '96, Berkeley, California, Lecture Notes in Computer Science 1190, Springer-Verlag*, pp. 92-100.
6. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
7. G. Kar, B. Madden and R. Gilbert, Heuristic Layout Algorithms for Network Presentation Services, *IEEE Network*, November 1988, pp. 29-36.

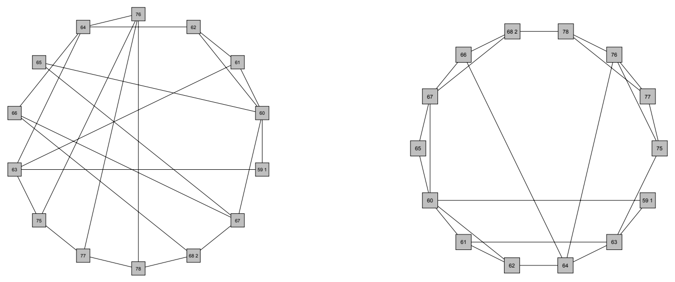


Fig. 11. The drawing on the left is produced by Tom Sawyer Software's Graph Layout Toolkit. The drawing on the right is of the same graph and is produced by CIRCULAR. The CIRCULAR drawing has 75% fewer crossings than the GLT drawing.

8. A. Kershenbaum, *Telecommunications Network Design Algorithms*, McGraw-Hill, 1993.
9. V. Krebs, Visualizing Human Networks, *Release 1.0: Esther Dyson's Monthly Report*, February 12, 1996, pp. 1-25.
10. S. Masuda, T. Kashiwabara, K. Nakajima and T. Fujisawa, On the NP-Completeness of a Computer Network Layout Problem, *Proc. IEEE 1987 International Symposium on Circuits and Systems, Philadelphia, PA*, 1987, pp.292-295.
11. S. Mitchell, Linear Algorithms to Recognize Outerplanar and Maximal Outerplanar Graphs, *Information Processing Letters*, 9(5), 1979, pp. 229-232.
12. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.
13. J. M. Six and I. G. Tollis, Algorithms for Drawing Circular Visualizations of Networks, Manuscript, 1999.
14. I. G. Tollis, Strategic Directions in Computing Research: Working Group on Computational Geometry: Graph Drawing and Information Visualization, *ACM Computing Surveys*, 28A(4), December 1996. Also available at `http://www.utdallas.edu/~tollis/SDCR96/TollisGeometry/`.
15. I. G. Tollis and C. Xia, Drawing Telecommunication Networks, *Proc. GD '94, Princeton. New Jersey, Lecture Notes in Computer Science 894, Springer-Verlag*, 1994, pp. 206-217.
16. M. Yannakakis, Edge-Deletion Problems, *SIAM J. Computing*, 10(2), May 1981, pp.297-309.

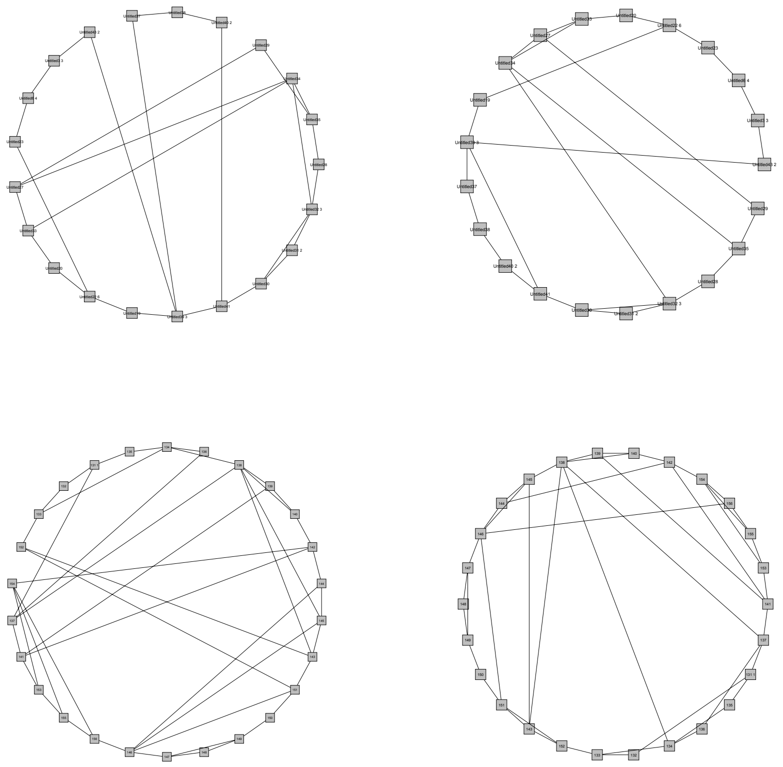


Fig. 12. The drawings on the left are produced by Tom Sawyer Software's Graph Layout Toolkit. The drawings on the right are of the same graphs and are produced by CIRCULAR. The CIRCULAR drawings have 53% and 55% fewer crossings than the respective GLT drawings.

Heuristics and Experimental Design for Bigraph Crossing Number Minimization

Matthias Stallmann, Franc Brglez*, and Debabrata Ghosh*

Department of Computer Science, PO Box 8206
N.C. State University
Raleigh, NC 27695-8206 USA
`http://www.cbl.ncsu.edu/experiments/`

Abstract. The *bigraph crossing problem*, embedding the two vertex sets of a bipartite graph $G = (V_0, V_1, E)$ along two parallel lines so that edge crossings are minimized, has application to circuit layout and graph drawing. We consider the case where both V_0 and V_1 can be permuted arbitrarily — both this and the case where the order of one vertex set is fixed are NP-hard. Two new heuristics that perform well on sparse graphs such as occur in circuit layout problems are presented. The new heuristics outperform existing heuristics on graph classes that range from application-specific to random. Our experimental design methodology ensures that differences in performance are statistically significant and not the result of minor variations in graph structure or input order.
Keywords. Crossing number minimization in graphs, graph equivalence classes, design of experiments

1 Introduction

The minimization of the crossing number in a specific graph embedding has often been motivated by factors such as (1) improving the appearance of a graph drawing [4,7,10,23,29], (2) reducing the wiring congestion and crosstalk in VLSI circuits, which in turn may reduce the total wire length and the layout area [19,22,25,26]. This paper is about *bigraph crossing* defined as follows for any bipartite graph (bigraph) $G = (V_0, V_1, E)$ [16]: Let G be embedded in the plane so that the nodes in V_i occupy distinct positions on the line $y = i$ and the edges are straight lines. For a specific embedding $f(G)$, the *crossing number* $C_f(G)$

* This research has been supported by contracts from the Semiconductor Research Corporation (94–DJ–553), SEMATECH (94–DJ–800), DARPA/ARO (P–3316–EL/DAAH04–94–G–2080), and (DAAG55-97-1-0345), and a grant from Semiconductor Research Corporation.

is the number of line intersections induced by f. This depends only on the permutation of V_i along $y = i$ and not on specific x-coordinates. The (bigraph) crossing number $C(G) = \min_f C_f(G)$. Garey and Johnson [11] proved that it is NP-hard to compute $C(G)$. Detection of a *biplanar* graph, a bigraph G for which $C(G) = 0$, however, is easy [15].

Previous work on bigraph crossing has focused primarily on the *fixed-layer* (version of the) problem, namely computing $C(G)$ subject to the constraint that the permutation of V_0 on $y = 0$ must stay fixed. Even that is NP-hard [9]. Work on heuristics for both fixed-layer and general bigraph crossing has mostly been theoretical [4,8,9,20,21]. Experimental evaluation has focused on dense graphs, for which good lower bounds are available [5,18], and sparse random graphs [18]. Graphs arising in circuit design are very sparse and highly structured. Moreover, the `dot` package [10], used widely for graph drawing, has not been compared with other heuristics. This paper addresses these shortcomings by (a) evaluating heuristics based on their performance on general (rather than fixed-layer) bigraph crossing, (b) presenting new heuristics that perform well on instances related to circuit design and random graphs of the same sparsity, (c) doing careful comparison of new heuristics with `dot` and other heuristics in the literature, and (d) presenting an experimental methodology that allows us to validate heuristics on data that ranges from application-specific to random.

The performance of a heuristic can vary widely even on a single input graph, depending on the order in which the input is presented. This motivates us to define a *presentation* (of a graph G) as $\langle G, \pi_0, \pi_1 \rangle$, where π_i is a permutation of V_i. As any heuristic implicitly sequences the input when it reads data, the presentation captures essential information about any of the common ways of describing a bigraph. If described as a list of neighbors for each V_0 node, π_0 describes the order of appearance of the V_0 nodes while π_1 is used to sort the adjacency lists. A list of edges is sorted using π_0 as primary key and π_1 as secondary key. Quite conveniently, a presentation also yields an embedding of G: use π_i to sequence the V_i vertices along $y = i$. Let $C(\langle G, \pi_0, \pi_1 \rangle)$ denote its crossing number. The object of the bigraph crossing problem, then, is to compute (or approximate) $C(G) = \min_{\pi_0,\pi_1} C(\langle G, \pi_0, \pi_1 \rangle)$.

Pushing this idea further, a heuristic h is a mapping from one graph presentation to another, that is $h(\langle G, \pi_0, \pi_1 \rangle) = \langle G, \pi_0', \pi_1' \rangle$ and we can study its behavior statistically by looking at how a distribution on random $\langle G, \pi_0, \pi_1 \rangle$ drawn from a class of presentations imposes a distribution on $C(h(\langle G, \pi_0, \pi_1 \rangle))$. Recently the distribution of $C(h(\langle G', \pi_0, \pi_1 \rangle))$ using a specific heuristic h became an important factor in characterizing a graph [14]: π_0 and π_1 were chosen randomly and G' was chosen from a well-defined set of perturbations of G. In this paper, we demonstrate the converse: sets of graph presentations have an important role in the design of experiments such that the performance of the various known bigraph crossing heuristics can be measured with statistical significance. Moreover, the experiments have stimulated the development of a new heuristics and we demonstrate that the improved performance is due to significant improvements of the algorithms and not due to chance.

The paper is organized as follows: Section 2 motivates the proposed approach, Section 3 presents the proposed heuristics, both existing and new, Section 4 describes the design of our experiments, in terms of heuristics applied and data sets used, Section 5 summarizes experimental results, and Section 6 presents tentative conclusions and suggests further work. An appendix describes how to access and use our data and programs via the world-wide web.

2 Background and Motivation

Routinely, we use `dot` [10] as the crossing number minimization algorithm to characterize and distinguish equivalence classes of graph presentations as defined in [14]. In 1997, we posted on the Web results of a very basic experiment on three families of equivalence classes of bigraph presentations: (a) *isomorphism* classes (classes in which G remains fixed but π_0 and π_1 vary) based on a biplanar graph, (b) isomorphism classes based on a graph with a two cycles joined at an articulation point ($C(G)$ is known for this graph, which we call *cyclic*), (c) perturbed classes (presentations have the form $\langle G', \pi_0, \pi_1 \rangle$ where G' is a well-defined perturbation of G) based on the graphs in (b) [2]. It quickly became apparent that `dot` performs far from optimal, even for graphs with as few as 9 nodes.

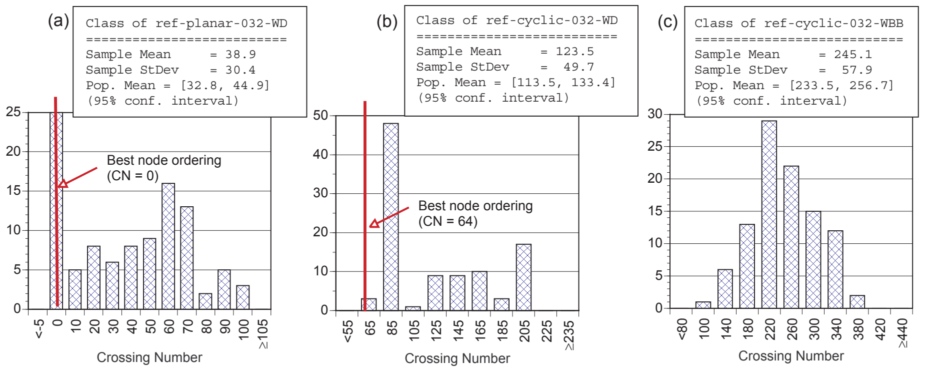


Fig. 1. Crossing number distributions and statistics observed with program `dot` on three equivalence classes of bigraphs: (a) 100 instances of isomorphic biplanar graphs, (b) 100 instances of isomorphic non-planar graphs with a minumum crossing number of 64, (c) 100 instances of mutant graphs of (b).

An example of the observed performance with `dot` on the three classes is shown in Figure 1. All graphs analyzed in Figure 1 are of comparable size and density: the biplanar graph has 128 nodes and 129 edges, the cyclic graph has 131 nodes and 132 edges and a crossing number of 64, each of the mutant graphs has 131 nodes and 132 edges and the crossing number is expected to be a random variable due to the construction of the mutant graphs. The histograms in Figure 1 demonstrate that for graphs of this size, $C(\texttt{dot}(\langle G, \pi_0, \pi_1\rangle))$, the crossing number as reported by `dot`, behaves as a random variable not only for the perturbed class in Figure 1(c) as expected, but also for the two isomorphism classes where, ideally, each distribution should have variance of 0 and population means of 0 for Figure 1(a) and 64 for Figure 1(b). Notably, only 25 of 100 graph presentations in Figure 1(a) achieve $C(\texttt{dot}(\langle G, \pi_0, \pi_1\rangle)) = 0$; there are 10 presentations whose crossing number distribution ranges from 80 to 100. For the 100 graph presentations in Figure 1(b), the best reported crossing number is 70, and there are 5 presentations with $C(\texttt{dot}(\langle G, \pi_0, \pi_1\rangle)) = 211$!

The results of simple experiments summarized in Figure 1 provide a strong motivation for the premise of this paper: *by reducing the dependency of the crossing number minimization algorithm on the initial order of nodes in the graph, we will have improved the overall performance of the algorithm.*

We illustrate the nature of the problem with the example in Figure 2. The

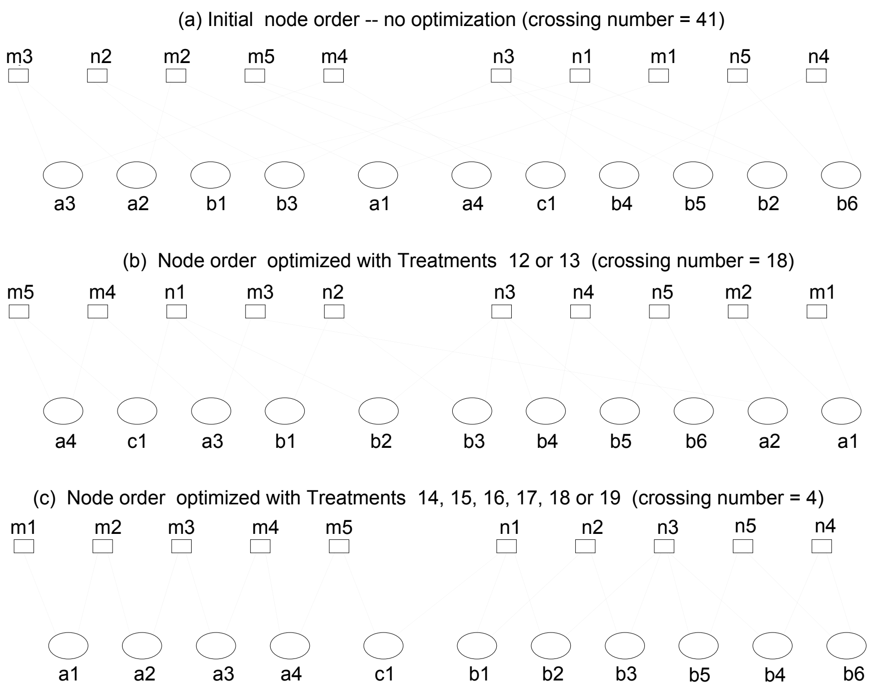

(a) Initial node order -- no optimization (crossing number = 41)
m3 n2 m2 m5 m4 n3 n1 m1 n5 n4
a3 a2 b1 b3 a1 a4 c1 b4 b5 b2 b6
(b) Node order optimized with Treatments 12 or 13 (crossing number = 18)
m5 m4 n1 m3 n2 n3 n4 n5 m2 m1
a4 c1 a3 b1 b2 b3 b4 b5 b6 a2 a1
(c) Node order optimized with Treatments 14, 15, 16, 17, 18 or 19 (crossing number = 4)
m1 m2 m3 m4 m5 n1 n2 n3 n5 n4
a1 a2 a3 a4 c1 b1 b2 b3 b5 b4 b6

Fig. 2. A simple graph example illustrating: (a) a presentation that is found 'difficult' for the `dot` heuristic, (b) the presentation found by `dot`, (c) an optimum presentation found by heuristic combinations `tr14-tr19` reported in this paper.

initial order of the bigraph in Figure 2(a) reports a crossing number of 41. The best node ordering achieved by `dot` reports a crossing number of 18 and is shown in Figure 2(b). However, the optimal node ordering achieved by several heuristics reported in this paper results in a crossing number of 4 and is shown in Figure 2(c).

`Dot`'s initial ordering, based on breadth-first search, folds the path from `m1` to `n1` into an awkward position so that even arbitrarily many iterations of the subsequent median heuristic (`dot` stops at 24 iterations) cannot disentangle it. The initial ordering imposed by our two-pass *guided breadth-first search*, described in the next section, is shown in Figure 2(c). It is optimal.

3 Overview of the Heuristics

Our heuristics (experimental treatments) follow the scheme outlined in the classical paper by Warfield [29]. There are two phases: an initial ordering and iterative improvement. The former involves some global computation on the graph to sequence the nodes in each layer and the latter repeatedly solves a fixed-layer problem on alternating sides. Various combinations of initial ordering and iterative improvement make up the experimental treatments discussed in Section 4.

3.1 Initial Ordering

Two observations suggest that a carefully computed initial ordering can avoid traps for subsequent attempts at improvement without incurring prohibitive execution time. First, a connected graph is biplanar iff it is a *comb* (tree that becomes a path when its leaves are removed) [15]. The embedding of a comb can be computed easily in linear time [6]: the trick is to embed the *spine* (path that remains when leaves are deleted from the comb) from left to right, zigzagging from layer to layer, and inserting any leaves attached to a spine node to the left of the next spine node. Second, a cycle of length $2n$ has an optimum embedding with $n-1$ crossings [16] and that embedding is achieved if the nodes are sequenced in breadth-first order starting at any node of the cycle.

We use two initial orderings in our experiments in addition to the random ordering (obtained by using the input presentation), for a total of 3 possibilities. One ordering uses a breadth-first search starting at a random node. This always gets the optimal solution for a simple cycle, and tends to perform better than input ordering on the classes of graph presentations we looked at. The `dot` heuristic also starts with a breadth-first search, but has additional features related to aesthetic objectives other than minimizing crossing.

Our contribution is an initial ordering heuristic called *guided breadth-first search (GBFS).* Two passes of breadth-first search are done. In the first pass, each node v is given $dist[v]$, the distance from the start node, and $depth[v]$, the maximum value of $dist[w]$ achieved by any descendant w of v in the breadth-first search tree. The second pass begins the search at a node s for which $dist[s]$ is maximized. Adjacency lists are sorted by increasing $depth$ and ties are broken by visiting nodes w with larger $dist[w]$ first. Sequence numbers are assigned to nodes based on the order of visitation in the second search. Finally, the sequence numbers are used to sort nodes on each side of the bigraph. All of the above can be accomplished easily in linear time: the *depth* values can be calculated as nodes are visited in reverse order at the end of the first bfs ($depth[v] = \max_{w \text{ a child of } v} depth[w]$), and all sorting is based on values that range from 1 to n (bin sorts of all adjacency lists can be combined). Our actual implementation is not linear time — it uses insertion sort — but it runs fast enough in practice (and timing is not an issue we address directly in this paper).

Figure 3 shows the two breadth-first searches of GBFS on the example in Figure 2. The first search begins at `n4`, but that choice is completely arbitrary.

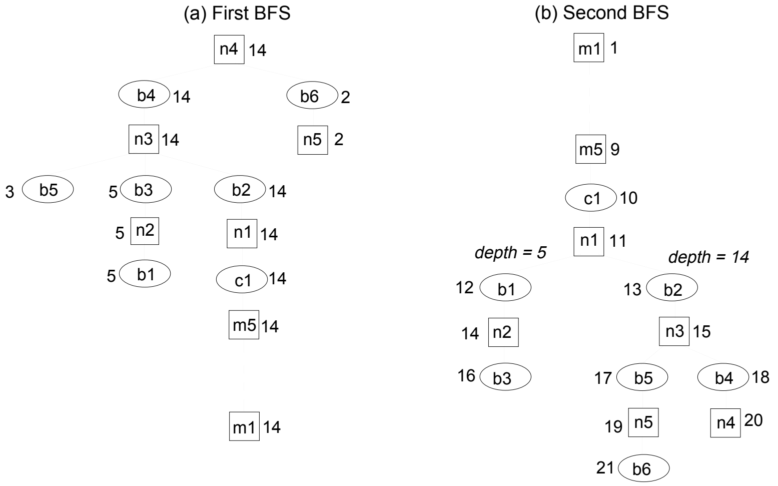


Fig. 3. GBFS starting at node `n4` of Figure 2(a). Numbers next to nodes indicate depth in (a) and sequence numbers in (b).

In some cases starting the first bfs at a node of maximum degree improves performance of GBFS considerably. A max-degree node is likely to be an articulation point included in more than one cycle, and beginning the search there reduces the likelihood that two cycles that can be embedded without interference will cross each other. We include this enhancement in our implementation, though it appears to be superfluous when a good iterative improvement strategy follows GBFS. It is easy to show that GBFS always obtains optimal solutions for biplanar graphs and simple cycles.

3.2 Iterative Improvement

To improve upon the initial orderings discussed above, we repeatedly apply a heuristic for fixed layer crossing. Our experiments use two popular previously-known heuristics and a new heuristic called *adaptive insertion*. These heuristics, described in detail below, are used separately or in combination. Regardless of which heuristic is used, our implementation has a parameter that governs the maximum number of iterations without improvement, set arbitrarily at 24. After each application of the heuristic, the current ordering is saved if it has fewer crossings than any ordering observed so far. The discovery of a better solution also resets the iteration count to 0. For purposes of the discussion below, we define each heuristic to mean all iterations of it. An *iteration* consists of two passes, one on each layer, where a *pass* is a single application of the original heuristic for the fixed layer problem.

The *median* heuristic treats the neighbors of each node as a set of integers representing their ordinal numbers on the opposite side. Nodes are sorted using the medians of these sets as keys. Implementations of the median heuristic differ in how the median of a set of even cardinality is computed. Ordinarily, the median would be defined as the mean of the two middle elements. The `dot` package [10] uses the mean when there are exactly two elements and a biased mean (biased toward the side on which neighbors are closer together in the ordering) otherwise. The median heuristic for which theoretical bounds are known (see [4,9,20]) always uses the smaller of the two candidates, but with the added condition that, in case of ties, nodes with odd degree always precede those of even degree. We adopt this latter median heuristic in our experiments. Assuming random initial ordering and a stable sort, the probability that the median heuristic embeds a simple path optimally is $O(1/n)$. However, it almost always embeds a simple cycle optimally (without the limit of 24 iterations, it would succeed every time).

The *barycenter* heuristic uses the mean of the set of neighboring positions, rather than the median, as a key for sorting (the name comes from the fact that it's a one-dimensional analog of Tutte's barycenter method for drawing graphs [27,28]). It performs poorly on paths, cycles, and other very sparse highly-structured graphs (since these require decisive movement of degree-2 nodes). On graphs that are more random and/or have several nodes of high degree, barycenter does better than median (high-degree nodes need to be centrally located wrt to their neighbors).

Sometimes a mix of median and barycenter does better than either. Our implementation has a parameter α and sorts nodes using keys $\alpha B + (1 - \alpha)M$, where B and M are the barycenter and median keys, respectively. When the mix is a significant improvement, the best choice of α appears to be 0.5 (as opposed to 0.25 or 0.75).

Each pass of the median heuristic can be implemented in linear time (the keys used for sorting are integers in the range $0, \ldots, n-1$), while the barycenter or mixed heuristics require $O(m + n \log n)$ per pass (m is the number of edges, n the number of nodes).

The new ordering computed by a single pass of the median or barycenter heuristic is independent of any cost considerations and therefore non-adaptive in the sense that any rearrangement is done whether or not it decreases cost. The `dot` heuristic, however, has a final phase that decides whether or not to exchange the nodes (at positions) i and $i+1$ on layer ℓ based on whether $d_\ell(i, i+1)$, the resulting difference in crossing number, is negative (represents improvement). The value $d_\ell(i, i+1)$ depends only on the relative positions of neighbors of nodes i and $i+1$ on layer $1-\ell$ and is therefore easy to compute [4, pp. 281–283].

Adaptive insertion, our contribution to iterative improvement, generalizes the conditional exchange idea and considers the effect of inserting node i before (after) any node $j < i$ ($j > i$) on layer ℓ. When $j < i$ the resulting cost change, $D_\ell(i, j)$, is $\sum_{j \le k < i} d_\ell(i, k)$ — the effect of the insertion is that of a series of swaps of i with $i-1, \ldots, j$ (situation is symmetric when $j > i$).

One pass of adaptive insertion does a right-to-left sweep of nodes on a layer ℓ. If the current node i has not already been inserted, the position j, before or after, with the best $D_\ell(i, j)$ is found (nodes are not allowed to stay in place even if any insertion would increase the number of crossings).

Consider the example in Figure 2(a) and suppose adaptive insertion is applied to the upper layer $\ell = 1$. The rightmost node is `n4` and $D_1(\texttt{n4}, \texttt{n5}) = d_1(\texttt{n4}, \texttt{n5}) = -1$. Since $d_1(\texttt{n4}, \texttt{m1}) = 2$, the value of $D_1(\texttt{n4}, \texttt{m1})$ increases to 1. As we move farther to the left, considering insertion positions for `n4`, the value of $D_1(\texttt{n4}, x)$ continues to increase, so the best choice is to insert `n4` before `n5`. The sweep then moves left to `n5`, which is skipped, having already been inserted (`n4` does not have an opportunity to be inserted in this phase; giving it one would be counterproductive, since it would simply be swapped with `n5`). Next `m1` finds its optimal position before `m5` with $D_1(\texttt{m1}, \texttt{m5}) = -5$. And so on. Figure 4 shows the results of the completed pass of adaptive insertion on the graph of Figure2(a). The node `m3` was forced to swap with `m2` (the least of evils) even though this caused a net gain of 3 crossings.

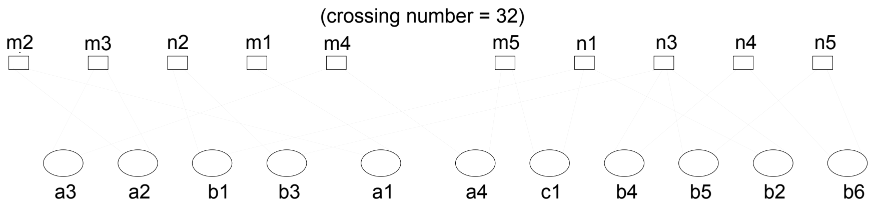


Fig. 4. One pass of adaptive insertion starting with presentation shown in Figure 2(a).

Adaptive insertion seems to do well where the median does badly and vice-versa. An effective combination alternates iterations of adaptive insertion with iterations of the median/barycenter mix. This, combined with an initial GBFS ordering, is the best overall performer among our heuristics.

One pass of adaptive insertion takes $O(m^2)$ time. If all adjacency lists are initially sorted using layer $1 - \ell$ positions as keys (this can be done in linear time), the calculation of $d_\ell(i,k)$, and hence $D_\ell(i,k)$, for all $k \neq i$ takes time $O(mk)$ where $k = deg(i)$. Summing over all nodes i gives the $O(m^2)$ bound. For denser graphs the time per node can be reduced to $O(m \log k)$ using a simple data structure, for an overall bound of $O(mn \log(m/n))$ per pass.

4 Experimental Design

Experiments with results such as summarized in Figure 1 demonstrate that particular executions of a heuristic to solve the optimum node ordering in bigraphs offer no indication of the quality of the solutions produced in general. Changing the starting point for the problem instance can induce unpredictable variability of results when experiments are repeated.

Two of the fundamental principles of experimental design are *randomization* and *replication*. We adopt these principles for the experimental evaluation of heuristics by (1) creating a *presentation equivalence class*, (2) repeating the experiments for each member in the class, and (3) making the equivalence class and the heuristics available so that other researchers can repeat our experiments (or variations of them; see the Appendix for more details). The basic abstractions for such experiments include [1]:

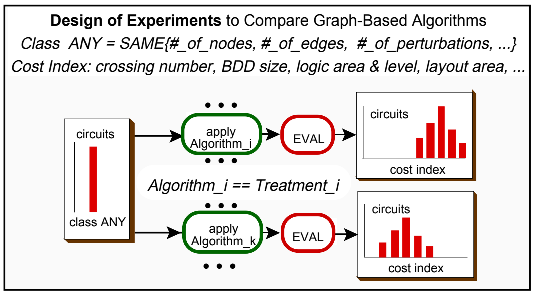
1. an equivalence class of experimental subjects, eligible for a treatment;
2. application of a specific treatment;
3. statistical evaluation of treatment effectiveness.

Here, a *treatment* is synonymous with a *heuristic* and an equivalence class of experimental subjects is synonymous with a graph presentation class. Figure 5 illustrates these abstractions in a generic flow. The cost index, minimized by permuting the nodes at both levels of the graph, is $C(h(\langle G, \pi_0, \pi_1 \rangle))$, where h is any of the treatments in Figure 5.

The treatments can be divided into three main groups based on the initial ordering (see the previous section) used: `tr00-tr05` represent random (input) order, with `tr00` playing the special role of a placebo treatment; `tr06-tr11` represent breadth-first ordering; and `tr14-tr19` represent ordering obtained from our GBFS. The remaining treatments, `tr12` and `tr13` are `dot` with 24 and 48 iterations, respectively.

The iterative improvement heuristic used (previous section) determines the treatment order within each group: the first treatment (`tr00`, `tr06`, `tr14`) does no iterative improvement; the second through fourth (`tr01-tr03`, `tr07-tr09`, `tr15-tr17`) do the median, median/barycenter mix, and barycenter; and the last two (`tr04-tr05`, `tr10-tr11`, `tr18-tr19`) do adaptive insertion, either by itself or alternated with a median/barycenter mix.

Experimental subjects are drawn from circuit layout problems. The example in Figure 2, is representative of graphs that can be extracted from VLSI designs


Design of Experiments to Compare Graph-Based Algorithms
Class ANY = SAME{#_of_nodes, #_of_edges, #_of_perturbations, ...}
Cost Index: crossing number, BDD size, logic area & level, layout area, ...
circuits
class ANY
apply Algorithm_i
EVAL
circuits
cost index
Algorithm_i == Treatment_i
apply Algorithm_k
EVAL
circuits
cost index

Legend	
natural	node order given by the input graph
BFS	breadth-first order
GBFS	2-pass guided breadth-first order (see Section 3)
dot	as described in [10]
median	as described in [4]
barycenter	as described in [4]
median/ barycenter	combines median and barycenter (see Section 3)
adaptive insertion	inserts each vertex optimally (see Section 3)
adaptive insertion++	adaptive insertion alternated with median/ barycenter

Treat-ment	Initial Ordering	Final Ordering
0	natural	none
1	natural	median
2	natural	median/ barycenter
3	natural	barycenter
4	natural	adaptive insertion
5	natural	adaptive insertion++
6	BFS	none
7	BFS	median
8	BFS	median/ barycenter
9	BFS	barycenter
10	BFS	adaptive insertion
11	BFS	adaptive insertion++
12	dot	dot (24 it.)
13	dot	dot (48 it.)
14	GBFS	none
15	GBFS	median
16	GBFS	median/ barycenter
17	GBFS	barycenter
18	GBFS	adaptive insertion
19	GBFS	adaptive insertion++

Fig. 5. Design of experiments with bigraph equivalence classes to compare the crossing numbers under distinctive vertex ordering algorithms (treatments).

such as shown in Figure 6. This particular example is based on the largest connected component found in a dag, referenced as a `classifier controller` or `cc` in [24]. Since the circuit `cc` implements the controller, there are a few nodes with a large out-degree (or fanout) that introduce unavoidable edge crossings; note however, that there is also a small biplanar region. Before introducing families of parameterized graphs that have properties of the graph in Figure 7, we briefly describe the graph mutation process as introduced in [14].

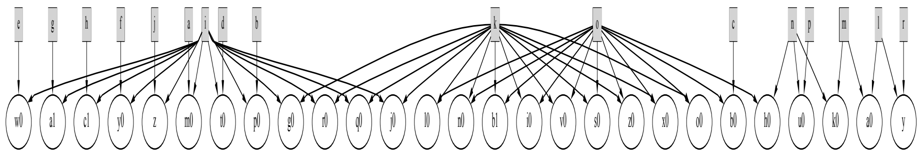


Fig. 6. Example of a bigraph, extracted as a *connected component* from a dag of a VLSI circuit.

The graphs in Figures 2 and 6 are directed bigraphs: the square nodes represent signal source nodes or *net nodes*, the oval nodes represent the signal processing nodes or *cell nodes*. Since the graph under consideration has two layers only, there is no need to define sink nodes. In VLSI circuits, the out-degree or *fanout* of net nodes is assumed to be a loosely bounded random variable, the in-degree or *fan-in* of the cell nodes is assumed to be a tightly bounded random variable. By design, and for simplicity of inducing the equivalence classes of bigraph presentations, all cell nodes considered in this paper have a fan-in of 2. The graph such as shown in Figure 2 illustrates an instance of a *reference graph* G_r with a *characteristic signature* $\sigma = \{10, 11\}$ where 10 is the total number of net nodes and 11 is the total number of 2-input cell nodes.

From each reference graph G_r, we can define three equivalence classes of *derived presentations* (see [14] for more details and more possibilities): (1) the *isomorphism class* for G_r (designated by the suffix `-WD` in the figures reporting results), consisting of presentations $\langle G_r, \pi_0, \pi_1 \rangle$ (only the permutation of nodes on each layer differs). (2) the *mutant class* for G_r (suffix `-WBB`), consisting of presentations $\langle G'_r, \pi_0, \pi_1 \rangle$, where G'_r is a random connected graph with the same signature as G_r (we assume G_r is connected as well), that is, the same number of degree-2 cell nodes on one layer (hence the same number of edges as well), and the same number of net nodes, and (3) the *random class* for G_r (suffix `-WRD`), consisting of presentations $\langle G'_r, \pi_0, \pi_1 \rangle$, where G'_r is completely random with the same number of nodes on each layer and the same number of edges as G_r (G'_r is not necessarily connected, nor does it have fan-in 2 for cell nodes, but it has no isolated nodes). We took a special precaution to assign to each instance of any class the following properties:

P1: the order of all cell and net nodes in each presentation is random relative to all other members in the class (that is, π_0 and π_1 are uniformly random),

P2: the names of all cell and net nodes in each presentation are assigned randomly relative to all other members in the class.

Properties **P1** and **P2** are essential to good experimental design and must be maintained universally for all equivalence classes. The purpose of **P1** is clear. Without **P2**, some programs that rely on hashing the input data may *unknowingly undo* the randomization of input presentations and confound the experiments. An important lesson on this subject has been learned and reported in [17].

In addition to using the `cc` graph directly, we use three types of reference graphs in sizes that are increasing powers of 2: (1) the *biplanar* type of size q is a comb with $2q + 1$ net nodes and $2q$ cell nodes for a total of $4q + 1$ nodes and $4q$ edges, (2) the *cyclic* type of size q consists of two simple cycles each having $q + 2$ nodes and edges and joined at a net node (which becomes an articulation point), and (3) the *combined* type of size q is a biplanar graph of size q joined to a cyclic graph of size q via an additional connector (cell) node for a total of $8q + 5$ nodes and $8q + 6$ edges. Figure 2 is an example of a combined graph of size 2.

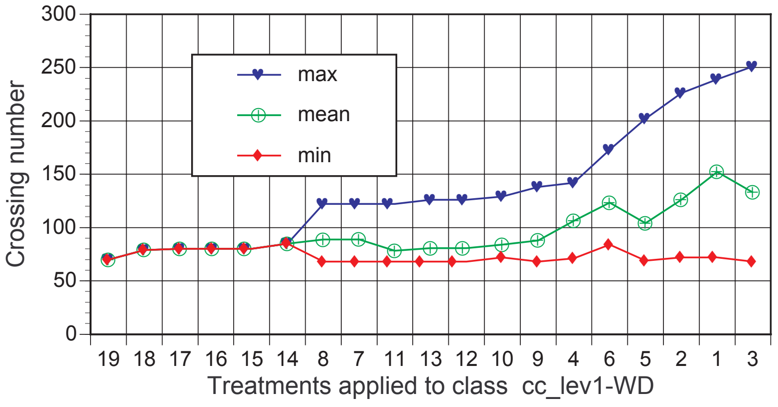
In the following section we report the results of 12 distinct experiments, consisting of isomorphism classes, mutant classes, and random classes based on the `cc` graph and each of the reference graph types biplanar, cyclic, and combined. The increasing sizes indicate asymptotic trends in the relative behavior of the heuristics.

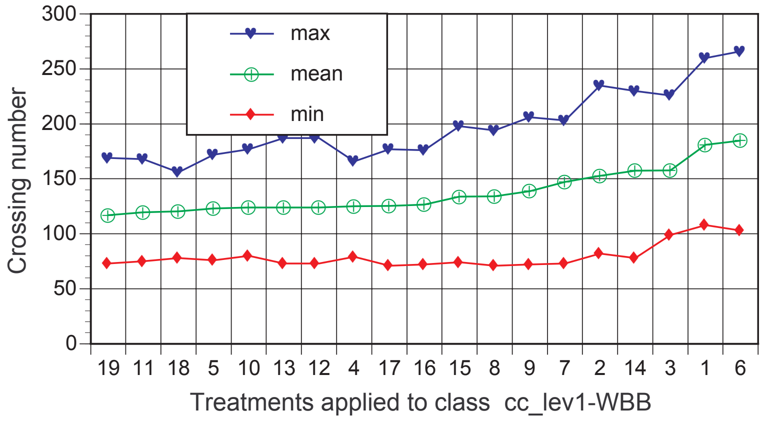
5 Experimental Results

The experiments are based on the model shown in Figure 5. Treatments 0–19 are applied to *all* classes of data described in the paper. For each presentation of input $\langle G, \pi_0, \pi_1 \rangle$, a treatment h_k induces a treated representation $h_k(\langle G, \pi_0, \pi_1 \rangle)$ as $\langle G, \pi_0', \pi_1' \rangle$ which is in turn evaluated by a *common crossing number evaluator*, `cn_eval`. The evaluator (1) reads a file of the graph description in `dot` format, as well as a companion file that simply stores the vertices V_i in the order determined by π_i', and (2) reports the crossing number of the presentation.

All classes of input data are organized for easy access through the Web, like the variety of illustrative experimental designs reported earlier [3]. As in [3], web-posting includes complete tables of crossing number results for all treatments and all classes of input data, along with statistical summaries that include confidence intervals of the reported means, and t-tests to analyze the significance of reported differences between the means. Since the treatment 19 is clearly the best overall treatment reported in the current work, we also post node permutations π_0', π_1' along with the respective crossing numbers reported by this treatment. Readers of this text should look under `http://www.cbl.ncsu.edu/experiments/-DoE_Archives/DoE_0003` for completed archives of experimental data and programs presented in this paper (see the Appendix for more information).

Due to space limitations, we confine the statistical summaries of our experiments to illustrations shown in Figure 7 and 8.





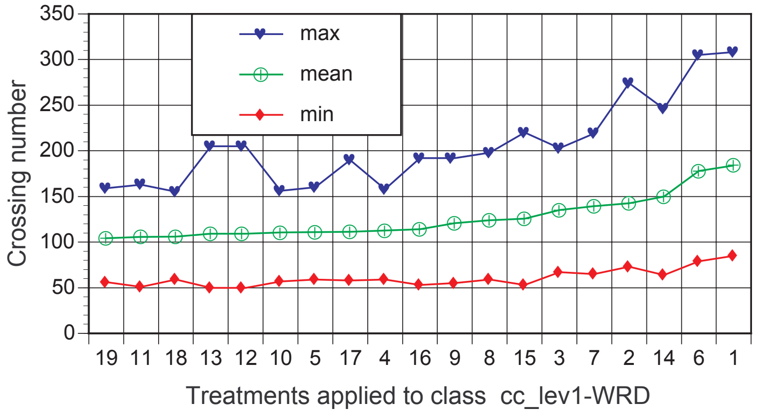


Fig. 7. Crossing number reports for treatments 0–19 applied to three equivalence classes derived from the reference bigraph `cc_lev1`.

5.1 Figure 7

Figure 7 shows treatments sorted by mean cost on the `cc` reference graph. The best heuristic overall is a combination of several: GBFS initial order, followed by adaptive insertion alternated with a median/barycenter mix. This multi-heuristic combination consistently does better on all presentation classes and also has better asymptotic behavior with respect to solution cost than any of the others. The placebo, `tr00`, is not shown: random solutions are far worse than those produced by any of the heuristics. For the isomorphic class, they range from 440 to 752 crossings; for the mutant class from 501 to 780; and for the random class from 314 to 818.

Even though the difference between bfs and GBFS is dramatic for the isomorphism class when these heuristics act alone (`tr06` and `tr14`), the difference decreases significantly as iterative improvement is added (`tr10` vs. `tr18`, `tr11` vs. `tr19`) or the classes become more random (mutant and random versus isomorphism). This is not surprising since GBFS was inspired by the circuit-derived examples.

Adaptive insertion is clearly a better strategy for iterative improvement than any of the others, but it is also the most time intensive. This suggests that bigraph crossing heuristics have not yet reached a point of diminishing return in terms of investment in execution time. Whether this should be exploited by increasing the number of iterations without improvement or by developing more sophisticated local search strategies remains unclear, most likely the latter.

5.2 Figure 8

Here we summarize results of the remaining 9 experiments (3 graph types, each with 3 different presentation classes). The plots that show only `tr19`, the best overall heuristic, illustrate the growth in objective function value on a log/log scale. Except for the biplanar isomorphism class, the number of crossings reported grows polynomially with the number of edges, the slope representing the exponent. For the biplanar classes, the difficulty (from the point of view of `tr19`) increases with the degree of randomness. The mutant class (`WBB`) contains trees that are not biplanar, but no cycles, while the random class may also contain cycles. This suggests that non-biplanar trees can be pretty difficult, but not as difficult as more general bigraphs. In the cyclic and combined (`multi1`) classes, the mutants are more difficult than the random bigraphs: recall that the mutants have a single connected component while the random graphs, because of their sparsity, are unlikely to have even a large connected component relative to the rest of the graph.

The other plots show that the difference between crossing numbers obtained by `tr19` and three competitors increases with increasing bigraph size. A log/linear scale is used here to make the separation more visible, but it should be pointed out that all of the differences grow nonlinearly (they curve upward even on a linear/linear scale but the lines are no longer as distinct from each other). `Dot`

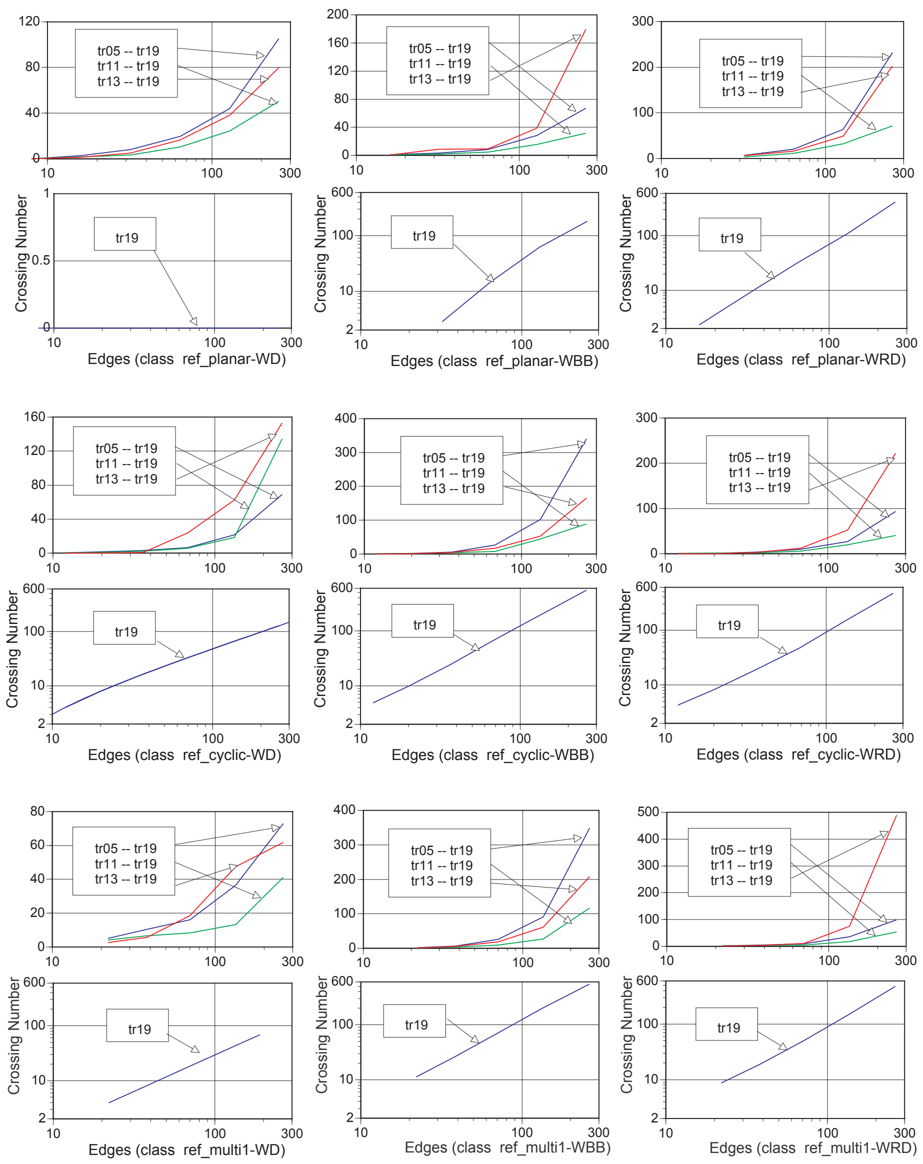


Fig. 8. Summary of asymptotic experiments on minimizing the *crossing number mean* with four treatments (tr05, tr011, tr13, tr19), applied to three parameterized families of reference bigraphs (biplanar, cyclic, and combined, i.e. `multi1`), each bigraph inducing three equivalence graph classes, each class with the same number of edges and nodes (and no isolated nodes): isomorhism class (WD), signature-invariant and component-invariant mutant class (WBB), and random class (WRD).

(represented by `tr13` is clearly a competitive heuristic and is put at a disadvantage with respect to the adaptive insertion heuristics as the graph size increases because of its fixed iteration limit. An adaptive bound on the number of iterations, as the authors of [10] suggest might improve its relative performance.

Our hypothesis that better initial orderings make a difference is clearly supported, even in the presence of sophisticated iterative improvement, as seen by the growing gaps between `tr19` and `tr11`, and again between `tr11` and `tr05`; all three use adaptive-insertion++ for iterative improvement.

6 Conclusions and Further Work

This study is only the beginning of what we hope is a new approach to experimental study of bigraph crossing and other intractable problems. Input data and treated output can be shared and verified so that different groups working on the same problem can conduct repeatable experiments. Better heuristics are often developed through a detailed understanding of why specific instances present difficulties, and such understanding is made more likely when data is a significant component of the experimental design.

Some directions to pursue later include

- Better lower bounds for sparse graphs: these could be based on the fact that a tree requires at least as many crossings as the minimum number of edges that need to be deleted in order to produce a comb and on the theoretical lower bound for cycles (it appears that these can be combined additively for a spanning tree of a graph and its fundamental cycles).
- Even better initial ordering: there are simple examples of trees on which GBFS does poorly. A useful approach might be to find biconnected components and then try to optimize the underlying tree structure. For a tree T, it is known that $C(T)$ can be computed in polynomial time [25], although the algorithm appears to be complicated.
- More powerful iterative improvement heuristics: all the ones presented here have at most quadratic time per pass. Are there heuristics that obtain superior solution quality for the price of more computation time?
- Use heuristics as a filter for creating presentation classes in which the distribution of π_0, π_1 changes from uniformly random to some other distribution in order to study which treatments work best in sequence.
- Theoretical work on the median and barycenter heuristics: previous work (as reported in [4]) only addresses the fixed-layer problem. It also appears that the median and barycenter heuristics have interesting convergence properties (they converge quickly to a “local optimum”).
- Relationship between bigraph crossing and other objective functions for layout of VLSI circuits: preliminary experiments [12,13] indicate a high correlation with wire length in the final routing obtained by at least two different design automation tools. In fact, our bigraph crossing heuristics appear to achieve better wire length than layout heuristics that are specifically designed to minimize wire length. Relevant theoretical work relates bigraph crossing to optimum linear arrangement [25].

Acknowledgments. Some of the initial crossing number experiments with 2-layer graphs posted on the Web in 1997 were organized and executed by Nevin Kapur. He also contributed the software utilities that support gathering and tabulating data about the experiments and the statistical summaries. Hemang Lavana contributed utilities to dynamically generate web-pages of directories that link the data related to the reported experiments. We appreciate their support.

References

1. F. Brglez, *Design of Experiments to Evaluate CAD Algorithms: Which Improvements Are Due to Improved Heuristic and Which Are Merely Due to Chance?*, Tech. Rep. 1998-TR@CBL-04-Brglez, CBL, CS Dept., NCSU, Box 7550, Raleigh, NC 27695, April 1998. Also available at `http://www.cbl.ncsu.edu/publications/-#1998-TR@CBL-04-Brglez`.
2. F. Brglez, N. Kapur, and D. Ghosh, *A CBL PosterNote: Wire Crossing Problem and Benchmarking*, aug 1997. Available from `http://www.cbl.ncsu.edu/-DiscussionGroups/Benchmark-reviews/frm00002.html`.
3. F. Brglez (Editor), *A Brief Tour From The Home Page on WWW Statistical Experiment Archives: Benchmark Descriptions and Posted Solutions to NP-hard Problems*, Tech. Rep. 1998-TR@CBL-01-Brglez, Version 1.0, CBL, CS Dept., NCSU, Box 7550, Raleigh, NC 27695, February 1998. Also available at `http://www.cbl.ncsu.edu/publications/#1998-TR@CBL-01-Brglez`.
4. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999.
5. P. Eades and D. Kelly, *Heuristics for reducing crossings in 2-layered networks*, Ars Combinatoria, 21-A (1986), pp. 89–98.
6. P. Eades, B. D. McKay, and N. C. Wormald, *On an Edge Crossing Problem*, tech. rep., Department of Computer Science, University of Newcastle, New South Wales 2308, Australia, 1976.
7. P. Eades and K. Sugiyama, *How to Draw a Directed Graph*, Journal of Information Processing, 13 (1990), pp. 424–437.
8. P. Eades and S. Whiteside, *Drawing graphs in two layers*, Theoretical Computer Science, 131 (1994), pp. 361–374.
9. P. Eades and N. C. Wormald, *Edge Crossings in Drawings of Bipartite Graphs*, Algorithmica, 11 (1994), pp. 379–403.
10. E.R. Gansner, E. Koutsifios, S.C. North and K.P. Vo, *A Technique for Drawing Directed Graphs*, IEEE Trans. Software Engg., 19 (1993), pp. 214–230. The drawing package `dot` is available from `http://www.research.att.com/sw/tools/graphviz/`.
11. M. R. Garey and D. S. Johnson, *Crossing Number is NP-complete*, SIAM J. Algebraic Discrete Methods, 4 (1983), pp. 312–316.
12. D. Ghosh, F. Brglez, and M. Stallmann, *First steps towards experimental design in evaluating layout algorithms: Wire length versus wire crossing in linear placement optimization*, Tech. Rep. 1998-TR@CBL-11-Ghosh, CBL, CS Dept., NCSU, Box 7550, Raleigh, NC 27695, October 1998. Also available at `http://www.cbl.ncsu.edu/publications/#1998-TR@CBL-11-Ghosh`.

13. ———, *Hypercrossing number: A new and effective cost function for cell placement optimization*, Tech. Rep. 1998-TR@CBL-12-Ghosh, CBL, CS Dept., NCSU, Box 7550, Raleigh, NC 27695, December 1998. Also available at `http://www.cbl.ncsu.edu/publications/#1998-TR@CBL-12-Ghosh`.
14. D. Ghosh, N. Kapur, J. E. Harlow, and F. Brglez, *Synthesis of Wiring Signature-Invariant Equivalence Class Circuit Mutants and Applications to Benchmarking*, in Proceedings, Design Automation and Test in Europe, Feb 1998, pp. 656–663. Also available at `http://www.cbl.ncsu.edu/publications/-#1998-DATE-Ghosh`.
15. F. Harary and A. J. Schwenk, *Trees with Hamiltonian Square*, Mathematika, 18 (1971), pp. 138–140.
16. ———, *A New Crossing Number for Bipartite Graphs*, Utilitas Mathematica, 1 (1972), pp. 203–209.
17. J. E. Harlow and F. Brglez, *Design of Experiments in BDD Variable Ordering: Lessons Learned*, in Proceedings of the International Conference on Computer Aided Design, ACM, Nov. 1998. Also available from `http://www.cbl.ncsu.edu/-publications/#1998-ICCAD-Harlow`.
18. M. Jünger and P. Mutzel, *2–Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms*, Journal of Graph Algorithms and Applications (JGAA), 1 (1997), pp. 1–25.
19. F. T. Leighton, *New Lower Bound Techniques for VLSI*, Math. Systems Theory, 17 (1984), pp. 47–70.
20. E. Mäkinen, *A Note on the Median Heuristic for Drawing Bipartite Graphs*, Fundamenta Informaticae, 12 (1989), pp. 563–570.
21. ———, *Experiments on drawing 2-level hierarchical graphs*, International Journal of Computer Mathematics, 37 (1990), pp. 129–135.
22. M. Marek-Sadowska and M. Sarrafzadeh, *The Crossing Distribution Problem*, IEEE Transactions on Computer–Aided Design of Integrated Circuits and Systems, 14 (1995), pp. 423–433.
23. P. Mutzel, *The AGD-Library: Algorithms for Graph Drawing*, 1998. Available from `http://www.mpi-sb.mpg.de/m̃utzel/dfgdraw/agdlib.html`.
24. S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide*, Tech. Rep. 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991. Now available from `http://www.cbl.ncsu.edu/publications/-#1991-IWLS-UG-Saeyang` and benchmarks from `http://www.cbl.ncsu.edu/-benchmarks/Benchmarks-upto-1996.html`.
25. F. Shahrokhi, O. Sýkora, L. A. Székely, and I. Vrt'o, *On bipartite drawings and the linear arrangement problem*, in Lecture Notes in Computer Science, no. 1467, Springer Verlag, 1997, pp. 55–68.
26. C. D. Thompson, *Area–Time complexity for VLSI*, in Proceedings, 11th Annual ACM Symposium on Theory of Computing, May 1979, pp. 81–88.
27. W. T. Tutte, *Convex Representations of Graphs*, Proc. London Math. Soc., 10 (1960), pp. 304–320.
28. ———, *How to Draw a Graph*, Proc. London Math. Soc., 13 (1963), pp. 743–768.
29. J. N. Warfield, *Crossing Theory and Hierarchy Mapping*, IEEE Transactions on Systems, Man, and Cybernetics, SMC–7 (1977), pp. 505–523.

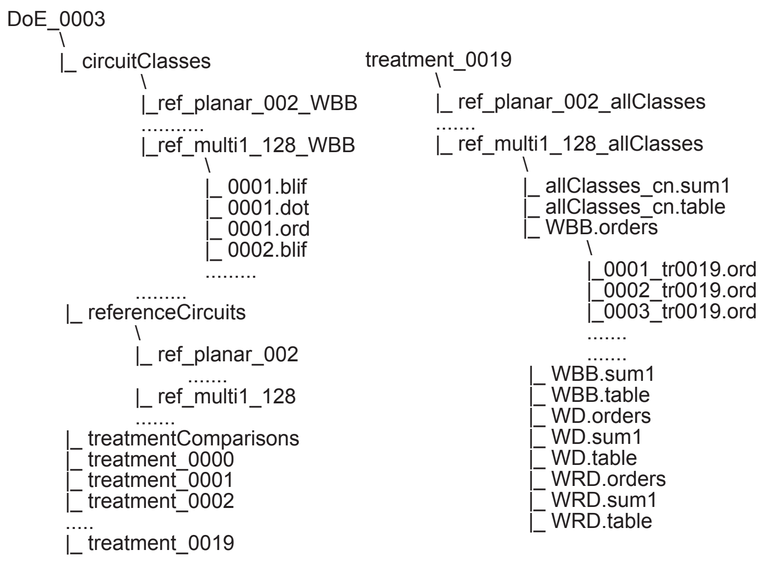
Appendix

Data set and programs used in our experiments are available on the Web at `http://www.cbl.ncsu.edu/experiments/DoE_Archives/DoE_0003`. The organization of this directory is outlined in Figure 9. Upon accessing the directory, researchers will find links and documentation to (1) reference circuits used to generate equivalence classes, (2) subdirectories of equivalence classes for each reference circuit (each class has 128 graphs), and (3) complete postings of results for treatments 0 to 19 as described in this paper.

For example, the `treatment_0019` archives results of experiments under Treatment 19 applied to each circuit class. The results of treatments for each class include (1) tables of crossing numbers for each graph instance in a *given class*; (2) statistics summary for each table such as mean, standard deviation, confidence interval for the mean, minimum and maximum; (3) tables of crossing numbers for each graph instance across *all classes*; (4) summary of crossing number statistics across all classes; (5) treatment-specific 2-layer graph orderings, archived for each equivalence class. Currently, orderings for Treatment 19 only are posted, since this treatment has generated node orderings with a crossing number mean that is consistently best across all equivalence classes.

The graph orderings, posted for Treatment 19, have been evaluated for a crossing number independently of the treatment that generated this order. The program we use is `cn_eval`, also available from the DoE_0003 archive. The program can be invoked with a simple command line

```
cn_eval graph.dot graph_trxxxx.ord
```



```
DoE_0003
    \
    |_ circuitClasses
            \
            |_ref_planar_002_WBB
            ...........
            |_ref_multi1_128_WBB
                   \
                   |_ 0001.blif
                   |_ 0001.dot
                   |_ 0001.ord
                   |_ 0002.blif
                   .........
            .........
    |_ referenceCircuits
            \
            |_ ref_planar_002
                  .......
            |_ ref_multi1_128
            .......
    |_ treatmentComparisons
    |_ treatment_0000
    |_ treatment_0001
    |_ treatment_0002
    .....
    |_ treatment_0019

treatment_0019
      \
      |_ ref_planar_002_allClasses
      .......
      |_ ref_multi1_128_allClasses
               \
               |_ allClasses_cn.sum1
               |_ allClasses_cn.table
               |_ WBB.orders
                      \
                      |_0001_tr0019.ord
                      |_0002_tr0019.ord
                      |_0003_tr0019.ord
                      .......
                      .......
               |_ WBB.sum1
               |_ WBB.table
               |_ WD.orders
               |_ WD.sum1
               |_ WD.table
               |_ WRD.orders
               |_ WRD.sum1
               |_ WRD.table
```

Fig. 9. DoE_0003 under http://www.cbl.ncsu.edu/experiments/DoE_Archives/

where `graph.dot` is an instance of the graph description in `dot` format, and `graph_trxxxx.ord` is the ordering associated with treatment `xxxx` (both formats are described briefly in this Appendix; `dot` format is also described on the Web [10]). The *initial orderings* for all instances of equivalence classes posted on the Web are associated with a random order, i.e. Treatment 0. The output of `cn_eval` is a single line of the form

`graph_trxxxx` *crossings*

where *crossings* is the number of crossings obtained when the graph is embedded using the given order.

The unique feature of our experimental design is thus the separation of the experiment into three parts: (1) the generation of experimental data (in this case programs that generate equivalence classes based on actual circuits), (2) the execution of heuristics, and (3) the evaluation of the cost function for the results produced by a heuristic. This separation of execution and evaluation makes it possible for other researchers to run their own heuristics on our data, submit new data to our heuristics, and evaluate any ordering independently with `cn_eval`.

Heuristics reported in this paper are implemented in the program `unfold2`, also available from the Web under the DoE_0003 archive. It is invoked with the command line

```
unfold2 -tr=N graph.dot order.ord
```

where `N` is the treatment number (as described in the paper). The extensions `.dot` and `.ord` are the expected extensions for graph description and layer ordering files, respectively. The program creates an output file called `graph_trxxxx.ord`, where `xxxx` is a 4-digit version of the treatment number (padded with 0's). This file specifies the ordering of the nodes on each layer as determined by the specified treatment.

Dot format. The expected input-file format for the graph.dot file is

```
digraph graph_name { statement; ... statement; }
```

where each statement defines an edge `layer0_node -> layer1_node`. Arbitrary white space is allowed between and within statements.

Ord format. An `.ord` file contains a sequence of layer descriptions separated by white space. Each layer description is of the form

```
layer_number { node_1 node_2 ... node_k }
```

where the layer number is an integer (0 or 1 in the case of bipartite graphs, but the notation can be generalized to more than 2 layers) and the `node_i`'s are names of nodes on the given layer. The current implementation requires that all nodes of the layer be present in the list and occur exactly once. The list specifies the left-to-right ordering of nodes on the given layer.

Anything between a `#` and the end of the line is interpreted as a comment. Arbitrary white space and/or comments can occur before the {, between nodes, or on either side of a layer description.

Binary Space ParitItions in Plücker Space*

David M. Mount[1] and Fan-Tao Pu[2]

[1] Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland. mount@cs.umd.edu
[2] Department of Computer Science, University of Maryland, College Park, Maryland. ftpu@cs.umd.edu.

1 Introduction

One of the important potential applications of computational geometry is in the field of computer graphics. One challenging computational problem in computer graphics is that of rendering scenes with nearly photographic realism. A major distinction in lighting and shading models in computer graphics is between *local illumination models* and *global illumination models*. Local illumination models are available with most commercial graphics software. In such a model the color of a point on an object is modeled as a function of the local surface properties of the object and its relation to a typically small number of point light sources. The other objects of scene have no effect. In contrast, in global illumination models, the color of a point is determined by considering illumination both from direct light sources as well as indirect lighting from other surfaces in the environment. In some sense, there is no longer a distinction between objects and light sources, since every surface is a potential emitter of (indirect) light. Two physical-based methods dominate the field of global illumination. They are *ray tracing* [8] and *radiosity* [3].

In both of these methods, and important problem that arises is that of determining visibility relationships for a collection of polygonal objects in 3-space. In this paper we propose a data structure to aid in solving this problem. We call this data structure a *visibility map*. Intuitively, a visibility map can be thought of as a sort of computer-generated hologram. Think of this abstract hologram as a piece of transparent plastic. When a viewer looks through the hologram, he sees what appears to be a three-dimensional scene. In particular, this is achieved by associating each ray shot from the eye of the viewer through the hologram with the illumination information associated with this ray. We can remove the viewer and just think of the visibility map as a function that maps rays emanating from the hologram to illumination information.

We show that a visibility map can be represented as a binary space partition tree in projective 5-dimensional space, through the use of a transformation that maps lines in 3-space to points in projective 5-space, called *Plücker coordinates*. An important practical question is how to build such trees and how large they

* The support of the National Science Foundation under grant CCR–9712379 is gratefully acknowledged.

are. We present an implementation of an algorithm for constructing these trees, and we analyze their size empirically as a function of the number of polygons in the 3-dimensional scene. We also consider methods for pruning the tree and study the effectiveness of these techniques.

1.1 Visibility Maps

Let $\mathbf{P}$ be a set of points and $\mathbf{D}$ be a set of directional vectors. Let $\mathbf{I}$ denote an (application-dependent) domain called the *visibility information*. For example, in rendering applications, the visibility information might be the color of an object, in radiosity applications it might be the radiance of a point. A *visibility map* $\boldsymbol{M}$ is a function

$$\boldsymbol{M} : \ D_M \mapsto \mathbf{I} \cup \{\Lambda\},$$

where D_M, the domain of $\boldsymbol{M}$ is a subset of $\mathbf{P} \times \mathbf{D}$. Given a point p in space and a direction $\boldsymbol{d}$, $\boldsymbol{M}(p, \boldsymbol{d})$ returns the visibility information arriving at the point p along the direction $-\boldsymbol{d}$. Intuitively, Λ is a special value, called the *null visibility information*, which is returned for rays that hit no objects.

In our application, a visibility map will not necessarily record visibility information coming from every possible object in the scene. A visibility map may have some region of three-dimensional space, called a *scope*, implicitly associated with it. The map records visibility information within the scope. For example, referring to Fig. 1, let $\boldsymbol{M}$ be the visibility map whose domain is the set of rays

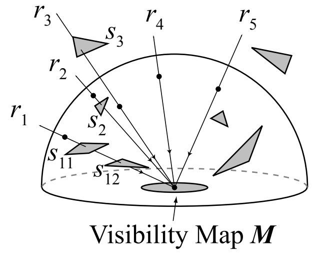


Fig. 1. Visibility map.

emanating upwards from the shaded disc and whose scope is the region enclosed in the hemisphere. The intersection of each ray with the hemisphere is indicated by a point on the ray. For rays r_1 and r_2, $\boldsymbol{M}(r_1)$ and $\boldsymbol{M}(r_2)$ will return visibility information of objects s_{12} and s_2, respectively. On the other hand, $\boldsymbol{M}(r_4)$ and $\boldsymbol{M}(r_5)$ both return Λ. Although ray r_3 intersects the object s_3, $\boldsymbol{M}(r_3)$ returns Λ because the intersection with object s_3 is outside of $\boldsymbol{M}$'s scope.

1.2 Line Geometry and Plücker Coordinates

Lines in three-dimensional space play an important role in computer graphics. When rendering a patch, we need to know the radiance of every point on the patch. Radiance arrives from every possible direction. To describe radiance in a faithful manner we can define a function that maps each point and each directional vector at the point to a radiance value arriving from this direction. If the patch is planar, it is clear that the domain of this function is isomorphic to the set of lines passing through this patch. A concise and elegant representation of lines in three-dimensional space is important for dealing with such infinite sets of lines. Plücker introduced a method of referencing lines as sets of coordinates, called *Plücker coordinates* [17], which maps a line in projective three-dimensional space to a point in projective five-dimensional space. This creates a new geometry where lines in three-dimensional space are the points of the new geometry.

To define Plücker coordinates, we first define the meet and join operations of two linear subspaces. The *meet* of two subspaces is defined to be their maximum common intersection and the *join* of two subspaces is defined to be the minimum space enclosing both subspaces. For example, in general position, a line meets a plane at a point and a line joins a point into a plane.

Let us consider projective three-dimensional space with homogeneous coordinates. We will present a derivation for line coordinates based on [20] and [10]. Let ℓ be a line and let x and y be two distinct points on ℓ and let ξ and ψ be two distinct planes that contain ℓ. Thus ℓ is the join of x and y and the meet of ξ and ψ. Assume the coordinates for points x, y and planes ξ, ψ are $[x_0, x_1, x_2, x_3]$, $[y_0, y_1, y_2, y_3]$, $[\xi_0, \xi_1, \xi_2, \xi_3]$ and $[\psi_0, \psi_1, \psi_2, \psi_3]$, respectively. Since points x and y both lie on planes ξ and ψ, we have following equations:

$$\begin{cases} \xi_0 x_0 + \xi_1 x_1 + \xi_2 x_2 + \xi_3 x_3 = 0 \\ \xi_0 y_0 + \xi_1 y_1 + \xi_2 y_2 + \xi_3 y_3 = 0 \\ \psi_0 x_0 + \psi_1 x_1 + \psi_2 x_2 + \psi_3 x_3 = 0 \\ \psi_0 y_0 + \psi_1 y_1 + \psi_2 y_2 + \psi_3 y_3 = 0 \end{cases} . \tag{1}$$

Let π_{ij} denote $x_i y_j - x_j y_i$. Eliminating the ξ_0, ξ_1, ξ_2, ξ_3, one at a time, from the first two equations above, we have equations

$$\begin{cases} 0\xi_0 + \pi_{10}\xi_1 + \pi_{20}\xi_2 + \pi_{30}\xi_3 = 0 \\ \pi_{01}\xi_0 + 0\xi_1 + \pi_{21}\xi_2 + \pi_{31}\xi_3 = 0 \\ \pi_{02}\xi_0 + \pi_{12}\xi_1 + 0\xi_2 + \pi_{32}\xi_3 = 0 \\ \pi_{03}\xi_0 + \pi_{13}\xi_1 + \pi_{23}\xi_2 + 0\xi_3 = 0 \end{cases} . \tag{2}$$

There are 16 π_{ij}'s. From the definition of π_{ij}, we know that $\pi_{ij} = -\pi_{ji}$ and $\pi_{ii} = 0$. Therefore, only six different values of π_{ij} determine the remainder. Let these six numbers be π_{01}, π_{02}, π_{03}, π_{23}, π_{31}, and π_{12}. These are called the *Plücker coordinates* of the line ℓ. The six Plücker coordinates derived from different pairs of points of a lines only differ by a non-zero constant factor. Thus the Plücker coordinates of a line are homogeneous coordinates of projective five-dimensional space, which is also called *Plücker space.* We will use the notation

$$\boldsymbol{\pi}(\ell) = [\pi_{01}, \pi_{02}, \pi_{03}, \pi_{23}, \pi_{31}, \pi_{12}]$$

to denote the Plücker coordinates of line ℓ. Since the system of homogeneous (linear) equations (2) has a nontrivial solution, the determinant

$$\begin{vmatrix} 0 & \pi_{10} & \pi_{20} & \pi_{30} \\ \pi_{01} & 0 & \pi_{21} & \pi_{31} \\ \pi_{02} & \pi_{12} & 0 & \pi_{32} \\ \pi_{03} & \pi_{13} & \pi_{23} & 0 \end{vmatrix}$$

vanishes. This gives the following constraint on the π_{ij}'s,

$$\pi_{01}\pi_{23} + \pi_{02}\pi_{31} + \pi_{03}\pi_{12} = 0. \tag{3}$$

The set of points satisfying this equation are said to lie on the *Grassmann manifold*.

Define a binary operator, *cross product* [9] ($\times$), on two Plücker points $\boldsymbol{\pi}(\ell_1)$ and $\boldsymbol{\pi}(\ell_2)$ as

$$\boldsymbol{\pi}(\ell_1) \times \boldsymbol{\pi}(\ell_2) = \pi^1_{01}\pi^2_{23} + \pi^1_{02}\pi^2_{31} + \pi^1_{03}\pi^2_{12} + \pi^1_{23}\pi^2_{01} + \pi^1_{31}\pi^2_{02} + \pi^1_{12}\pi^2_{03}. \tag{4}$$

The concept of *directed line*, line with one direction, is used to resolve the ambiguity of two opposite directions a line represents. The homogeneity condition for Plücker coordinates is modified to allow multiplication by any strictly positive constant. This operator returns 0 if ℓ_1 and ℓ_2 are incident, and otherwise its sign can be used to determine the relative orientation of the (directed) lines.

1.3 Binary Space Partitions

The *binary space partition* (BSP) tree [6] is an example of a data structure based on a recursive hierarchical space partitioning. A BSP tree is a binary tree, which encodes a hierarchical subdivision of d-dimensional space. The cell associated with each node v of a BSP tree is a convex polytope R_v. Each internal node v in a BSP tree is associated with a $(d-1)$-dimensional *cutting hyperplane* H_v. Let H_v^+ and H_v^- denote the two halfspaces introduced by hyperplane H_v. Then cutting hyperplane H_v cuts polytope R_v into two polytopes $R_v \cap H_v^+$ and $R_v \cap H_v^-$ which are associated with the left and right subtrees of node v, respectively.

The splitting procedure is performed recursively at each node until either all objects are separated or some application-dependent termination conditions are satisfied. Such termination conditions may be designed to avoid an excessive fragmentation of object or to bound the maximum tree depth. A node v in a BSP tree stores the objects that intersect the interior of the polytope R_v. If the cutting hyperplane H_v intersects an object, the object is split by H_v and each portion will be stored in the corresponding subtree.

There is no special rule for selecting the cutting hyperplanes for a BSP tree. However, the choice of cutting hyperplanes affects the size and maximum depth of the BSP tree and the number of object fragments that arise. Several works [1,2,4,14,15] have been devoted to the problem of selecting the cutting hyperplanes so as to minimize the complexity of the resulting BSP tree. For example, if there is a facet of an object lying on a hyperplane which does not intersect the interior of any other objects of this node, then this is a good candidate for the cutting hyperplane.

2 Data Structures for Visibility Maps

Before discussing the representation of visibility maps, we begin by discussing how to represent complex functions of line-space by simpler piecewise functions. Henceforth, we assume that lines in three-dimensional space are represented as directed lines using signed Plücker coordinates.

Let $\boldsymbol{\pi}(\ell) = [\ell_{01}, \ell_{02}, \ell_{03}, \ell_{23}, \ell_{31}, \ell_{12}]$ denote the Plücker coordinates of a directed line associated with some ray in the domain of the visibility map. Let $\boldsymbol{\pi}(e) = [e_{01}, e_{02}, e_{03}, e_{23}, e_{31}, e_{12}]$ denote the Plücker coordinates of the line supporting an edge of some convex patch in the scene. For a discontinuity to occur when ℓ intersects e, it must be that these two directed lines are incident, implying that

$$(\boldsymbol{\pi}(\ell) \times \boldsymbol{\pi}(e)) = \ell_{01}e_{23} + \ell_{02}e_{31} + \ell_{03}e_{12} + \ell_{23}e_{01} + \ell_{31}e_{02} + \ell_{12}e_{03} = 0.$$

This is a linear equation in $\boldsymbol{\pi}(\ell)$. We can use the sign of the orientation to determine whether ℓ passes to the left or right of e. Given a convex polygonal patch, P, let $\ell_{e_1}, \ell_{e_2}, \ldots, \ell_{e_m}$ denote the directed lines supporting the counterclockwise oriented edges of the patch. Then ℓ intersects, or *stabs*, P if all of these orientations are of the same sign, that is, if

$$\begin{array}{cc} \boldsymbol{\pi}(\ell) \times \boldsymbol{\pi}(\ell_{e_i}) \geq 0 & \text{for } 1 \leq i \leq m \\ \text{or} & \\ \boldsymbol{\pi}(\ell) \times \boldsymbol{\pi}(\ell_{e_i}) \leq 0 & \text{for } 1 \leq i \leq m. \end{array}$$

Thus the set of lines in three-dimensional space that stab a convex polygonal patch from one side or the other is a closed polyhedron in the projective five-dimensional space or Plücker space. See Fig. 2.

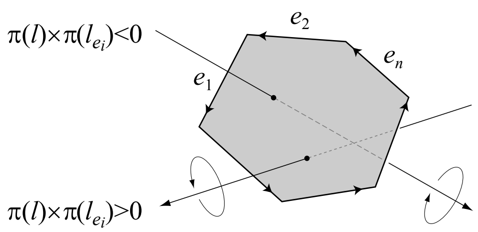


Fig. 2. The orientation between a patch and its stabbing line.

2.1 Plücker Space Partition Trees

Suppose that we want to represent the visibility map associated with some convex polygonal patch P. Let us assume that we are interested in visibility information arriving from only one side of P, indicated by an outward pointing normal vector. Take the set of directional vectors for the map to be the set of vectors whose angle with respect to P's normal vector is at most $\pi/2$. Each point on P and each directional vector corresponds to a directed line passing through the point and having this direction. As the point and/or direction vary continuously, the visibility information varies continuously as well, until the visibility ray strikes a different object of the scene. Thus discontinuities arise only when the line supporting the visibility ray intersects an object edge. Hence the visibility map is a piecewise continuous function where discontinuities occur along hyperplanes in Plücker space. We can think of this function as a subdivision of five-dimensional Plücker space, where each cell of the subdivision is associated with one visibility information function. We will concentrate on showing how to represent this subdivision through the use of a hierarchical space partition, which we call a *Plücker space partition tree* or *PSP tree* for short.

Recall the definition of a binary space partition from the Section 1. The space to be partitioned is Plücker space. Each node is implicitly associated with a convex polyhedral region of space. The root of the PSP tree implicitly represents all of Plücker space. (However we will modify this below.) Each internal node of the PSP tree is associated with a directed *splitting line* in three-dimensional space, or equivalently, a four-dimensional splitting hyperplane in the Plücker space. Each internal node has two children, one for lines positively oriented with respect to the splitting line and the other negatively oriented.

We will be storing visibility maps for axis-aligned rectangles in three-dimensional space. Each PSP tree will be implicitly associated with the set of directed lines passing through an axis-aligned rectangular "window" in three-dimensional space. Let ℓ_1, ℓ_2, ℓ_3, and ℓ_4 denote the directed lines supporting the edges of this rectangle, oriented counterclockwise around the rectangle so that the directed lines passing through the rectangle are positively oriented with respect to these lines. These four lines correspond to four halfspaces in Plücker space. Since we will only be interested in directed lines that lie in this region of Plücker space, rather than storing these four lines in the PSP tree (e.g. as the top four nodes), we store them separately as a *header* for the PSP tree.

The algorithm for constructing the rest of the Plücker space partition tree is as follows. Starting with the root, we insert each of the lines that bounds each of the three-dimensional patches of the scene as splitting lines into the PSP tree. For example, if we assume that all of the polygonal patches from the scene are convex, then we could insert the directed lines supporting the edges of each polygonal patch into the PSP tree one by one. The insertion of each splitting line corresponds to the insertion of a four-dimensional hyperplane into the tree. This is done done by an appropriate generalization of the standard insertion scheme for standard BSP trees (see, for example, Patterson and Yao [14]) but in dimension five. Each leaf of the final tree corresponds to a set of directed lines that are equally oriented with respect to all the edges of the patches and thus stab the same set of polygonal patches. Hence, all the lines in this cell share the same visibility information function.

General Structure Each node in a PSP tree, called a *PSPNode*, denotes a region in Plücker space. This region may be further subdivided by a Plücker hyperplane into two subregions. The subregion resides on the positive side of the cutting hyperplane is represented by its positive child and the one on the negative side is represented by its negative child. A node without a cutting hyperplane is a leaf node. Every internal node in this data structure has two children. Since for rendering we are only interested in the Plücker regions represented by leaf nodes of the tree, we adapted the idea of a *threaded tree* [11] by chaining all leaf nodes into a doubly linked list. This list provides a direct way to enumerate these leaf nodes.

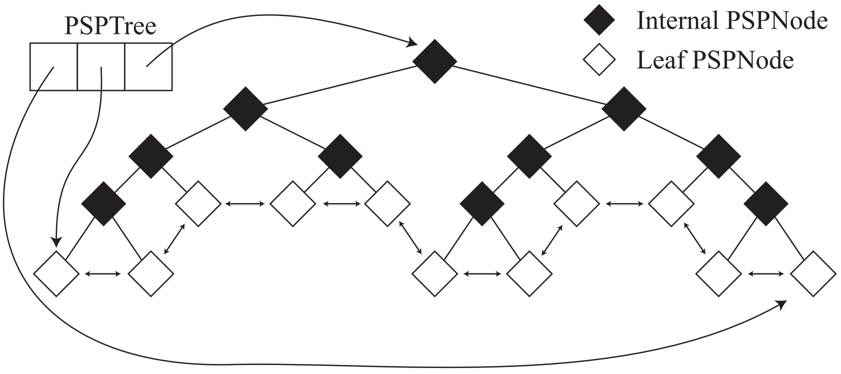


Fig. 3. Structure of Plücker space partition tree.

Insertion A Plücker space partition tree is constructed by inserting the polygons of the scene one-by-one. Each polygon is inserted into a Plücker space partition tree by inserting all of the supporting lines for its edges. The basic update of the Plücker space partition tree is the insertion of a line (as a Plücker hyperplane) into the tree. Recall that each node of the Plücker space partition tree represents a convex polyhedral region of Plücker space. A line is inserted recursively into a node's two children provided that the line's corresponding Plücker hyperplane cuts the region represented by the node. The recursion stops when the node is a leaf. This line (the Plücker hyperplane) becomes the leaf node's cutting plane and two new leaf nodes are created as its children. This algorithm is given in Fig. 4.

```
Insert(history,iplane)
    if (is_leaf)
        make iplane as its cutting_plane
        create two leaf nodes as its children
    else
        PosPolytope = Polytope(history, halfspace(cutting_plane,+1))
        NegPolytope = Polytope(history, halfspace(cutting_plane,-1))
        if (intersect(PosPolytope, iplane))
            new_history = union(history, halfspace(cutting_plane,+1))
            Insert(new_history, iplane)
        endif
        if (intersect(NegPolytope, iplane))
            new_history = union(history, halfspace(cutting_plane,-1))
            Insert(new_history, iplane)
        endif
    endif
end Insert
```

Fig. 4. Insertion.

3 Trimming the Plücker Space Partition Tree

The size of a Plücker space partition tree grows rapidly as a function of the number of triangles. See Section 4.3. In fact, it is too large to deal practically with any non-trivial scene. We investigate methods for eliminating redundant nodes from the tree. We consider two simple ways in which redundant nodes may arise: (1) Two siblings encode the same visibility information. (2) Leaf nodes do not encode any line. The former suggests merging sibling leaf nodes having the same visibility information. The latter suggests deleting leaf nodes which do not interest Grassmann manifold. They are described next.

3.1 Merging by Constant Visibility Information

When two sibling leaf nodes carry the same *constant visibility information*, they can be merged by deleting them and making their parent a new leaf node. Under the assumption we have made earlier, constant visibility information means either transparency (no polygon is stabbed by the lines of the node) or the same single polygon is stabbed from the same direction. If sibling leaf nodes stab the same set of polygons and the size of the set is more than two, then they cannot be merged because the visible polygon in the set is view-dependent and may differ.

The merging process is performed by a recursive postorder traversal of the Plücker space partition tree. The recursion stops at a leaf and returns the number of stabbing polygons. Each internal node compares the values returned from its two children. If both children are leaves and either they stab no polygon, or they both stab the same single polygon then the internal node merges information of its children and deletes them. See Fig. 5.

```
ConstVisInfoMerge(node)
    if (is_leaf(node))
        return number of stabbed polygon
    else
        num_pos_stab = ConstVisInfoMerge(positive_child)
        num_neg_stab = ConstVisInfoMerge(negative_child)
        if ((num_pos_stab==0)&&(num_neg_stab==0))
            merge children
        else if ((num_pos_stab==1)&&(num_neg_stab==1))
            if the two stabbing polygons are the same then
                merge children
            endif
        endif
    endif
End
```

Fig. 5. Constant visibility information merging algorithm.

3.2 Merging by Empty Grassmann Intersection

Recall that a Plücker polyhedron does not represent any line in three-dimensional space if it does not intersect the Grassmann manifold. Any leaf node whose associated region does not intersect the Grassmann manifold may be deleted as redundant.

To test whether a given node in the Plücker space partition tree intersects the Grassmann manifold we consider the polyhedron it represents. In [18] we showed that such a polyhedron does not intersect the Grassmann manifold if and only if its one-dimensional boundary (1-skeleton) does not intersect the manifold. Therefore, the emptiness test involves the following steps:

1. Enumerate the vertices of the Plücker region in projective five-dimensional space as well as their adjacency relation,
2. Interpolate every pair of adjacent vertices to construct the 1-skeleton, and
3. Test whether each edge of the 1-skeleton intersects the Grassmann manifold, and return empty-intersection if no intersection is found.

The basic structure of the merging algorithm is similar to the one for the constant visibility information merge, see Fig. 6. We discuss enumeration below.

```
EmptyGManifoldMerge(node)
    if (is_leaf(node))
        return EmptinessTest(node)
    else
        pos_emptiness = EmptyGManifoldMerge(positive_child)
        neg_emptiness = EmptyGManifoldMerge(negative_child)
        if ((pos_emptiness==EMPTY) && (neg_emptiness==EMPTY))
            merge children
            return EMPTY
        else if ((pos_emptiness==EMPTY) || (neg_emptiness==EMPTY))
            merge children
            return NON_EMPTY
        else
            return NON_EMPTY
        endif
    endif
End
```

Fig. 6. Trimming empty Grassmann intersection algorithm.

3.3 Auxiliary Routines

Two nontrivial tasks have been ignored in the above description. One is used when inserting a hyperplane into a Plücker space partition tree and another is used when testing whether a node in Plücker space partition tree does not intersect the Grassmann manifold. They are stated more formally below. Let C be a given Plücker polyhedron defined as the intersection of a set of Plücker halfspaces. Note that the polyhedron C is a cone in Euclidean six-dimensional space.

1. For a Plücker hyperplane h, determine whether h intersects C. (See Section 2.1.)
2. What are the extremal rays of cone C? (See Section 3.2)

The first is a linear-programming problem and the second is a polytope enumeration problem.

A convex polyhedron can be defined in two ways. It can be described either as an intersection of a set of halfspaces, or as a convex combination of its vertices and/or extremal rays. Note that extremal rays arise if the polyhedron is not closed, i.e., unbounded. The former is called the *H-representation* and the latter is called the *V-representation* [7] of the polyhedron. An application program called *cdd/cdd+* by Fukuda [7] utilizes linear programming techniques to implement an algorithm called the *double description method* [13], which converts one representation to another for a given polyhedron. We have adapted and modified the routines in the cdd/cdd+ program for solving the following tasks:

1. Find the existence of a feasible region (except the origin) of a given set of homogeneous six-dimensional inequalities, and
2. Find the extremal rays of a cone defined by a set of homogeneous six-dimensional inequalities.

Note that we perform computation for projective five-dimensional space in Euclidean six-dimensional space.

4 Experiments

To explore various properties of the Plücker space partition tree we ran a number of experiments which generates Plücker space partition trees from randomly generated inputs and than measure these properties. Each experiments involves generating 100 different random inputs, each consisting of a set of nonintersecting triangles in three-dimensional space. The generating program is described in Section 4.1. In Section 4.2, we discuss the result of these experiments.

4.1 Random Triangle Generator

A small program which generates triangles in a bounding box randomly is used as data generator for the experiment. This program is given the number of triangles n, and outputs n nonintersecting triangles of various sizes distributed through out a given bounding box.

To generate nonintersecting triangles we first decompose space hierarchically using a k-d tree [19]. The k-d tree splitting rule guides this distributing. A recursive algorithm directs the process. Let n be the number of triangles to be generated in a bounding box b. If $n > 1$ then an axis is chosen at random. A plane h perpendicular to this axis is then randomly generated to split the bounding box b into boxes b' and b''. The number n is partitioned according to the ratio between b' and b'' into n' and n'' ($n = n' + n''$), respectively. The process is recursively applied separately to box b' with number of triangle n', and to box b'' with number of triangles n''. If $n = 1$ then a triangle is generated by randomly picking three points on the walls of box b.

4.2 Results

Plücker Space Partition Trees without Pruning To explore the properties of the Plücker space partition tree, we build Plücker space partition trees for scenes consisting of a number of triangles ranging from 1 to 14. Each input size is tested over 100 randomly generated scenes. We presents the plots for the size of tree (Fig. 7), the height of tree (Fig. 8) and the time (measure in seconds) for building the tree (Fig. 9). We use the average number of leaf nodes for showing the size of Plücker space partition tree. Leaf nodes are further classified as being either transparent (stabbing no triangles) or not transparent as denoted by “Trans.” and “Non-Trans.” in the legend of Fig. 7.

Plücker Space Partition Trees with Pruning The effect of trimming Plücker space partition tree is presented in Fig. 10 in regular scale and Fig. 11 in logarithm scale for comparing the average number of nodes in a tree when without trimming, with merging by constant visibility information only, with merging by empty Grassmann intersection only, or with both merging heuristics. They are denoted as "No TRIM", "C TRIM", "G TRIM" and "CG TRIM" in the legend box, respectively.

4.3 Analysis

The Plücker space partition tree is a variation of binary space partition tree in higher (five) dimensions. We know of no nontrivial complexity analysis of binary space partition trees in these dimensions. The Plücker space partition tree defines a subdivision of space which is a coarsening of the arrangement of the set of hyperplanes of the five-dimensional space generated by all the lines that were inserted into the tree. It follows from standard results in combinatorial geometry that, the worst case complexity of Plücker space partition tree is $O(n^5)$ [5] where n is the number of lines. McKenna and O'Rourke [12] established that the number of distinct isotopy classes for n given lines is $O(n^4\alpha(n))$. The number of leaves in the Plücker space partition tree can generally exceed this amount, since some leaves of the tree might not intersect the Grassmann manifold, and hence may not correspond to any isotopy class. However, after pruning away leaf cells that do not intersect the Grassmann manifold, this bound should apply in our case. Pellegrini and Shor [16] showed that for a given set of convex polyhedra, the complexity of lines that stabs these polyhedra, a subset of isotopy classes induced by these lines, is $O(n^3 2^{c\sqrt{\log n}})$, where n is the number of facets of the given set of polyhedra and c is a constant. However, this bound does not apply immediately, because the Plücker space partition tree computes leaf cells that might not intersect any of the objects.

It follows that the size of a Plücker space partition tree is at most $O(n^5)$, where n is the number of input triangles. Let $s(n)$ be the size of a Plücker space partition tree of input size n. We conjecture that the size of the Plücker space partition tree is of the following form

$$s(n) \approx an^c \qquad \text{or equivalently} \qquad \log s \approx c \log n + \log a,$$

for some constants a and c. Based on this conjecture, the values of constants a and c can be estimated from our experimental results by fitting a line to a log-log scale plot of tree-size versus n.

We consider the un-trimmed Plücker space partition tree first. Figure 12 (a) shows the curve obtained by connecting the 14 points whose x-coordinate is the logarithm of the number of triangles and y-coordinate is the logarithm of the average tree size. We deleted the leftmost two points (since low values are less likely to provide good estimation of asymptotic values) and applied a least squares line fit. See Fig. 12 (b). We found that the line of fit is $(\ln s) = 5.17(\ln n) - 1.16$, i.e.,

$$s(n) \approx 0.315 n^{5.17}. \tag{5}$$

The exponent is quite close to 5, which suggests that the $O(n^5)$ upper bound may be tight. The fact that the exponent exceeds 5 is likely due to the fact that the input sizes are not large enough to accurately gauge asymptotic growth rate. Of course, the input sizes are quite small, and so our extrapolations should be taken with a grain of salt. Next is the trimmed (by both heuristic methods) Plücker space partition tree. Figure 13 (a) presents the curve by connecting points of the average trimmed size of Plücker space partition tree versus the number of triangles in log-log scale. Again, after deleting the leftmost two points in this figure, least square fit returns the line, see Fig. 13 (b), $(\ln s) = 4.31(\ln n) - 0.129$, i.e.,

$$s(n) = 0.879n^{4.31}. \tag{6}$$

Again, extrapolations to asymptotic bounds is risky with such small input sizes, but it does bear a similarity to the $O(n^4\alpha(n))$ bound of McKenna and O'Rourke.

In summary, (5) and (6) suggest the complexities for the un-trimmed and trimmed Plücker space partition tree are roughly $O(n^{5.17})$ and $O(n^{4.31})$, respectively. These are similar the upper bounds on the complexity of an arrangement in five-dimensional space and the complexity of isotopy classes, respectively.

References

1. Pankaj K. Agarwal, Leonidas J. Guibas, T. M. Murali, and Jeffrey Scott Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.
2. Pankaj K. Agarwal, T. Murali, and J. Vitter. Practical techniques for constructing binary space partitions for orthogonal rectangles. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 382–384, 1997.
3. Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis.* Academic Press Professional, Cambridge, Mass., 1993.
4. M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane. *Comput. Geom. Theory Appl.*, 8:317–333, 1997.
5. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Heidelberg, West Germany, 1987.
6. H. Fuchs, M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):124–133, July 1980. Proc. SIGGRAPH '80.
7. Komei Fukuda and Alain Prodon. Double description method revisited. *Lecture Notes in Computer Science*, 1120, 1996. The URL for cdd+ package is: http://www.ifor.math.ethz.ch/ifor/staff/fukuda/cdd_home/cdd.html.
8. Andrew S. Glassner, editor. *An Introduction to Ray Tracing.* Academic Press, San Diego, CA, 1989.
9. Václav Hlavatý. *Differential Line Geometry.* P. Noordhoff Ltd., Groningen, Holland, 1953. Translated by Harry Levy.
10. C. M. Jessop. *A Treatise of Line Complex.* Combridge, Combridge, England, 1903.
11. D. E. Knuth. *The Art of Computer Programming.* Addison Wesley, second edition, 1978.

12. M. McKenna and J. O'Rourke. Arrangements of lines in 3-space: a data structure with applications. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 371–380, 1988.
13. T. S. Motzkin, H. Raiffa, G. L. Tompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tuker, editors, *Contributions to theory of games*, volume 2. Princeton University Press, Princeton, RI, 1953.
14. M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete Comput. Geom.*, 5:485–503, 1990.
15. M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.
16. M. Pellegrini and P. Shor. Finding stabbing lines in 3-space. *Discrete Comput. Geom.*, 8:191–208, 1992.
17. J. Plücker. *Neue Geometrie des Raumes*. Leipzig, 1868.
18. F.-T. Pu. *Data structures for global illumination computation and visibility queries in 3-space*. PhD thesis, Department of Computer Science, University of Maryland, College Park, Maryland, March 1998.
19. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
20. D. M. Y. Sommerville. *Analytic Geometry of Three Dimensions*. Cambridge University Press, Cambridge, England, 1934.

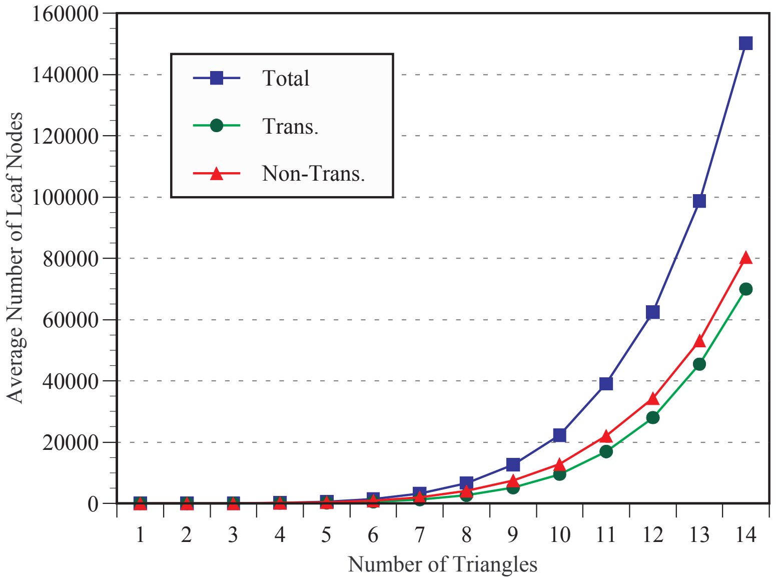


Fig. 7. Leaf nodes.

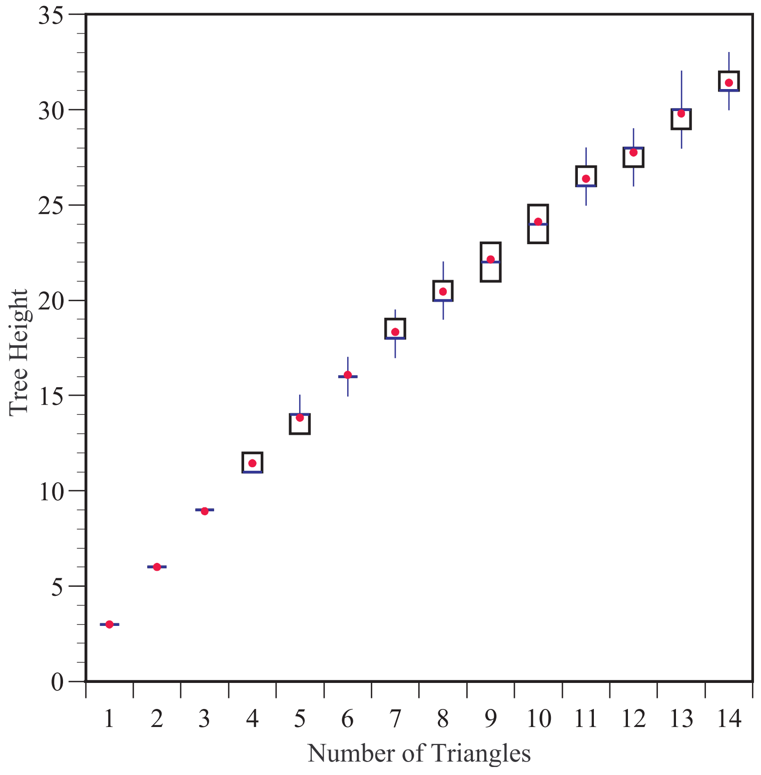


Fig. 8. The height of Plücker space partition tree.

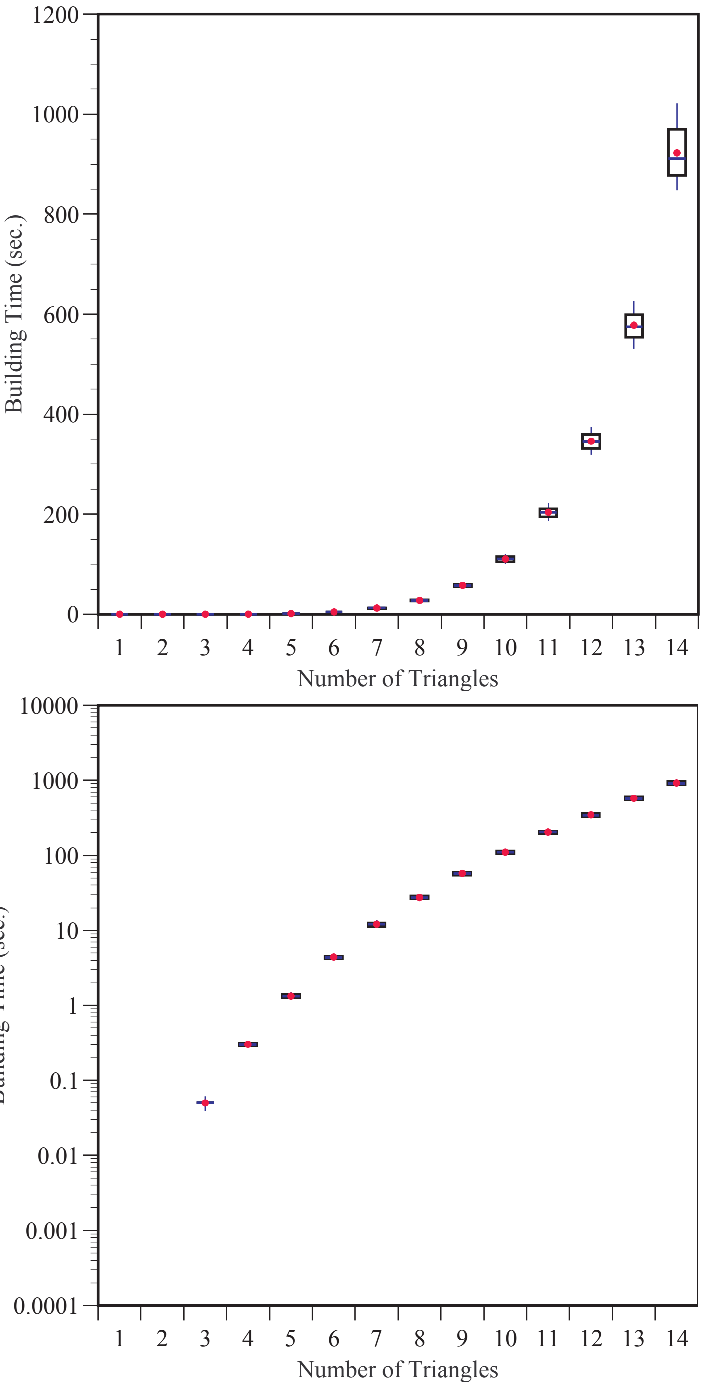


Fig. 9. Building time of Plücker space partition tree.

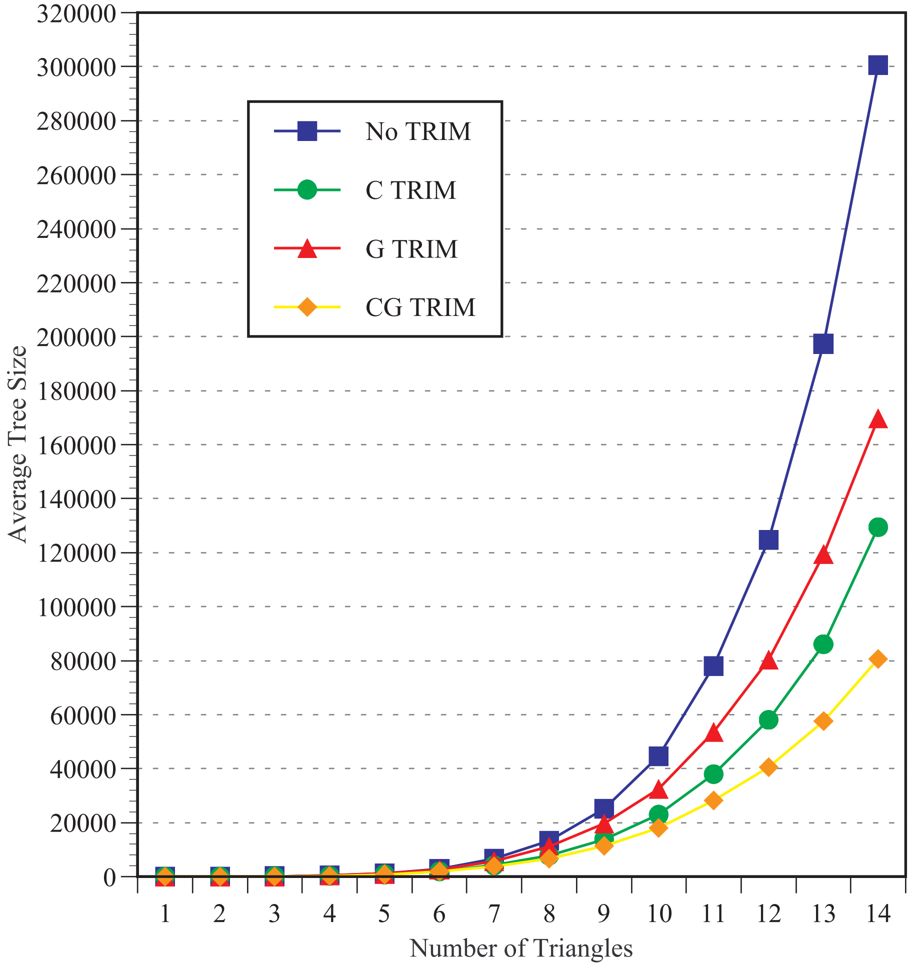


Fig. 10. The average tree size for different trimming.

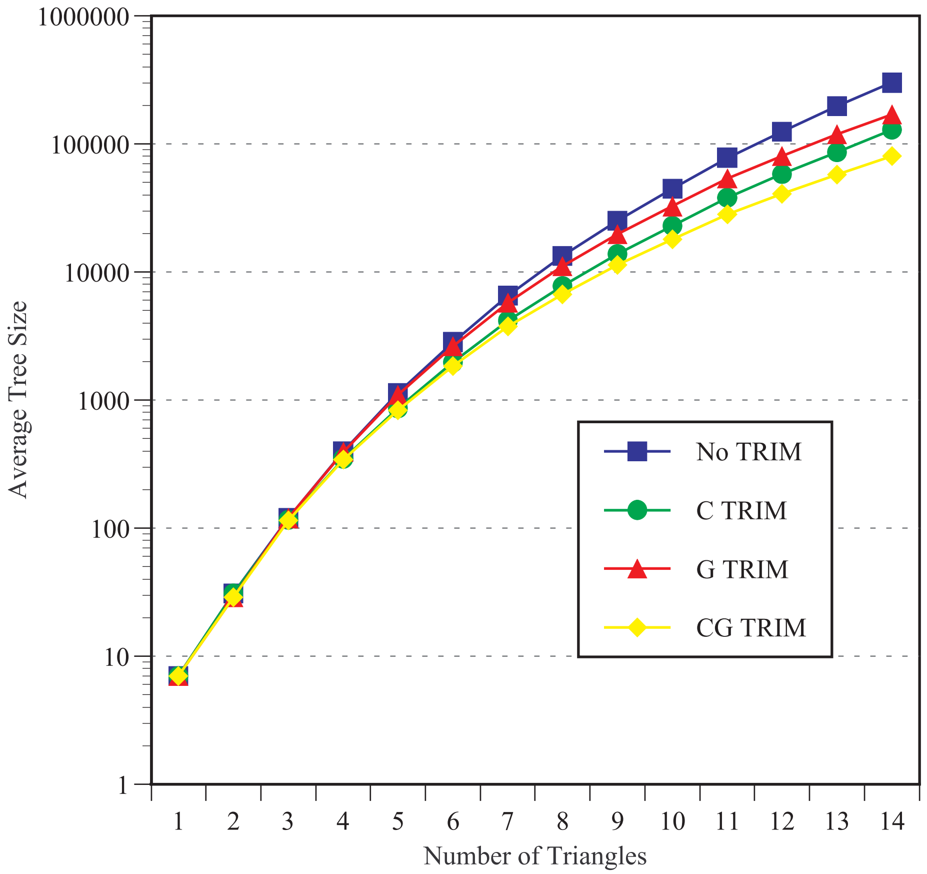


Fig. 11. The average tree size for different trimming (log scale).

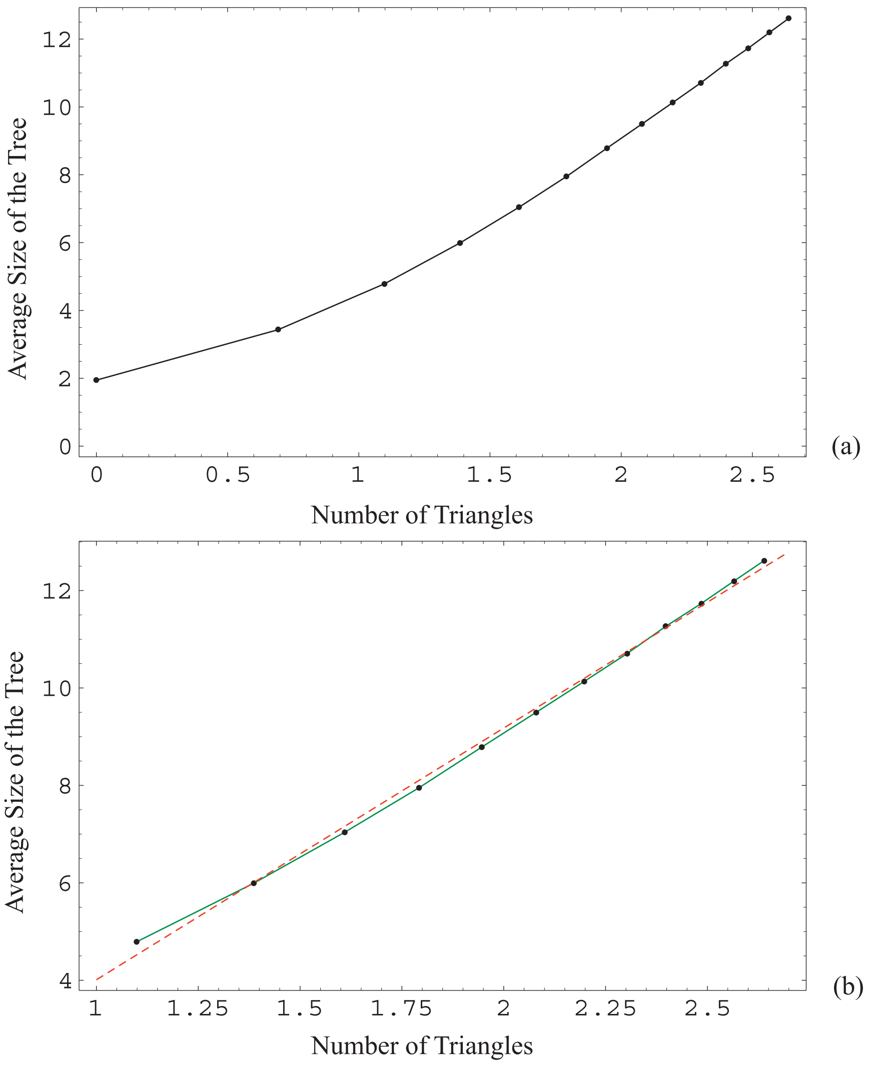


Fig. 12. Average size of un-trimmed tree in log-log scale.

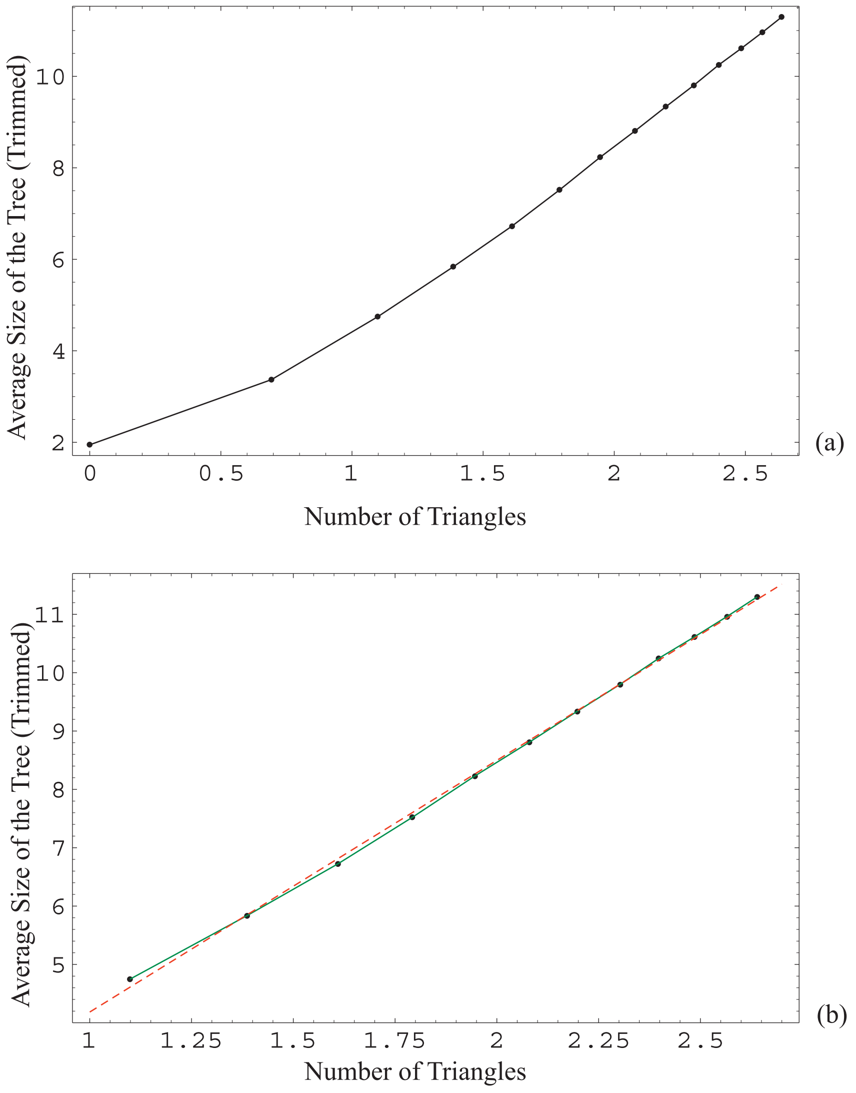


Fig. 13. Average size of trimmed tree in log-log scale.

Practical Point-in-Polygon Tests Using CSG Representations of Polygons

Robert J. Walker and Jack Snoeyink

UBC Dept. of Computer Science
201–2366 Main Mall
Vancouver, BC Canada V6T 1Z4
{walker, snoeyink}@cs.ubc.ca

Abstract. We investigate the use of a constructive solid geometry (CSG) representation in testing if a query point falls inside a polygon; in particular, we use a CSG tree whose leaves correspond to halfplanes defined by edges and whose internal nodes are intersections or unions of the regions defined by their subtrees. By preprocessing polygons into this representation, we obtain a linear-space data structure for point-in-polygon tests that has a tight inner loop that can prune unnecessary edge tests during evaluation. We experiment with opportunities to optimize the pruning by permuting children of nodes. The resulting test is less memory-intensive than grid methods and faster than existing one-shot methods. It also extends to ray-shooting in 3-space.

1 Introduction

Point-in-polygon tests are common in computer graphics and are often used in raytracing where many points must be tested against a given polygon. The goal is to minimize the average cost per test in a practical situation. Unlike in standard computational geometry situations, we are concerned with the absolute performance of algorithms on polygons that will occur for such an application, and not so much on their asymptotic performance. Practical situations tend not to contain worst-case polygons, and the number of edges dealt with have an upper limit.

Practical point-in-polygon test algorithms can be classified into two major categories: those that utilize a vertex-list representation of the polygon as-is ("one-shot" methods), and those that perform preprocessing on the vertex-list representation, possibly transforming it into a different representation, before using the processed structure to perform multiple tests. The latter methods generally have a lower amortized cost when many tests will be performed on a particular polygon.

We discuss a new method utilizing preprocessing via a constructive solid geometry (CSG) tree representation of polygons. This method prunes unnecessary edge comparisons during the test, as described in Section 2. By permuting sibling nodes within the tree, as described in Section 3, we can attempt to optimize the pruning.

1.1 Existing Point-in-Polygon Tests

Haines [9] gives a thorough comparative treatment of existing point-in-polygon algorithms.

Haines tested different types of polygons: regular polygons, and random, possibly self-intersecting, polygons. His results indicate that the fastest one-shot method is the crossings test of Shimrat [11] as corrected by Hacker [8]. For algorithms that require a little preprocessing and extra storage, Haines found that Green's half-plane algorithm [7] and Spackman's algorithm [13] were fastest for random polygons with few sides, while the crossings test was faster for many sides. The hybrid half-plane test was fastest for regular polygons with few sides, and for many sides the inclusion test [10] was fastest. An exception occured when the ratio of bounding box area to polygon area is high, in which case the exterior edges algorithm was fastest. Haines' results also indicate that grid methods [1] were fastest when preprocessing and extra storage were available in abundance.

Haines assumed that all test points would first be clipped against the polygons' bounding boxes; in his test implementation, he approximated bounding boxes by generating the polygon vertices within the unit square. His tests were conducted on one particular architecture. Details of these algorithms, test timings, and code are available in [9].

1.2 CSG Representation of Polygons

In a polygon, each bounding edge defines the boundary of two half-planes; orienting the edges counterclockwise about the polygon, we have that the half-plane to the left of the edge is considered to be inside the polygon, and the one to the right outside. The intersection and union of a set of such half-planes, performed in the correct order, defines the polygon and these half-planes are said to *support* the polygon. Two half-planes are illustrated in Figure 1. When testing a point against an edge, a Boolean value can be used to represent the truth of the statement: "The point lies inside." The full CSG representation is then interpreted as a Boolean formula for the whole polygon, as in Figure 2.

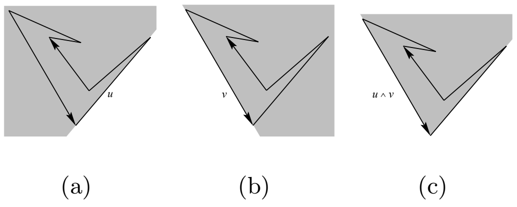


Fig. 1. *Halfplanes for two edges and their Boolean combination $u \wedge v$.*

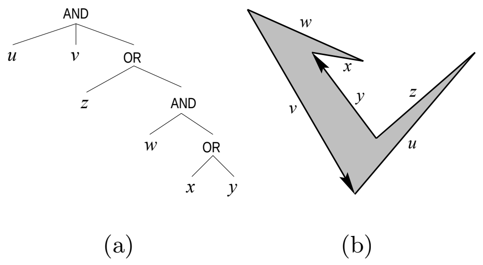


Fig. 2. *A tree for Boolean formula* $u \wedge v \wedge (w \wedge (x \vee y) \vee z)$ *that expresses the interior of the polygon (b)*

Dobkin et al. [6] give an algorithm to construct a CSG representation of a non-self-intersecting polygon. The resulting representation is in the form of a directed binary tree whose nodes contain either Boolean operators or half-plane representations. This tree may then be traversed to derive a monotone formula for the polygon—one in which every supporting half-plane appears exactly once, and no complementation is required. Thus, there are $n - 1$ logical operators appearing in the formula, and testing a point against such a structure will require $O(n)$ operations in the worst case: the hope is that the constants that occur in real cases will be small enough to make this method competitive in practical situations.

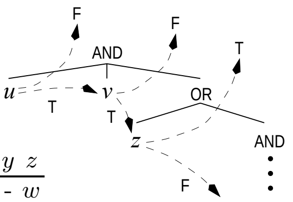
2 Point-in-Polygon Tests Using CSG Representations

Conceptually, we wish to test a given point against each edge of the polygon. The location of the point can then be determined by evaluating the Boolean formula represented by the CSG tree. We take advantage of the alternating structure of the logical operators, which can be seen in the CSG tree representation of Figure 2, to eliminate unnecessary tests before computing their results.

Specifically, as soon as a subtree that is the child of an AND node is known to be FALSE, we may stop testing the children of that node, and assign the Boolean value FALSE to that AND node. Likewise, as soon as a subtree that is the child of an OR node is known to be TRUE, we may stop testing the children of that node, and assign the Boolean value TRUE to that OR node. The base case is at the leaves where we assign TRUE if the testing point is to the left of the edge, and FALSE if it is to the right. When a Boolean value is selected for the root of the tree, a value of TRUE indicates that the point is inside the polygon and FALSE indicates outside.

2.1 Obtaining a Tight Inner Loop

If we fix the ordering of the children of each node in the CSG tree, we can decide in advance which polygon edge to test next after each edge is tested. This allows us to eliminate a costly selection process from the inner loop of the testing algorithm. The selection of the next edge is determined only from the Boolean result of testing the current edge. These choices can be encoded in a new data structure, the *edge-sequence graph* (ESG) (see Figure 3).



	u	v	w	x	y	z
F	-	-	-	y	-	w
T	v	z	x	-	-	-

Fig. 3. *Test paths through the tree of Figure 2 are represented by an* edge-sequence graph*, shown here as a table. The entire tree does not need to be tested; instead, "short-circuit" paths may be taken when certainty of the location of a point is achieved.*

An ESG is a data structure representing a single polygon; it contains two members: an array of edges defining the polygon, and the lookup array containing indices into the edge array. The lookup array is two dimensional: there are two integers in it for each member of the edge array. One integer indicates the index of the next edge to test if the current test yields the Boolean value FALSE, and the other is for TRUE. Arrays are assumed to be indexed from 0 to size $- 1$.

Given an ESG, testing a particular point against the corresponding polygon is straightforward. We begin by testing the point against the first edge in the edge array of the ESG; the next edge to test is determined by the appropriate index found in the lookup array. Such testing continues until the indicated next index is -1, in which case the current Boolean test result becomes the result for the full point-in-polygon test.

In the following algorithm, Points are represented by an x-y pair of Cartesian coordinates while Edges are represented by the oriented lines upon which they lie.

```
PointInPolygon( ESG esg, Point point) returns Boolean

Edge edge; Integer result; Integer index:= 0
do
   edge:= esg.edges[index]
   result:= Edge-PointLeftOf( edge, point)
   index:= esg.lookup[index][result]
until index= −1
return result
```

If edges are represented as lines in homogeneous coordinates, as they were in our implementation, then in 2-space the call to Edge-PointLeftOf() in line 6 would set *result* equal to

$$(edge.w + point.x \times edge.x + point.y \times edge.y \geq 0).$$

If strict containment within the polygon is required, the operator in this formula should be changed from "$\geq$" to "$>$". The PointInPolygon() procedure lends itself to optimization depending upon the implementation language and system architecture.

2.2 Experimental Results

The algorithm was tested in accordance with the methodology of Haines [9], with a few exceptions. Polygon vertices were randomly generated then scaled to fit the resulting polygon's bounding box to the unit square. Since the CSG representation can only cope with non-self-intersecting polygons, a simple edge-swapping algorithm was used to unfold the generated polygons prior to testing. All points to be tested against the polygons were generated pseudo-randomly, uniformly distributed across the polygon's bounding box.

Method	Variety	3	4	5	6	7	8	9	10	20	50	100	1000
angle sum	-	15.9	20.4	24.9	29.6	34.0	38.6	43.1	47.7	94.5	232.7	461.8	4656.8
barycentric	-	2.0	3.5	4.9	6.2	7.6	8.9	10.3	11.6	24.5	62.8	125.5	1252.1
crossings	-	1.9	1.8	1.9	1.9	2.1	2.2	2.4	2.5	3.9	7.9	14.5	127.5
crossings	multiply	1.3	1.5	1.6	1.8	1.9	2.1	2.3	2.4	4.0	8.3	15.2	132.5
crossings	winding	2.1	2.1	2.2	2.2	2.5	2.7	2.8	2.9	4.8	9.9	18.4	170.2
CSG	-	0.9	1.2	1.3	1.6	1.7	1.8	2.1	2.1	3.6	7.0	11.3	71.7
CSG	sorted	0.8	1.0	1.2	1.5	1.7	1.8	1.9	2.1	3.1	5.7	9.7	41.4
grid	100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.3
grid	20	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.2	1.3	1.5	3.1
half-plane	-	0.7	1.2	1.6	2.1	2.5	2.9	3.3	3.8	7.9	20.5	41.8	630.8
half-plane	sorted	0.7	1.1	1.4	1.8	2.2	2.5	3.0	3.3	7.0	17.3	35.0	600.6
Spackman	-	0.9	1.3	1.7	2.1	2.5	2.9	3.3	3.7	7.4	18.5	37.0	676.8
Spackman	sorted	0.8	1.2	1.6	2.0	2.4	2.8	3.2	3.6	7.2	18.1	35.9	540.8
trapezoid	100	1.0	1.1	1.1	1.1	1.1	1.2	1.3	1.3	1.7	2.5	3.4	14.7
trapezoid	20	1.1	1.2	1.3	1.4	1.4	1.5	1.6	1.7	2.3	4.1	6.3	39.9
Weiler	-	2.3	2.4	2.7	2.8	3.1	3.5	3.8	4.0	7.0	16.0	31.1	307.2
Weiler	winding	2.3	2.4	2.7	2.8	3.1	3.4	3.7	4.0	7.1	16.1	30.9	307.4

Table 1. *Average times for point-in-polygon tests (in microseconds), number of edges per polygon vs. method for random polygons.*

Each algorithm was tested to see how long it took to perform point-in-polygon tests on an identical set of 50 pseudo-randomly generated points against each of 50 pseudo-randomly generated polygons. Some tests were repeated and averaged to alleviate timing inaccuracies. Tests were performed on a Silicon Graphics Indy workstation with a 133 MHz IP22 CPU; all processes save system daemons were disabled during test execution to further minimize variability of measurements.

Method	Variety	3	4	5	6	7	8	9	10	20	50	100	1000
angle sum	-	15.5	20.2	25.1	29.7	34.5	39.5	44.0	48.5	95.3	235.0	471.0	4669.6
barycentric	-	2.2	3.7	5.0	6.4	7.6	8.8	10.1	11.3	23.5	59.6	120.3	1227.1
crossings	-	1.7	1.6	1.7	1.7	1.8	1.9	2.0	2.1	3.1	6.6	12.7	122.6
crossings	convex	1.7	1.6	1.6	1.6	1.6	1.7	1.8	1.8	2.5	5.0	9.2	86.3
crossings	multiply	1.3	1.4	1.6	1.7	1.8	1.9	2.0	2.1	3.4	7.0	12.8	123.5
crossings	winding	1.9	1.8	1.9	2.1	2.1	2.2	2.4	2.6	4.0	8.6	16.4	161.7
CSG	-	1.0	1.3	1.6	1.9	2.2	2.5	2.8	3.1	6.1	15.1	30.2	380.0
CSG	sorted	0.9	1.3	1.6	1.9	2.2	2.5	2.8	3.0	5.8	14.4	27.5	345.4
exterior	-	0.7	0.9	1.1	1.4	1.6	1.7	1.9	2.2	4.1	9.8	19.7	309.1
exterior	randomized	0.7	1.0	1.2	1.3	1.6	1.7	1.9	2.1	4.0	9.4	18.1	282.2
grid	100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
grid	20	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.5
half-plane	-	0.8	1.3	1.7	2.2	2.6	3.1	3.5	4.0	8.4	21.7	43.9	721.0
half-plane	convex, hybrid	0.7	1.1	1.4	1.8	2.1	2.5	2.8	3.2	6.5	16.2	33.1	519.0
half-plane	convex, sorted	0.7	0.9	1.1	1.3	1.5	1.6	1.8	2.0	3.7	8.6	17.0	311.1
half-plane	convex, sorted, hybrid	0.7	0.9	1.1	1.2	1.4	1.6	1.7	1.9	3.3	8.1	15.0	294.6
half-plane	sorted	0.7	1.1	1.4	1.6	1.9	2.2	2.5	2.8	5.4	13.5	26.9	550.0
inclusion	-	3.1	3.2	3.3	3.3	3.3	3.4	3.4	3.5	3.7	4.2	4.2	5.3
Spackman	-	0.9	1.4	1.8	2.2	2.5	2.9	3.2	3.6	7.0	17.2	34.3	574.2
Spackman	sorted	0.9	1.3	1.7	2.0	2.4	2.7	3.0	3.3	6.5	16.4	32.4	508.8
trapezoid	100	1.1	1.2	1.2	1.3	1.3	1.3	1.3	1.3	1.4	1.6	1.8	5.0
trapezoid	20	1.2	1.4	1.4	1.5	1.5	1.6	1.6	1.6	1.9	2.7	3.6	19.9
Weiler	-	2.1	2.2	2.4	2.6	2.9	3.1	3.4	3.7	6.5	15.2	29.9	308.6
Weiler	winding	2.1	2.1	2.4	2.6	2.8	3.2	3.4	3.6	6.5	15.4	30.4	305.4

Table 2. *Average times for point-in-polygon tests (in microseconds), number of edges per polygon vs. method for convex polygons only.*

The results for the various methods are shown in Table 1. Only the half-plane method is faster (14%) for triangles than the CSG method using the simply-sorted heuristic described in Section 3.3, and only the 100×100 grid method is equally fast for quadrilaterals. Only the memory-intensive grid and trapezoid methods outperform the CSG method as polygons grow large; the nearest other rival (the crossings method) is 208% more time consuming for a 1000 vertex polygon.

Tests were also performed for sets containing only randomly-generated convex polygons; the results are given in Table 2. The CSG method is not competitive for this restricted class of polygons when test points are uniformly distributed and confined to each polygon's bounding box.

2.3 Memory Requirements

There exists a tradeoff between time to perform point-in-polygon tests and space to store the data structures to support the algorithms. The memory requirements described in this section refer to final storage of the data structures used for the actual point-in-polygon tests; preprocessing requirements were not examined, and our implementation of the CSG method is not optimized for preprocessing performance or memory footprint.

The CSG method described in this paper requires an array of edges (each of which consists of three reals), an integer recording the number of edges, a number-of-edges $\times$ 2 array of indices into the edge array, and a single bit to indicate orientation of the polygon. This totals $3Rn + (1 + 2n)I + B$ bytes for an n edge polygon; in our implementation for a Silicon Graphics Indy architecture, double-precision reals, short integers, and a full byte for the single bit are used. The total memory usage is therefore $28n + 3$ bytes.

Haines' implementation of the $p \times q$ grid method uses a constant three integers, eight double-precision reals, and three pointers, plus $p+q+2$ doubles and pq grid cells. Each grid cell requires two integers and a pointer, plus one grid record for each edge passing through that cell. A grid record requires 10 double-precision reals. There is a minimum of n grid records in total if each edge is contained in exactly one grid cell; there is a maximum of npq grid records if each edge is contained in every cell. The average practical case tends to be much closer to the minimum. Haines' implementation works out to $3I + 10R + 3P + (p+q)R + pq(2I+P) + 10Rn$ bytes minimum and $3I + 8R + 3P + (p + q + 2)R + pq(2I+P) + 10Rnpq$ bytes maximum. This reduces to a range of $80n + 3618$ to $32000n + 3618$ bytes for the 20×20 grid method, or $80n + 11698$ to $80000n + 11698$ bytes for the 100×100 grid method.

Haines' implementation of the p-bin trapezoid method uses a constant one integer, six doubles, and one pointer, plus p trapezoids. Each trapezoid uses two doubles, one integer, and one pointer, plus one edge record for each edge falling within the bin. An edge record uses two integers and two doubles. There is a minimum of n edge records if each edge falls exactly into one bin, and a maximum of np if each edge falls within every bin (when the testing region is limited to the polygon bounding box, there is a minimum of 2 edge records per bin, and so the minimum is bounded below by $\max\{n, 2p\}$). The average distribution of edges would not be as skewed towards the minimum as for the grid method. This implementation works out to $6R + I + P + p(2R + I + P) + n(2I + 2R)$ bytes minimum and $6R + I + P + p(2R + I + P) + np(2I + 2R)$ bytes maximum. This reduces to a range of $20n + 494$ to $400n + 494$ bytes for the 20-bin trapezoid method, or $20n + 2254$ to $2000n + 2254$ bytes for the 100-bin trapezoid method.

The grid method tends to be more memory-intensive than the CSG method (2.5 to 2500 times as expensive), but significantly faster (15 to 40 times faster). The trapezoid method is also more memory-intensive (0.7 to 70 times as expensive) in general, but only somewhat faster (1.5 to 3 times faster). Application-dependent point-in-polygon algorithm selection is necessary for maximal performance.

3 Building the CSG Tree and Edge-Sequence Graph

In this section we briefly describe how we build the CSG tree and the Edge-Sequence Graph. The most interesting aspect is the opportunities to optimize the pruning by reordering children of nodes in the tree. Since we are effectively pruning the CSG tree for some test points, we wish to permute the children of nodes within the tree to minimize the number of edges that need to be tested against, on average. In general, finding the permutation that yields optimal average performance requires us to analyze all such possible permutations. We resort to heuristic methods to approximate the optimal permutation while performing less work. Such heuristics will be discussed in Section 3.3.

3.1 Building the CSG Tree

Dobkin et al. [6] give a constructive proof that a CSG representation exists for every non-self-intersecting polygon, consisting of a Boolean formula in which each edge appears once and without complementation. We give a conceptual overview of the algorithm and refer the interested reader to the original for the details.

Each edge of a polygon can be described as being a segment of a line dividing the plane in half. By orienting the edges counterclockwise, we define the half-plane to the left of an edge to be the one of interest, and say that it supports the polygon. The interior of the polygon can then be described by a combination of Boolean AND and OR operators acting on the half-planes supporting the polygon. Whenever two edges meet at an acute angle, the interior of the polygon is described by an AND of the corresponding half-planes; an obtuse angle requires an OR of these half-planes. As a result, we can build up the sequence of operations that describe the interior, but the correct order in which these operations must be applied (the parenthesization of the formula) is still to be determined.

Parenthesization begins by dividing the polygon into two bi-infinite chains by splitting it at its left- and rightmost vertices and extending the edges incident upon these vertices to infinity. The convex hull of a bi-infinite chain is determined and the extremal vertex in the direction opposite the vector sum of the two semi-infinite edges of the chain is found. The chain is then split at this vertex, and the incident edges extended to infinity, forming two smaller chains; the process proceeds recursively, ending on a chain when it contains only one edge. Whenever a split occurs, the half-planes corresponding to edges in a chain are grouped within parentheses in the Boolean formula for the polygon; this results in a properly-nested formula consisting of unambiguously ordered binary operations. The difficulty is in performing this construction efficiently; Dobkin et al.'s algorithm is a practical $O(n \log n)$ for an n-edge polygon.

3.2 Building the ESG

An ESG for a simple polygon contains an array of edges (*edges*), an array of indices into the edge array (*lookup*), an integer indicating the number of edges in the polygon (*edgec*), and a Boolean flag indicating whether the polygon is oriented counterclockwise (*ccw*). The latter is not strictly required, but is of practical value when comparing the algorithm against other algorithms that ignore orientation of edges in defining the true interior of a polygon. The *lookup* array contains two elements for each edge occurring in the polygon.

An ESG is constructed from an *embedded* CSG tree, in which the children of each node have been assigned an order (perhaps arbitrarily). After enumeration, ESG-Construct() proceeds to walk through the leaves of the CSG tree. At each leaf, it determines which edge would next need to be tested given that the current edge tested either TRUE or FALSE against a particular point. The indices of these edges are then stored in the next two positions in the lookup array; at the same time, the current edge is also copied to the edge array. Finally, the Boolean flag indicating whether the polygon is oriented counterclockwise is set—the root node of the CSG tree is an AND node if and only if the polygon is oriented counterclockwise. Detailed code is given in the full paper.

3.3 Embedding the CSG Tree

Figure 2 shows a particular order for our n-ary tree CSG representation of a polygon. This ordering is not unique. Since the order of edges in the ESG data structure can greatly affect average performance of point-in-polygon tests, it is important that the CSG tree (the structure the ESG is derived from) be properly embedded in the plane by permuting sibling nodes.

In the full paper. we give a polynomial-time algorithm for computing, for a fixed tree, the expected time per point location under the assumption that all queries in a given bounding box are equally likely. To find the best tree, we look at all permutations of siblings. Here we compare the default permutation with a simple heuristic of sorting sibling subtrees by increasing number of edges in them, breaking ties by testing longer edges first. More sophisticated heuristics are possible.

Table 1 illustrates that this heuristic (noted as the sorted variety) does improve average performance over the default permutation 0–20%, becoming greater for larger polygons. It is important to note that this is average performance, however; this heuristic does slow down performance for some instances of polygons.

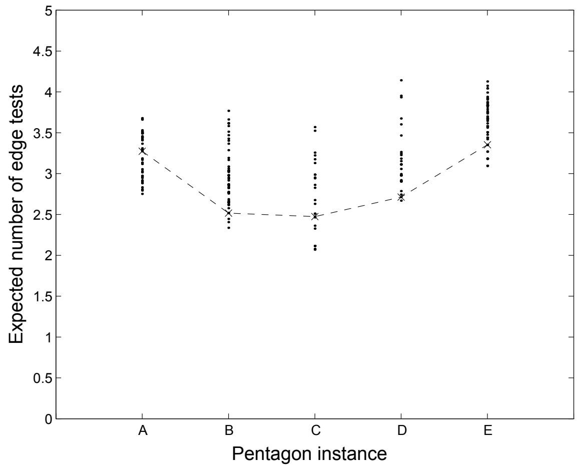


Fig. 4. *Analytic performance of the simple heuristic, pentagon instance vs. expected number of edge tests; the cost of the permutations selected by the simple heuristic are connected with a dotted line.*

Figure 4 illustrates the analytic performance of this simple heuristic for five random pentagons A–E.[1] The range of possible costs for performing point-in-polygon tests against each of five random pentagons was calculated for all the possible permutations of each; points to be tested were assumed to be distributed uniformly across the bounding box of each. Note that the simply-sorted heuristic generally selects a permutation with better than median or mean cost.

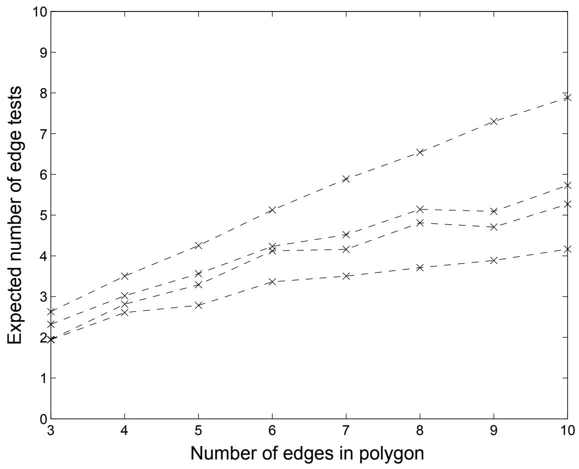


Fig. 5. *Analytic performance comparison, polygon size vs. cost metric. The four lines represent (from top to bottom) average maximum cost permutation, average default permutation, average simply-sorted heuristic permutation, and average minimum cost permutation. Data computed on 50 sample polygons used in Table 1 for each of three through ten edges per polygon.*

This computation was repeated for the random polygons, with 10 or fewer edges, used in the point-in-polygon tests listed in Table 1. Figure 5 shows, from top to bottom the average maximum cost, average cost of the default permutation, average cost of the simply-sorted permutation, and average minimum cost, across the 50 sample polygons. The predicted cost of the simply-sorted permutation is consistently lower than that of the default; however, there is significant room for improvement, as the gap between the heuristic permutation and the true minimum is trending towards growth.

[1] These pentagons were, in order: (*a*) (110,-1000), (655,-323), (484,-263), (-324,156), (-655,1000), (*b*) (-1000,467), (248,-303), (728,-467), (329,-315), (1000,-36), (*c*) (-976,1000), (-963,695), (-550,456), (363,-503), (976,-1000), (*d*) (197,-1000), (669,313), (149,321), (426,593), (-669,1000), and (*e*) (-1000,-837), (1000,373), (-85,682), (-4,307), (-789,837). These vertices were generated pseudo-randomly and scaled so that their bounding rectangles were centered at the origin; the true bounding box was used for calculation of the cost metric.

4 Extensions

Ray/polygon intersection is the basis of raytracing algorithms in computer graphics rendering. The algorithms described in this paper generalize to detect ray/ polygon intersections in 3-space in a straightforward way, once one is familiar with *Plücker coordinates* [14]. They also use the minimum arithmetic precision possible for the problem, which allows them to be implemented exactly. For small polygons, the number of arithmetic operations is less than that for Badouel's algorithm [2] for example, but this is an unfair comparison. Badouel uses a clever reduction to 2D point location, but then employs one of the poorer 2D point location schemes. Using his reduction with a better 2D scheme gives fewer arithmetic operations than detection directly in 3-space. This reduction, however, significantly increases the arithmetic complexity.

To generalize our algorithms, we use Plücker coordinates to represent edges and lines in 3-space. Using homogeneous coordinates, an oriented edge will start at point $\mathbf{p}$ and end at point $\mathbf{q}$. Such an edge can be represented as a Plücker point through a 6-tuple of 2×2 determinants:

$$\widehat{\mathbf{pq}} = \left(\begin{vmatrix} p_w & p_x \\ q_w & q_x \end{vmatrix}, \begin{vmatrix} p_w & p_y \\ q_w & q_y \end{vmatrix}, \begin{vmatrix} p_w & p_z \\ q_w & q_z \end{vmatrix}, \begin{vmatrix} p_x & p_y \\ q_x & q_y \end{vmatrix}, \begin{vmatrix} p_x & p_z \\ q_x & q_z \end{vmatrix}, \begin{vmatrix} p_y & p_z \\ q_y & q_z \end{vmatrix} \right).$$

Similarly, an oriented line passing from the point $\mathbf{r}$ to the point $\mathbf{s}$ can be represented as a Plücker hyperplane through a different 6-tuple of determinants:

$$\widetilde{\mathbf{rs}} = \left(\begin{vmatrix} r_y & r_z \\ s_y & s_z \end{vmatrix}, -\begin{vmatrix} r_x & r_z \\ s_x & s_z \end{vmatrix}, \begin{vmatrix} r_x & r_y \\ s_x & s_y \end{vmatrix}, \begin{vmatrix} r_w & r_x \\ s_w & s_x \end{vmatrix}, -\begin{vmatrix} r_w & r_y \\ s_w & s_y \end{vmatrix}, \begin{vmatrix} r_w & r_z \\ s_w & s_z \end{vmatrix} \right).$$

The dot product of these two vectors happens to be identical to the determinant:

$$\delta = \begin{vmatrix} p_w & p_x & p_y & p_z \\ q_w & q_x & q_y & q_z \\ r_w & r_x & r_y & r_z \\ s_w & s_x & s_y & s_z \end{vmatrix}; \tag{1}$$

this is reasonable, since the expansion of δ into 2×2 minors gives

$$\delta = \sum_{i=1}^{6} \widehat{pq}_i \, \widetilde{rs}_i. \tag{2}$$

This determinant will be zero if all four points are coplanar; the right-hand rule explains the sign of a non-zero δ. Point the thumb of your right-hand in the direction of the edge $\mathbf{pq}$ and curl your fingers around this line. If the line $\mathbf{rs}$ passes $\mathbf{pq}$ in this direction, δ will be positive; if it passes in the opposite direction, δ will be negative. Since we only care about the sign of δ, we can scale it or the two vectors arbitrarily; this means that we can normalize each 6-tuple by dividing it by one of its elements, effectively reducing it to a 5-tuple.

We can use this for performing raytracing by altering the PointInPolygon() procedure slightly. Edge-sequence graphs will now contain the Plücker point coordinates, and the oriented line (i.e., the ray) to be tested is passed to the new procedure instead of a single point. We simply replace the Edge-PointLeftOf() procedure with a test of the sign of the determinant d between the edge being tested and our ray:

$$result := (\ \mathsf{DotProduct}(\ \mathsf{edge},\ \mathsf{line}\ ) \leq 0\ );$$

notice that the operator sign has been flipped.

There is one additional concern: we need to make sure the polygon is oriented correctly. Depending on whether we are looking at its backface or its front-face, the interior and exterior of the polygon will be switched. To determine the side of the polygon on which a point lies, we can use three vertices $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ of the polygon to expand Equation 1 for a variable point $\mathbf{s}$ to obtain the equation of the plane π that contains the polygon. We store this with the polygon. If $\pi(\mathbf{s})$ is greater than zero, the point $\mathbf{s}$ is on the side of the polygon where the edges are oriented counterclockwise, i.e., the front-face; if it is less than zero, $\mathbf{s}$ is on the other side. For a ray with origin $\mathbf{u}$ passing through point $\mathbf{v}$, if $|\pi(\mathbf{u})| \leq |\pi(\mathbf{v})|$, the ray does not intersect the plane of the polygon. Otherwise, if $\pi(\mathbf{u}) > \pi(\mathbf{v})$, the ray is pointing towards the front-face of the polygon; if $\pi(\mathbf{u}) < \pi(\mathbf{v})$, the ray is pointing towards the backface.

The disadvantages of this alteration to the CSG approach are in time and storage. In 3-space, each edge must store its Plücker coordinates in either 5 rationals or 6 integers. PointInPolygon() performs, for each edge tested, 4 rational multiplications if the Plücker coordinates have been normalized (using different coordinates to normalize Plücker points and Plücker hyperplanes) or 6 integer multiplications.

The main advantage is the low arithmetic precision required. Suppose that the input points have w-coordinates of unity, that other coordinates are b-bit integers, and that for endpoints of an edge or for the ray points $\mathbf{r}$ and $\mathbf{s}$ the maximum difference in any coordinate is an integer of $d \leq b$ bits. Then the coordinates of the Plücker points and hyperplanes are integers of d or $b+d+1$ bits, and $b+2d+2$ bits are sufficient to contain the result of any Plücker dot product. Thus, we could, in 53-bit native arithmetic, exactly evaluate Plücker tests on coordinates with $b = 24$ bits when the difference between endpoints can be expressed in $d = 12$ bits.

Badouel [2] gives a two-step process to reduce ray/polygon intersection into a point-in-polygon test of a projection on one of the coordinate planes. Let P be the polygon in 3-space and let π denote the plane containing P. In the first step, after checking that the ray $\mathbf{rs}$ does intersect π, this intersection point $\rho = \mathbf{rs} \cap \pi$ is calculated. In the second step, the intersection is projected onto one of the coordinate planes—which plane is determined by the two longest dimensions of the axis-aligned bounding box for P in 3-space [12]. The ray intersects P if the projection of ρ is inside the projection of P.

Consider the arithmetic complexity in more detail. The two-step process parameterizes the ray $\mathbf{rs}$ and solves for the parameter t that determines when

$\rho = r + t(s - r)$ lies on the plane π. This takes 1 division of the dot products of r and s with the plane; since these dot products are also used by the Plücker algorithm, we will not count them, but will count the division. Now, t is a rational number with numerator and denominator each having $b + 2d$ bits. To obtain the rational coordinates of ρ in the projection plane takes 2 more multiplications. Then a 2D point location such as the unaltered CSG method can be used in the projection, with 2 multiplications per edge tested. With integers, the division is avoided, 4 multiplications are needed to find the projection of ρ in homogeneous form with $2b + 2d$ bits, then each edge test takes 3 multiplications and uses $2b + 3d$ bits, plus two or three bits for rounding. Thus, the two-step process performs fewer multiplications/divisions if more than 2 edges are tested, but native 53-bit arithmetic is limited to $b = 14$ bit coordinates with the difference between endpoints expressed in $d = 7$ bits.

The projection onto a coordinate plane can reduce the performance of the faster 2D point location methods, since the projection will tend to produce more long, skinny triangles. Although the grid method performs well on long, skinny triangles under normal circumstances, the projection will also alter the distribution of test points in a similar manner. As a result, the heaviest concentration of test points will occur in those grid cells in the vicinity of the polygon—where the grid method performs least well.

The final arbiter of performance will be empirical measurement of implementations. As with the 2D form of the CSG algorithm, the space-, time-, and precision-tradeoffs here need to be considered in an application-specific context.

5 Conclusions

We have presented an algorithm for performing point-in-polygon tests using a constructive solid geometry (CSG) representation of polygons. This algorithm uses a new data structure, the edge sequence graph (ESG), that is constructed from a CSG representation of a given polygon during a preprocessing phase. The speed of the CSG point-in-polygon algorithm derives from a tight inner loop and the pruning of unnecessary edge tests, made possible by ESGs. Algorithms for the construction of an ESG and the calculation of the expected cost of using a given ESG to test a random point were also presented. Because the ordering of edges affects the cost of using an ESG, heuristics were presented for selecting a good ordering—deterministic selection of the fastest permutation is NP-hard. Three-dimensional raytracing and other extensions were briefly discussed.

The CSG point-in-polygon algorithm provides a tradeoff of slower execution for reduced storage space requirements over faster, existing methods. We have demonstrated empirically that this new point-in-polygon algorithm using a simple heuristic is faster on most non-convex polygons than all the one-shot methods presented by Haines. Only the half-plane method is faster for triangles (by 13%) and only the 100×100 grid method is equally fast for quadrilaterals. Only the grid and trapezoid methods outperform this algorithm as polygons grow large.

We have shown that the average storage space required for this algorithm is significantly less than for the grid and trapezoid methods.

Most of the experimental and analytical work in this paper on determining cost of the algorithms has concentrated on point-in-polygon tests in which the points to be tested are uniformly distributed. In particular application domains, such as geographical information systems, test points could be heavily concentrated in particular regions, e.g., queries about average annual rainfall at a city on a map. Specialized heuristics are required for non-uniform distributions, and the performance of all the methods tested here are likely to change in this situation.

The simply-sorted heuristic for embedding the CSG tree does a better than average job of speeding up the performance of the point-in-polygon algorithm; however, there is significant room for improvement, especially for polygons with many sides, as evidenced by Figure 5.

We have begun an investigation of other possible heuristics by analyzing the cost of all the permutations of randomly-generated 4- and 5-sided polygons, to see why the fast ones are fast and the slow ones are slow. Selection of the initial edge to test against is often a major determinating factor in cost. We generally want to test against an edge whose corresponding line creates a region large in probability that is completely outside (or completely inside) the polygon. Such an edge is often not available, and it remains unclear if there exists one or two simple determining factors of cost in such an instance. Randomization of child nodes is another possible heuristic for increasing average performance. Given that the simply-sorted heuristic gives better than median or mean performance, on average, for polygons with a few sides, randomization will not improve performance here on average. For polygons with more sides this might not be the case.

It has come to our attention that the binary decision diagram (BDD) data structure from the field of formal verification is closely related to CSG trees and that ordered BDDs (OBDD) [4] are closely related to ESGs. A survey of OBDDs can be found in [5]. Reordering of variables in OBDDs is analogous to reordering edge tests in ESGs; it is recognized as a difficult problem and finding an optimal solution for variable ordering in an OBDD has been proven to be NP-hard [3]. With OBDDs, the chief concern is to reduce their size; with ESGs, it is to reduce their probability-weighted average path length. It is unclear whether these two concerns are compatible.

References

1. Franklin Antonio. Faster line segment intersection. In David Kirk, editor, *Graphics Gems III*, chapter 1.4, pages 199–202. Academic Press, Boston, MA, USA, 1992.
2. Didier Badouel. An efficient ray-polygon intersection. In Andrew S. Glassner, editor, *Graphics Gems*, pages 390–393. Academic Press, Boston, MA, 1990.
3. B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

4. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, August 1986.
5. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
6. David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 31–40, August 1988.
7. Chris Green. Simple, fast triangle intersection. *Ray Tracing News*, 6(1), 1993. `ftp://ftp.princeton.edu/pub/Graphics/RTNews`.
8. R. Hacker. Certification of algorithm 112: position of point relative to polygon. *Communications of the ACM*, 5:606, 1962.
9. Eric Haines. Point in polygon strategies. In Paul S. Heckbert, editor, *Graphics Gems IV*, chapter 1.4, pages 24–46. Academic Press, Boston, MA, USA, 1994.
10. Franco P. Preparata and Micheal Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, Berlin, Germany, 1985.
11. M. Shimrat. Algorithm 112: position of point relative to polygon. *Communications of the ACM*, 5:434, 1962.
12. John M. Snyder and Alan H. Barr. Ray tracing complex models containing surface tessellations. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):119–128, July 1987.
13. John Spackman. Simple, fast triangle intersection, part II. *Ray Tracing News*, 6(2), 1993. `ftp://ftp.princeton.edu/pub/Graphics/RTNews`.
14. Jorge Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations.* Academic Press, 1991.

Accessing the Internal Organization of Data Structures in the JDSL Library*

Michael T. Goodrich[1], Mark Handy[2], Benoît Hudson[2], and Roberto Tamassia[2]

[1] Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218
`goodrich@cs.jhu.edu`

[2] Department of Computer Science
Brown University
Providence, Rhode Island 02912
`{mdh, bh, rt}@cs.brown.edu`

Abstract. Many applications require data structures that allow efficient access to their internal organization and to their elements. This feature has been implemented in some libraries with *iterators* or *items*. We present an alternative implementation, used in the Library of Data Structures for Java (JDSL). We refine the notion of an item and split it into two related concepts: *position* and *locator*. Positions are an abstraction of a pointer to a node or an index into an array; they provide direct access to the in-memory structure of the container. Locators add a level of indirection and allow the user to find a specific element even if the position holding the element changes.

1 Introduction

In using a data structure, the user must be granted some form of access to its elements, preferably an efficient form. In some cases, the access can be very limited: in a stack, for example, we can look only at the top element. But in many cases, we need a more general mechanism to allow the user some handle on the elements. In an array structure, we can use indices. In a linked structure, we can use pointers to the nodes.

From the perspective of object-oriented design, however, we need to restrict access to the internals of the data structures. Otherwise, the user can make a container invalid by modifying data required to maintain the consistency of the data structure. For instance, if a list gives the user pointers to its nodes, the user should not be able to modify explicitly the successor and predecessor pointers of the node.

* Work supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grants CCR–9625289, CCR–9732327, and CDA-9703080.

1.1 Previous Work

Two well-known abstractions for safely granting access to the internals of data structures are *iterators* and *items*. STL [13], JGL [20] and the Java Collections Framework of Java 1.2 [18] use iterators. LEDA [11, 21] uses items.

Iterators provide a means to walk over a collection of elements in some linear order. At any time, an iterator is essentially a pointer to one of the elements; this pointer can be modified by using the incrementing functions the iterator provides. In C/C++, a pointer (other than to void) fits this profile: it points to a position (an address in memory), and the increment operation '++' makes the pointer point to the next element in an array. In a linked list, a simple class with a pointer to a node of the list will do. The access function will return the element of the node, and the increment function will move to the next node in the list. This discussion is of an STL *input iterator*. Other iterators in the STL hierarchy can also move backwards, write to the current element, and so on. The capabilities of iterators do fulfill the requirement to hide the data structure organization from the user, while still allowing the user to refer to an element efficiently. They also make it easy to act on every element of a list.

Iterators, however, have their limitations. For example, if a program gets an iterator over a container and subsequently modifies the container, the question arises as to how the iterator will behave. The modification may be reflected by the iterator, or it may not, or the iterator may simply be invalidated. Furthermore, iterators work well on linearly arranged data, but it is not clear how to extend them to non-linear data structures, such as trees and graphs.

Items, as they are referred to in LEDA, are the nodes of a data structure. In order to keep the user from having access to the internal structure, all fields of an item are private; all access is done through the container, passing in the item. This works quite well to describe arbitrary linked structures, including linear data structures. This approach does not preclude the use of iterators: one can easily be written by having it store a current item, which it changes on incrementing. For array-based structures, LEDA does not provide items. For uniformity's sake, it would be possible to provide them, either by storing a pointer to an item-object in each slot of the array or by creating an item class that simply hides an array index.

1.2 Overview

We have refined the notion of an item and split it into two abstractions: *position* and *locator*. Positions abstract the notion of a place in memory, and provide direct but safe access to the internals of a data structure. Locators are a handle to an element; while the position of the locator within a container may change, the element will not.

In the following sections, we discuss these concepts at greater length, and introduce two container types which use positions and locators in their interfaces. We then present our implementation of these abstractions in the framework of JDSL, a Java library that provides a comprehensive set of data structures.

Examples show how one would use positions and locators, notably in the context of Dijkstra's shortest-path algorithm. We also note some of the constant-factor costs of the design and how they can be reduced. Finally, we discuss the current state of JDSL in the contexts of teaching and of research.

2 Positions and Locators

We have developed a pair of notions related to the notion of items. On the one hand, we can talk about a *position* within a data structure; on the other hand, we can talk about a *locator* for an element within a data structure. A position is a topological construct: one position might be before another in a sequence, or the left child of another in a binary tree, for example. The user is the one who decides the relationships between positions; a container cannot change its topology unless the user requests a change specifically. An element in a container can always be found at some position, but as the element moves about in its container, or even from container to container, its position changes.

In order to have a handle to an element regardless of the position at which the element is found, we introduce the concept of a locator. Whereas the position-to-element binding can change without the user's knowledge, the locator-to-element binding cannot. Thus, positions closely resemble items, and the distinction between position and locator is new.

These two concepts can be distinguished more clearly by understanding, at a high level, their implementations. A position refers to a "place" in a collection. More precisely, a position is a piece of memory that holds

- the user's element
- adjacency information (for example, next and prev fields in a sequence, right-child and left-child fields in a binary tree, adjacency list in a graph)
- consistency information (for example, what container this position is in)

Thus, a position maps very closely to a single node or cell in the underlying representation of the container. Methods of some containers are written primarily in terms of positions, and a method that takes a position p can immediately map p to the corresponding part of the underlying representation. Section 3.1 gives more detail about the implementation of positions in JDSL.

A locator is a more abstract handle to an element, and its implementation does not map so directly to the underlying memory. Three principles constrain the implementation:

- For correctness, the container must guarantee that the locator-element binding will remain valid, even if the container moves the element to a different position.
- All containers, even containers with very abstract interfaces, must be realized ultimately in memory, and memory is positional.
- Methods of some containers are written in terms of locators, and a method that takes a locator ℓ needs to be able to map ℓ quickly to the underlying (positional) representation of the container.

Therefore, in implementation, a locator must hold some positional information (often a pointer to a position). The container uses this information to map the locator to the container's representation, and the container takes care to update the information when the locator changes position. Section 3.2 gives more detail about the implementation of locators in JDSL.

2.1 Positional Containers and Key-Based Containers

We distinguish between two kinds of containers, positional and key-based. In JDSL, the interfaces of positional containers are written primarily in terms of positions, and the interfaces of key-based containers are written primarily in terms of locators.

A *positional container* is a graph or some restriction of a graph. Examples are sequences, trees, and planar embeddings. A positional container stores the user's elements at vertices of the graph, and in some cases also at edges of the graph. Its interface maps very closely to its implementation, which usually involves linked data structures or arrays. The positions mentioned in its interface map to individual nodes or cells of the implementation.

A *key-based container* is an abstract data type that stores key-element pairs and performs some service on the pairs based on the keys. Its in-memory representation is entirely hidden by its interface; nothing is revealed about where it stores its elements. Since the hidden representation must be positional, a key-based container maps the locators mentioned in its interface to the representation using positional information stored in the locators.

2.2 Examples

We illustrate these concepts using the distinction between a binary tree and a red-black tree. Drawings of these two data structures might look the same, but the semantic differences are important: An unrestricted binary tree allows the user to modify the connections between nodes, and to move elements from node to node arbitrarily. Hence, a binary tree is positional, and its interface is written in terms of positions. A red-black tree, on the other hand, manages the structure of the tree and the placement of the elements on behalf of the user, based on the key associated with each element. It prevents the user from arbitrarily adjusting the tree; instead, it presents to the user a restricted interface appropriate to a dictionary. Hence, a red-black tree is key-based, and its interface is written in terms of locators. A locator guarantees access to a specific element no matter how the red-black tree modifies either the structure of the tree or the position at which the element is stored.

In both cases, the container provides access to its internal organization. The binary tree provides direct (but limited) access to its nodes (as positions). The red-black tree provides locators that specify only the order of the keys in the dictionary, without contemplating the existence of nodes.

A binary tree would support operations like these:

```
leftChild (Position internalNode) returns Position;
cut (Position subtreeRoot) returns BinaryTree; // removes and returns a subtree
swap (Position a, Position b); // exchanges elements and locators
```

A red-black tree would support operations like these:

```
first() returns Locator; // leftmost (key,element) pair
insert (Object key, Object element) returns Locator; // makes new locator
find (Object key) returns Locator;
replaceKey (Locator loc, Object newKey) returns Object; // old key
```

3 JDSL

JDSL, the Data Structures Library for Java, is a new library being developed at Brown and at Johns Hopkins, which seeks to provide a more complete set of data structures and algorithms than has previously been available in Java. Other goals are efficiency, run-time safety, and support for rapid prototyping of complex algorithms. The design of containers in JDSL is closer to that of LEDA and CGAL [5] than to that of STL and JGL. In addition to the standard set of vectors, linked lists, priority queues, and dictionaries, we provide trees, graphs, and others. We also have algorithms to run on many of the data structures, and we are developing a set of classes for geometric computing. The implementations of data structures are hidden behind abstract interfaces.

The interface hierarchy in figure 1 has two major divisions. One, between inspectable and modifiable containers, allows us to restrict subinterfaces to include only methods which are valid. The other, between PositionalContainer and KeyBasedContainer, implements the distinction we drew in section 2.1 between these two types of containers. Finally, two interfaces not pictured here implement the distinction between Position and Locator. PositionalContainer interfaces are written mainly in terms of Position, while KeyBasedContainer interfaces are written mainly in terms of Locator.

3.1 Position

The Position interface is present in most methods of positional containers, and implementations of the Position interface are the basic building blocks of those containers. Positions are most commonly used to represent nodes in a linked structure (such as a linked list), or indices in a matrix structure (such as a vector). In data structures which have different types of positions—such as vertices and edges in graphs—we use empty subinterfaces of Position useful only for type-checking purposes. While positions are closely tied to the internal workings of their container, their public interface is limited, so that they are safe to return to user code. Most operations must actually be done by calling upon the position's container, passing in the position as an argument—for example, Position

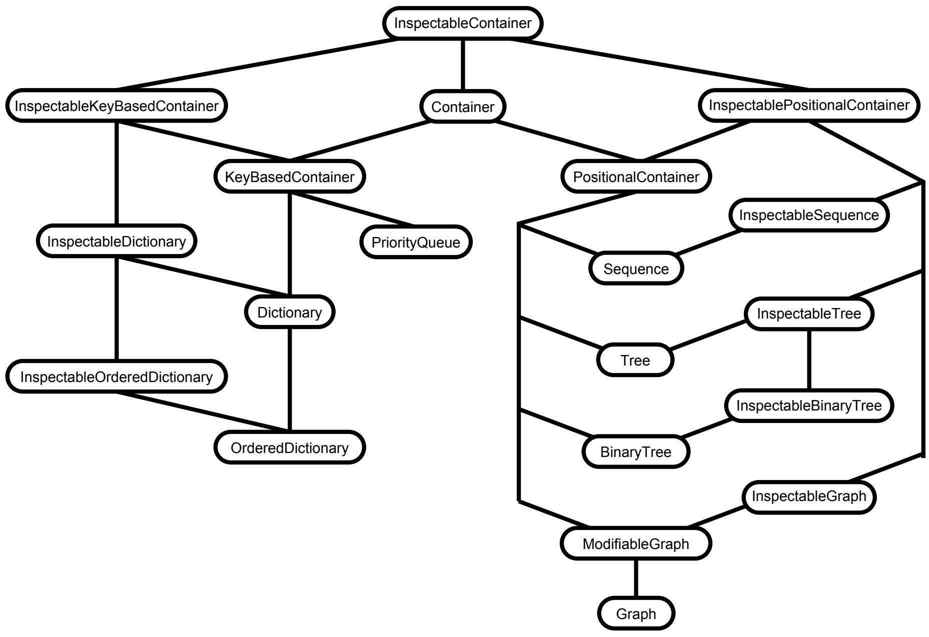


Fig. 1. A partial hierarchy of the JDSL interfaces.

Sequence.after(Position), which returns the position following the argument in the list. Only operations applicable to positions from any container are included in the interface.

```
public interface Position extends Decorable {
  public Object element() throws InvalidPositionException;
  public Locator locator() throws InvalidPositionException;
  public Container container() throws InvalidPositionException;
  public boolean isValid();
}
```

Note that locators are present even in positional containers, so the Position interface provides a way to find the locator stored at a given position.

In most implementations, the position classes are inner classes of the container that will use them, and have private methods allowing access by the container to their internals.

Since positions are so strongly tied to the internal structure of the data structures which contain them, they have no meaning once removed from a container. Hence, a deleted position is marked as invalid and any subsequent use of it will raise an exception. In addition, containers throw an exception if they are passed a position of the wrong class, a null position, or a position from a different container.

The Position interface is typically implemented either by a node in a linked data structure, or by a special object in an array-based data structure (see figure 2). We use a special object for two reasons: to store consistency checking information, and to adhere to the Java requirement that only objects can implement an interface. Without that requirement, we could use simply an integer index as the position.

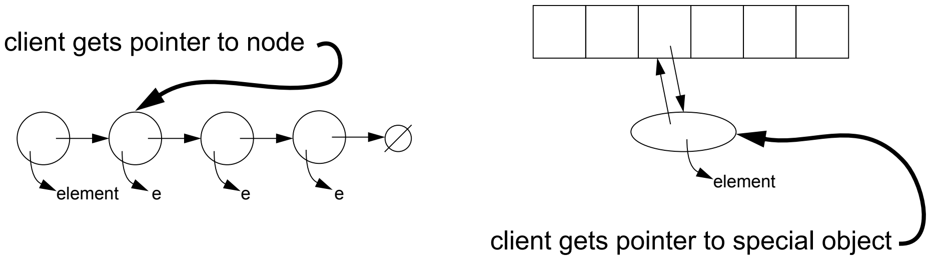

client gets pointer to node
element
e
e
e
element
client gets pointer to special object

Fig. 2. Two sample implementations of positions in a sequence.

As an example of using Positions in a simple application, we can look at code which reverses a Sequence (such as a linked list or a vector):

```
void reverse(Sequence S) {
    if(S.isEmpty()) return;
    Position front = S.first(), back = S.last();
    for(int i=0; i<S.size()/2; i++) {
        S.swap(front, back);
        front = S.after(front);
        back = S.before(back);
    }
}
```

Positions are thus fairly similar to LEDA's items. The latter have only private members, and are friends (in the C++ sense) of their container, just as we hide most Position members except to the container.

Unlike an iterator, a position is always tied to a particular node, and cannot be used directly to traverse a data structure. Instead, an iterator would use a pointer to a position as its method of indexing into the data structure.

3.2 Locator

From the point of view of the user, Locators are essentially pointers to elements. They allow the user to efficiently locate an element within a data structure, typically in constant time, and to make a request of a data structure to act on an element. A locator provides a contract to its user that it will always be associated with a specific element even if the position of that element changes.

```
interface Locator extends Decorable {
  public Object element();
  public Container container();
  public boolean isContained();
  public Object invalidate();
  public boolean isValid();
}
```

Locators are typically used to access *(key, element)* pairs within KeyBased-Containers. Upon inserting an element, a locator is created and returned to the user. The locator can also be retrieved, in a dictionary, by calling find(Object). This locator can then be used to refer to the pair in constant time: the user need not search for it for each operation. For instance, the call replaceElement(Locator, Object) will typically take constant time, whereas a search would likely have taken logarithmic time. Similarly, replaceKey(Locator, Object) in a priority queue realized as a binary heap will take logarithmic time, while searching would take linear time.

In order to guarantee these time bounds, the container must be able to immediately map the locator to the underlying, positional data structure. Because of this constraint, we have found it useful to implement only one class, UniversalLocator, which stores its element and its position within its container. The user will only see the objects through the Locator interface above. However, the UniversalLocator class provides some additional methods which allow containers to modify the locator.

This allows us to implement the concept of *universality* of locators across a set of containers. A locator created by one container can be removed from it, and moved to another container within the set. The locator remains a valid handle to the element, even when it is not contained in any data structure. In JDSL, all containers accept UniversalLocator unless they specifically state otherwise.

Universality allows a user to move locators from one data structure to another, as long as all the data structures support UniversalLocator. This is useful, for example, in sorting applications, where the position-to-element binding is broken, but the locator-to-element binding need not be. By inserting and removing locators rather than elements, the author of the sort can allow the user to keep useful handles to the elements through the locators. For example, consider the code for a PQ-Sort in figure 3.

Universal locators require some overhead: a Position object needs to be created. In most cases, this overhead is acceptable: preliminary experiments indicate that our red-black tree, written under the paradigm of universal locators, is at least as fast as the red-black tree in JGL or JDK 1.2 [3]. But the interfaces do not require that universality be implemented by all locators. This allows potentially smaller or faster data structures in applications where the generality is not needed. For instance, a red-black tree locator could store its color, children, and parent, rather than requiring a separate position to store that information (see figure 4).

```
public void sort(Sequence S, Comparator c) {
    PriorityQueue Q = new VectorHeap(c); // use your favorite PriorityQueue

    // remove all elements from S, but retain the same locators
    while(!S.isEmpty()) {
        Locator loc = S.first().locator() ; // loc is in S
        S.removeFirst(); // after this, loc is in no container
        Q.insert(loc); // now loc is in Q
    }

    // remove all elements from Q in ascending order
    while(!Q.isEmpty()) {
        S.insertLast(Q.removeMin()); // move the locator from Q to S
    }
}
```

Fig. 3. A PQ-sort implementation. Note how the locators can be preserved as they move from the sequence into the priority queue and back.

4 Labeling and Iterating in JDSL

4.1 Decorable

Many algorithms need to store extra state associated with the positions or elements of the data structures they use. For example, in a breadth-first or depth-first search over a graph, we need to mark nodes as visited. Essentially, we want to add decorations or attributes to the positions of our data structures.

We can imagine the system as a two-dimensional matrix of values, with positions indexing the rows and attribute names indexing the columns [15]. We need to be able to find the value of a specific attribute for a specific position. The three solutions commonly used now are:

1. to copy the data structure into one which has the extra instance variables,
2. to use an external dictionary indexed by the positions (the *column-based* option), and
3. to associate an internal dictionaries with each positions (the *row-based* option); the dictionaries are indexed by attribute names.

The first solution clearly involves a large amount of copying of data structures. In a situation where a large number of relatively quick algorithms are being used, this could affect speed significantly.

The second solution is implemented in LEDA's node_map, edge_map, node_array, and edge_array. In JDSL, the column-based solution can be implemented simply by instantiating a hash table or other dictionary, using the hashcode() function of the positions as the key into the dictionary. Thus, no library support is required for the column-based solution.

JDSL supports the third solution; library support is required if the row-based solution is to exist at all. Position extends the Decorable interface, which requires that all positions have a dictionary associated with them. This is also the approach taken in the ffGraph library [17], which calls the attributes "labels". Under the assumption that there are more positions than there are decorations, the row-based solution is asymptotically faster than the column-based in the worst case. In the average case, the efficiencies are the same.

4.2 Enumerations

While one use of iterators is to encapsulate a position, they are also useful in iterating over an entire container. To support this functionality, JDSL requires that its data structures provide Enumerations over interesting sets of elements, positions, and locators.

One difficulty we noted with an iterator is its behavior upon modification of the underlying container. To avoid the issue, we specify that Enumerations provide a *snapshot* of the data. That is, subsequent modifications to the data structure over which we are enumerating do not modify outstanding enumerations. In general, a user who asks for an enumeration will use every element of it, so we are not affecting asymptotic efficiency. Also, we have developed optimizations to further reduce the cost (see section 4).

5 Examples of JDSL Containers

To bring together all the concepts we have discussed so far, we present examples implementations of three data structures. The following schematic figures show our standard implementations of binary trees, red-black trees, and general graphs:

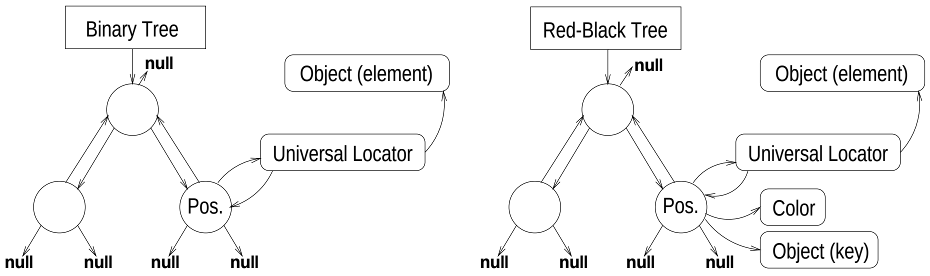


Fig. 4. Implementation of a BinaryTree, and implementation of a Dictionary as a red-black tree. Note the two have very similar diagrams, although the user sees very different interfaces.

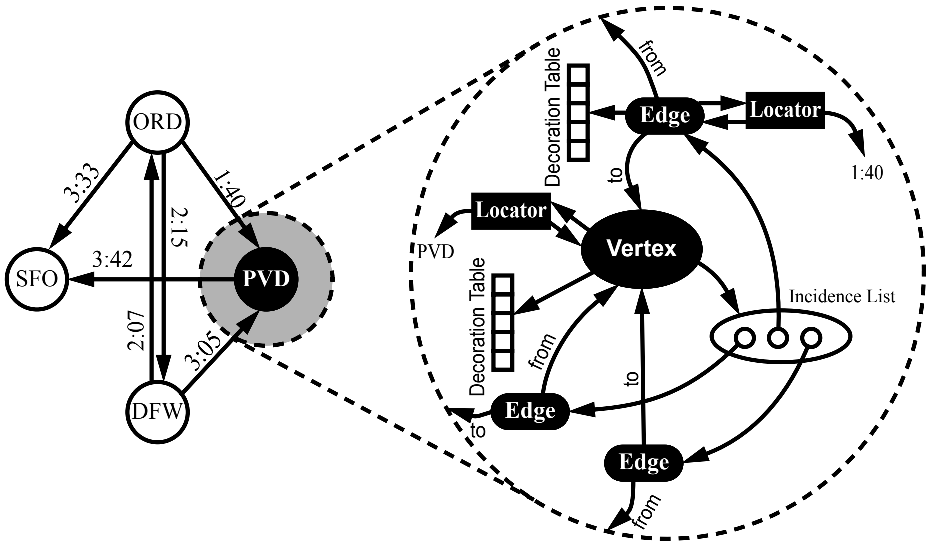


Fig. 5. Implementation of a Graph. Note Vertex and Edge are both empty subinterfaces of Position.

6 Example of an Algorithm

We present, as an example, a simple implementation of Dijkstra's algorithm for finding shortest paths; the implementation is based on [4]. The algorithm is typically used to illustrate the programming methodology of a library [15]. We assume the edges of the graph are weighted, using the Decorable mechanism, with non-negative Integer weights.

Here we use both positions and locators. The positions of a graph are its edges and vertices—this is reflected in JDSL by having Edge and Vertex extend Position. It is by using these that the algorithm traverses the graph. The locators we use allow us to find the vertices in the priority queue. We insert pairs *(distance, vertex)* into the queue, which represent the shortest yet known distance to the vertex. A vertex is chosen only if it is the closest vertex to the source that we have not yet encountered. When we choose a vertex, we discover all its edges. Through these edges, we may find a shorter path to another vertex. If we do, we need to update the key of the vertex within the queue (since the vertex should be removed from the queue earlier than previously believed), which we can do efficiently because we kept a locator to the vertex.

```
/**
 * Dijkstra's algorithm on an InspectableGraph, using integer weights.
 * The output is a list of edges which define the shortest path from s to t.
 */
public class Dijkstra {
    public Dijkstra(InspectableGraph g, Vertex s, Vertex t, Object weight) {
        graph_  = g;
        weight_ = weight;
        s_ = s; t_ = t;

        initialize();
        run();
        buildPath();
        cleanup();
    }

    private void run() {
        while( !pq_.isEmpty() ) {
            Vertex v = (Vertex) pq_.removeMin();
            if(v==t_) return; // if true, we've found the shortest path to t

            int distance = distance(v);

            // for all outgoing edges...
            for(Enumeration edges = graph_.outIncidentEdges(v);
                    edges.hasMoreElements() ; ) {
                Edge e = (Edge) edges.nextElement();
                Vertex dest = graph_.destination(e);

                int cumulative = weight(e) + distance;

                if( cumulative < distance(dest) ) { // relax the edge if applicable
                    Integer newdist = new Integer(cumulative);
                    pq_.replaceKey( locator(dest), newdist);
                    dest.set(incoming_, e);
                }
            }
        }
    }

    // Create appropriate decorations and put vertices into the heap
    private void initialize() {
        for(Enumeration verts = graph_.vertices(); verts.hasMoreElements(); ) {
            Vertex v = (Vertex)verts.nextElement();
            v.create(locator_ , pq_.insert(INFINITE, v));
            v.create(incoming_, null);
        }
        // we have set s to have infinite distance, but it has 0 distance
        pq_.replaceKey(locator(s_), ZERO);
    }
    // Build the list of edges from source to destination
    private void buildPath() {
        Vertex v = t_; // we're going backwards...
        while(v!=s_) {
            Edge e = incoming(v);
            output_.insertFirst(e); // so we insert at the head
            v = graph_.origin(e);
        }
    }

    // Clean up: destroy all decorations we created.
    private void cleanup() {
        for(Enumeration verts = graph_.vertices(); verts.hasMoreElements(); ) {
            Vertex v = (Vertex)verts.nextElement();
            v.destroy(locator_);
            v.destroy(incoming_);
        }
    }

    // Return the path as an enumeration of edges from s to t.
    public Enumeration getPath() { return output_.elements(); }

    // some accessors
    private Locator locator(Vertex v) { return (Locator)v.get(locator_); }
    private int distance(Vertex v) {
        return ((Integer)pq_.key(locator(v))).intValue();
    }
    private Edge incoming(Vertex v) { return (Edge)v.get(incoming_); }
    private int weight(Edge e) { return ((Integer)e.get(weight_)).intValue(); }

    // data structures we will be using
    private InspectableGraph graph_;
    private PriorityQueue pq_ = new VectorHeap(new IntegerComparator());
    private Sequence output_ = new NodeSequence();

    // the source and destination of the path
    private Vertex s_, t_;

    // keys for decorations
    private Object weight_ ;
    private Object locator_ = new Object();
    private Object incoming_ = new Object();

    // "infinity" and zero for the purposes of this program
    private static final Integer INFINITE = new Integer(Integer.MAX_VALUE);
    private static final Integer ZERO = new Integer(0);
}
```

7 Optimizations

JDSL's use of locators and positions is asymptotically efficient; algorithms do not suffer increased complexity, either in the worst case or in the average case, because access via locators and positions requires constant time. But a library which claims to be all-powerful invariably suffers from constant-factor efficiency losses: there are clear tradeoffs among power, elegance, and speed. We have developed some optimizations which should make JDSL competitive with other data structures libraries in terms of speed. Experiments are in progress to compare actual performance of JDSL with that of other libraries.

7.1 Lazy Allocation of Locators

One concern is caused by the requirement for each element to be associated with a locator. This obviously creates a large number of extra objects. Further, in PositionalContainers, the locators often go unused—they may be powerful, but many applications do not need the power. Therefore, most data structures in the library follow an *allocate-on-use* policy for creating locators. If the user asks to get a locator, via the Position.locator() method, for example, one is created. Otherwise, we short-circuit and store an element directly.

This can save considerable time and space, especially in situations where we are creating and destroying large numbers of positions, without ever querying them for a locator. The only cost here is a check and conditional branch whenever the element or locator is queried from a position, to see if the locator has been allocated or not.

7.2 Enumeration Caching

Another concern is the time used in building enumerations. Since these are snapshots of the data structures, creating them will take time at least linear in the number of elements to be included in the enumeration. Here we have applied a *copy-on-write* policy for some data structures: when an enumeration is asked for, we create an array which stores all the objects over which to enumerate, then return an enumeration over that array. If the same enumeration is asked for again, we can simply return another enumeration object, over the same array—in constant time. If a call is made to a function which modifies the correct data to return, we discard our array (Java's garbage-collection scheme takes care of disposing of it).

In this way, we are optimizing for having a number of read-only calls in between phases of modification. This is in fact realistic: a common use of a data structure is to fill it with data, then run some algorithms on it which only inspect the data.

The cost of this method is none in terms of time, but in most implementations, we have to keep an extra copy of the data around between the time an enumeration has been called for and the next modification. If we have multiple enumerations simultaneously, however, we save space since we share a single copy among them.

The method works especially well for array-based data structures. In these, the array itself can be used as the cached copy, so that even the first enumeration call can be done in constant-time. Only if the container is modified while there are enumerations outstanding do we actually need to copy any data.

8 Experience with JDSL

JDSL has a companion *teach* version [19] geared for pedagogical use. It has been used successfully in first-year, one-semester, CS2-level classes at Brown [16] and Johns Hopkins, and is the library discussed in the textbook by Goodrich and Tamassia [9]. At Brown, over a hundred CS2 students implemented the following data structures and algorithms using the JDSL *teach* library:

- Sequence, implemented with a circular array
- Binary search and quicksort of a Sequence
- Binary tree
- Binary heap, reusing the binary tree
- Red-black tree, reusing the binary tree
- Hash table
- Rabin-Karp string-searching algorithm
- Convex hull algorithm, using package wrap
- Spanning tree of an undirected graph
- Directed graph
- Prim's minimum-spanning-tree algorithm
- Dijkstra's shortest-path algorithm

The methodology of the library allows projects that are more theoretically sophisticated than were possible in past years, and more full-featured. Data structures met the interfaces specified by JDSL. For instance, most data structures included a full suite of modifiers (insertion, deletion, and replacement) and thorough error handling via exceptions. Students were supported by visualizers and testers written within JDSL [1].

A number of implementation projects have been written based on the research version of the library, as well. Various point location algorithms have been implemented [14]. A planar map (an embedded planar graph, or EPG) has been implemented [10], with operations that preserve planarity. The EPG is built on top of an ordered graph from the library. On top of the EPG, there are algorithms for orthogonal drawings of graphs [7] and for finding the shortest path between two points in the interior of an arbitrary polygon [2]. Finally, a vector class with efficient insertion at arbitrary positions has been implemented [8].

Acknowledgments

We would like to thank the entire JDSL team for their work on the project—in particular, Andy Schwerin and Maurizio Pizzonia for their constructive criticism of this paper, and John Kloss for helpful comments regarding topics related to the paper.

References

1. R. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. In *Proc. ACM Symp. Computer Science Education*, 1999.
2. J. Beall. Shortest path between two points in a polygon. http://www.cs.brown.edu/courses/cs252/projects/jeb/html/cs252proj.html.
3. M. Boilen, A. Schwerin, and J. Kloss. Personal communication.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.
5. A. Fabri et al. The CGAL kernel: A basis for geometric computation. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, pages 97–103, May 1996.
6. N. Gelfand, M. T. Goodrich, and R. Tamassia. Teaching data structure design patterns. In *Proc. ACM Symp. Computer Science Education*, 1998.
7. N. Gelfand and R. Tamassia. Algorithmic patterns for graph drawing. In *Proc. Graph Drawing '98.* Springer-Verlag, to appear.
8. M. T. Goodrich and J. Kloss. Tiered vector: An efficient dynamic array for JDSL. Poster at *OOPSLA'98.*
9. M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* Wiley, New York, NY, 1998.
10. D. Jackson. The TripartiteEmbeddedPlanarGraph. Manuscript.
11. K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.
12. M. Nissen. Graph iterators: Decoupling graph structures from algorithms. Diploma thesis, Max-Planck-Institut für Informatik, Univ. Saarlandes, Saarbrücken, Germany, 1998.
13. B. Stroustrup. *The C++ Programming Language (3rd Edition).* Addison-Welsey, Reading, MA, 1997.
14. R. Tamassia, L. Vismara, and J. E. Baker. A case study in algorithm engineering for geometric computing. In *Proc. Workshop on Algorithm Engineering*, pages 136–145, 1997.
15. K. Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proc. OOPSLA '97*, pages 34–48, 1997.
16. CS 16 home page. http://www.cs.brown.edu/courses/cs016.
17. ffGraph home page. http://www.fmi.uni-passau.de/~friedric/ffgraph/main.shtml.
18. Java 1.2 API. http://java.sun.com/products/jdk/1.2/docs/api/index.html.
19. JDSL home page. http://www.cs.brown.edu/cgc/jdsl.
20. JGL home page. http://www.objectspace.com/jgl.
21. LEDA home page. http://www.mpi-sb.mpg.de/LEDA.

Object-Oriented Design of Graph Oriented Data Structures[⋆] (Extended Abstract)

Maurizio Pizzonia and Giuseppe Di Battista

Dipartimento di Informatica e Automazione, Università di Roma Tre
via della Vasca Navale 79, 00146 Roma, Italy
{pizzonia, gdb}@dia.uniroma3.it

Abstract. Applied research in graph algorithms and combinatorial structures needs comprehensive and versatile software libraries. However, the design and the implementation of flexible libraries are challenging activities. Among the other problems involved in such a difficult field, a very special role is played by graph classification issues.
We propose new techniques devised to help the designer and the programmer in the development activities. Such techniques are especially suited for dealing with graph classification problems and rely on an extension of the usual object-oriented paradigm. In order to support the usage of our approach, we devised an extension of the C++ programming language and implemented the corresponding pre-compiler.

1 Introduction

Libraries dealing with combinatorial structures like graphs are quite often medium-large systems because of the great variety of known algorithms and because of the possibility of graphs to be classified in a plethora of ways. Also, most of the existing algorithms require a large amount of software to be implemented. To give an example, the GDToolkit [20] system consists of about 40,000 lines of code. A limited list of object-oriented systems somehow related to graphs includes: ALF [5], ffgraph [6], JDSL [15], Leda [10], and LINK [4].

Modern large software systems are usually built using object-oriented technologies and methodologies. The main motivations of this choice are: development and maintenance time reduction, reusability of components, and extensibility of the system. A key issue here is to increase the abstraction level of the code by modeling the reality of interest with high-level representation structures.

The above mentioned goals are especially difficult to meet in the application domain that we are considering. Our opinion is that one of the main reasons for this difficulty is in the lack, in currently used object oriented methods and languages, of representation primitives suitable for representing complex combinatorial structures. The experience of designing a large library of graph algorithms

⋆ Research supported in part by the ESPRIT LTR Project no. 20244 - ALCOM-IT.

offers a clear perception of this type of modeling problems, that are especially related to the classification of graphs. We give an overview and an analysis of such problems in Section 2.

The main contributions of this paper can be summarized as follows:

- We propose an extension of the object-oriented paradigm, specifically tailored to tackle problems arising in the classification of graphs (Section 3). Such an extension (called *ECO*) consists of new representation primitives.
- We show how to use ECO to build large graph libraries. This is done by suggesting several design schemas. A project experience conducted with such design schemas has put in evidence an improved flexibility of the system and a decrease of the development time. The saving of time was mainly due to the high abstraction level of the produced code (Section 4).
- We describe an extension of the C++ programming language that embodies the ECO primitives. Further, we present a pre-compiler that we have implemented for ECO, which gives an easy way to use the concepts of the new paradigm (Section 5).

2 Classification Problems for Graphs

Classification is a key part in software libraries that manage combinatorial structures like graphs. In fact, each graph algorithm is suited to work on a specified class of graphs for which given properties hold. However, notwithstanding its importance, the actual role of graph classification is usually reduced by the flexibility and the extensibility problems that a large hierarchy of graph classes introduces (see, e.g. [5]).

In the following we briefly analyze the most, in our opinion, relevant aspects of the interplay between graph classification problems and the object-oriented paradigm.

2.1 Inheritance and Subclasses

Various authors point out that saying that B is a subclass[1] of A can have several meanings (for example see [12,17]).

We refer to *extension* subclassing if new information is added to B and each instance of A can become an instance of B if suitably equipped with new information (e.g. in this sense Labeled Graph is a subclass of Graph). In the following we call *extension* the “added part” of B and *support* the “inherited part”. This is very different from the case where no new information is added to B and there are instances of A that are not instances of B, i.e. B is a strict subclass of A in the mathematical sense (e.g. Connected Graph is a subclass of Graph). In this case we talk about *restriction* subclassing.

[1] We do not distinguish between classes and types (see [1]) because it is not relevant to our discussion. However, what we say can be easily extended to handle such distinction.

The inheritance mechanisms of most object-oriented languages are well suited in modeling extension subclassing, but fail in modeling restriction subclassing. Namely, the restriction subclassing infringes the soundness of polymorphism because not all values that are legal for objects that belong to A are also legal for objects that belong to B. I.e. B violates the *Liskov substitution principle* [9]. Hence, the programmer must either exploit the exception mechanism when the invariant of B is violated, or, even worse, must rely on the user to behave correctly (such policies are used in libraries like Leda [10], JDSL [15], LINK [4], and GDToolkit [20]). Such sort of situations induce the programmer to modify A in order to consider particular situations that are actually related to B, thus violating another well known principle, the *open-closed principle*: "every module must be closed to modification and open to extensions" (first stated in [11]).

Consequently, restriction subclassing cannot be modelled in any elegant way using standard inheritance mechanisms, unless we make use of the functional-style design[2], which is known to be rather inefficient.

2.2 Crossed Classifications

Complex combinatorial structures like graphs can be classified in a large number of ways. If more than one independent classification exist, *crossed* classes must be created to give to the system complete classification capabilities. For example, Fig. 1 shows how combining connectivity, planarity, and orientation may give raise to several crossed classes, even if none of such classes adds new features to the inherited ones. Of course, the number of crossed classes increases exponentially with the classification coordinates.

This problem is especially relevant because usual object-oriented paradigms do not allow objects to belong to more than one class. The only exception is the usage of multiple inheritance that, however, is not helpful in decreasing the number of subclasses.

Several authors address this problem. For example, [12] uses a "delegation technique", while [7] uses the "decorator pattern". However, the proposed solutions always show an overwhelming burden for the programmer and/or a quite tricky game of conventions and limitations that cannot be expressed in a programming language.

2.3 Multiple Decorations

There are two types of subclasses defined with the extension subclassing.

1. The added information is (if needed) univocally determined by the support. For example, if the superclass is a graph, then the subclass might be a graph equipped with the set of the connected components. Given a graph, such information is univocally determined.

[2] A functional-style design avoids any changes to existing objects; methods that change the graph actually produce a new copy of the object. The class of the new copy can also be different from the starting class.

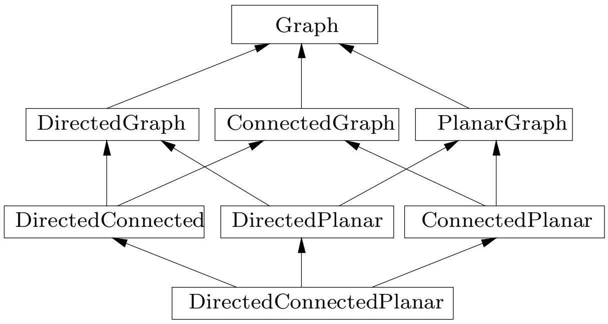


Fig. 1. A hierarchy affected by the crossed classification problem.

2. The added information is not univocally determined by the support. For example, an *st*-numbering of a graph is (at least partially) arbitrary.

In Case 1, it is reasonable to exploit the inheritance for adding the information about the set of the connected components.

Case 2 is simple if we do not need to add more information (e.g. other *st*-numberings or orientations). If this is true, then the usage of the inheritance remains a reasonable choice. However, if we require the support to be extended with other information of the same type, then it is not possible to use the inheritance. In other words, referring to the example, the inheritance forces to specify just one *st*-numbering, while some applications may require to *st*-number the vertices of the graph in several ways at the same time.

A possible solution is to instantiate one new object for each new information (e.g. for each *st*-numbering) which contains the new information. It is also needed to state links between elements of the first object (e.g. the vertices) and the new objects (e.g. the numerical labels) by means of suitable structures (e.g. hash-tables). Even though many libraries provide mechanisms to state such links, it may become difficult (or even practically impossible) to maintain the consistency among the objects when the graph changes.

2.4 Promotion Efficiency

During the execution of an algorithm a graph may change its properties. This has the effect of virtually moving the graph to another class, where new methods become meaningful. Further, even if the properties of a graph do not change, it is sometimes useful, for efficency reasons, to dynamically equip the graph with new capabilities when they are needed.

In general, when an object is *promoted* (*demoted*) from a class into its child (parent), it requires a complete copy of the support. For example, if the graph is recognized to be planar, and we decide to promote it into the suitable class to take advantage of the new properties, a complete copy of the graph (with its "heavy" implementation structures) must be done. If the internal implementation is the same (e.g. linked), a large amount of time is wasted to make the copy. This happens because the membership relation is usually not dynamic in the available programming languages. The class of an object is statically defined at the moment of its creation.

This problem can be addressed by means of the "bridge pattern" [7]. This technique keeps separated the interface and the implementation structures using two distinct objects linked by a pointer. In this way, it is very efficient, although not very elegant, "to steal" the implementation from a graph to create another graph that has an extended interface. This technique is used in GDToolkit [20].

3 New Representation Primitives for Graphs Classification

The *ECO* (Extender and Classer Oriented) paradigm is an extension of the usual object-oriented design paradigm. This means that ECO imports the set of representation primitives usually defined in object-oriented design methodologies and enlarges such a set with new concepts. The new concepts are expecially targeted to address the problems illustrated in Section 2.

3.1 Extenders and E-Methods

The *extender* is the main new concept introduced by ECO. Extenders are classes that have the following additional features. An extender is associated to a *support-class*. An instance of an extender, called *extender-object*, is created over a *support-object* belonging to the support-class. It cannot be created before the support-object and it shall be destroyed before the support-object. Thus, we can talk about the current extender-objects of a given support-object. Given a support-class C and an extender E for C, it is possible for an instance of C to be support for more than one instance of E. To give an example, a class Graph can be support-class for the extenders Orientation and Embedding, so, more than one orientation and/or embedding may be instantiated over a given graph.

When an event occurs, then the support-object can notify it to its current extender-objects. The notification is done by using specific methods, called *E-methods*. An E-method signature is defined in a support-class and the behavior is specified in its extenders. An E-method equips an extender-object with two capabilities. An extender-object

– can change its state when the state of the support-object changes,
– can add constraints to the possible changes of the state of the support-object (using exception mechanisms)[3].

[3] Adding this kind of constraints we still violate the Liskov substitution principle, but we obtain great flexibility (see 4)

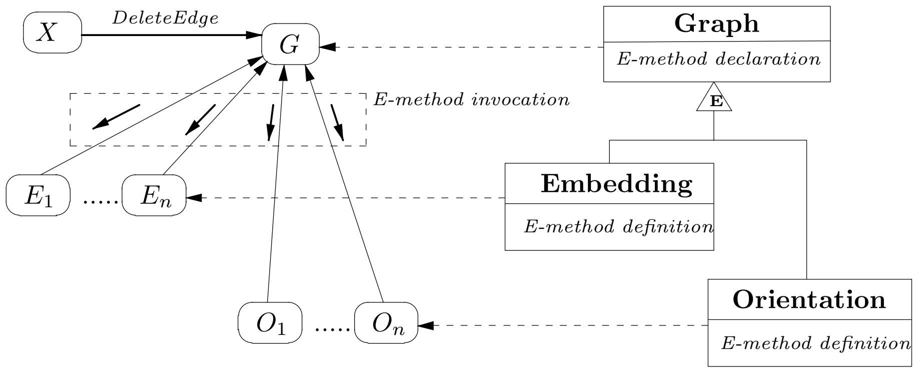


Fig. 2. In this mixed (classes-objects) diagram, dashed lines represent instance relationships (E_i are embeddings and O_j are orientations), thick arrows represent method (or E-method) invocations, plain lines represent extension relationships. When a method is invoked on a Graph G (for example *DeleteEdge*), the E-method mechanism can be used to notify the event to the current extender-objects.

E-methods are invoked only in the methods of the support-class. The invocation of an E-method triggers the execution of its behavior for each current extender-object (see Fig. 2). The behavior of the E-methods is allowed to modify only the state of the extender-object, which it is invoked for. The executions sequence is not specified. Intuitively, they can be considered parallel executions, while the support-object waits for the termination of all of them.

Fig. 3 shows the evolution of a system in which a graph G (a support-object) and some embedding and orientation $E_1, \ldots, E_n, O_1, \ldots, O_m$ for G (extender-objects) interact. The vertical direction represents the time. Each vertical line represents the evolution of an object. A vertical line becomes thick when a method is executed on the corresponding object. The E-method Post_AddVertex(v) is invoked from a method AddVertex( ) of G. This triggers the execution on each embedding and each orientation of the behavior of Post_AddVertex( ) associated to such objects. Observe that distinct extender-objects may have a distinct behavior depending on the extenders they belong to. So, in the example, orientations can have a behavior that is different from the behavior of embeddings.

The E-methods are most likely to be used in methods that change the state of the support-object. In fact, such changes can lead to inconsistencies in the current extender-objects. E-methods can update the extender-object or inhibith the change in the support, depending on design choices.

For each method that performs modifications on the state of the support-object the designer can provide an E-method that is called as soon as the method is called. The definitions of these E-methods are supposed to inhibit the modification (i.e. throwing an exception) if the extender constraints are violated. Two more E-methods can be provided in order to allow each extender-object to

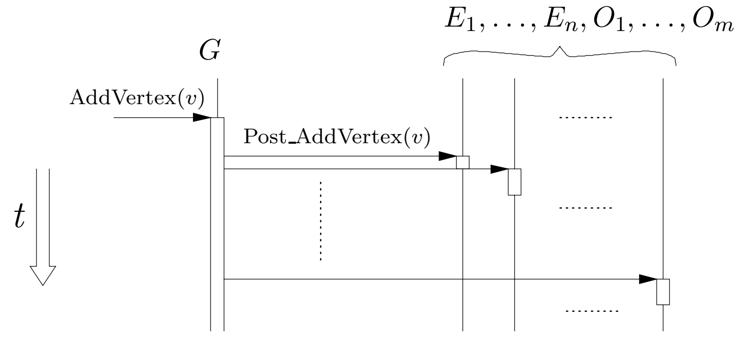


Fig. 3. Interaction between a support-object G and its extender-objects $E_1, \dots, E_n, O_1, \dots, O_m$.

update its state on legal modifications of the graph. These are called before and after the modification of the support-object.

Extenders are especially suited to support the extension subclassing where the multiple decoration problem arises (see Section 2.3). They are useful to dynamically change behavioral and/or structural aspects of an object during its life-time. Further, an extender can be developed even much time after the development of its support-class. This allows to add capabilities to the system without changing any other part of it. The new extender will coexist with other extenders already present in the system, and will work in conjunction with them.

3.2 Classers

The *classer* is the second new concept. A classer is a constrained extender for which just one instance is allowed for a given support-object[4]. In spite of the simplicity of the definition, classers are as important as extenders and give much more expressiveness to the paradigm.

Consider the case when new information, and operations related to it, have to be added to an object and such information can be univocally determined by its state by means of a functional dependency (for example the set of connected components, see Section 2.3). Also, consider the need of attaching the new information dynamically. In this case a dynamic classification system would be needed allowing an object to change its class during its life-time. Even if extenders seem to be a good choice to support dynamic classification issues, their capability of having more than one instance, for each support-object, yields several disadvantages.

The main problem is that, under the above conditions, all the extender-objects of a given support-object have the same state, because the information

[4] It is important not to misidentify the constraint for the classer with the *singleton* pattern [7]. More than one instance is admitted for classers overall, but only one for each support-object.

that they contain functionally depends on the state of the support-object. Single extender-objects cannot change independently because of their peculiar semantic. So, having more than one instance is not useful, inefficient, and conceptually misleading.

Then, we can state that the new information and operations are conceptually bound to the support-object. Nevertheless, the programmer has to refer explicitly to the extender-objects using variables or pointers, as he/she refers to any other object. Also, when classification hierarchies are non-trivial the programmer has to deal with much more objects than he/she needs.

Classers are extenders that can have at most one instance for each support-object. Hence, known the support-object and the classer, we can access the underlying extender-object without maintaining an explicit reference to the extender-object. Also, using classers restrain the user from instantiating the extender many times, thus avoiding inefficiency.

Classers are especially suited to support the restriction subclassing and the extension subclassing in the case of information univocally determined by the state of the support-object (see Section 2.3). We would like to point out that using classers in such situations avoids the crossed classification problem and the promotion efficiency problem.

4 Using the ECO Paradigm

In this section we present some techniques especially suited to build large systems using the ECO paradigm. We provide some examples that show how ECO addresses the problems mentioned in Section 2 while respecting the open-closed principle and keeping extendibility and flexibility.

4.1 Using Classers to Add Structure

A library for combinatorial mathematics cannot handle all the needs that the user could ask. So, it has to provide powerful and flexible ways to extend its capabilities without modifying the library itself.

Many features that a non naive user can ask require addictional structures to be maintained with a graph and updated when the graph changes (for example, connected components set, block cut-vertex tree, etc.). Such kind of structures are not basic features (there are many applications that do not require them) and could be an unnecessary burden if they are not really needed. For this reason, we prefer not to insert them into the main graph class.

Classers are particularly useful to address such problem. In fact, using classers yields the following advantages:

- the new information can be attached when needed, so, the user is not compelled to choose a cumbersome implementation at the instantiation time;
- the E-methods mechanism provides a clean way to dynamically update the structure and permits to place the updating code in the classer definition;

– the introduction of new classers does not require changes to old code and the new and the old classers can be used simultaneously without any sort of limitation, thus, obtaining good extendibility and flexibility.

Following the presented approach, classers turn out to be the natural containers for dynamic algorithms (for a similar, but ad hoc, approach see [2]).

4.2 Hierarchies of Classers and Extenders

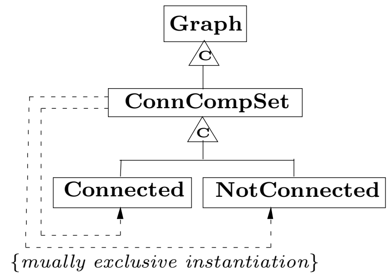


Fig. 4. ConnCompSet is a classer that dynamically manage the set of connected components. Connected and NotConnected classers represent the status of the graph. They act like an "hook" for attaching new classers or extenders that deals only with graphs for which the relative property holds.

In Section 3.1, we already provided an example of using extenders to model the concepts of orientation and embedding (permitting more instances at once of both), and in Section 4.1 we described how to use classers. Now, we describe how each extender and classer can be a support-class, as well, accepting its own extenders and classers, and thus, allowing even more expressiveness.

An extender can be used to associate new structures to an embedding in order, for example, to represent the orthogonal shape of the edges[5], and an orientations can be equipped with costs and capacities to became a flow network in conjunction with its support graph.

Using this technique the extender-objects can make up a tree. When a support-object changes its state, the consistancy in the lower levels of the extender-objects tree have to be maintained. E-methods can be easily used to address this problem.

The depth of such tree is usually not greater then three or four levels, and constant in any case (it is statically determined at compile time). On the other hand, there is no constraint to the degree of its nodes (i.e. the number of extender-objects). However, note that the number of extender-objects for a given support-object are very often constant for a given algorithm and, more precisely, it does

[5] In the Graph Drawing area, the sequence of left/right bends of an orthogonal drawing. The shape of a graph plays an important role in algorithms like GIOTTO [14].

not depend on the input size[6]. Due to the above considerations, the paradigm allows the update to be performed in constant time for a given algorithm.

The classers can also be used to represent restriction subclassing. In fact, we can associate a classer to a property and the presence of a classer instance means that the given property holds for the graph, hence, the graph belong to a specific mathematical class. In Fig. 4 we shows a design schema that permits to dynamically attach the connected component set structure, and then (only if such feature is present) to classify the graph as Connected or NotConnected. The instantiation of such classers can be ascribed to the ConnCompSet classer itself. In this way, we can provide a full-fledged module to handle connectivity with automatic dynamic classification[7], that also permits further restriction subclassing (attaching classers to Connected or NotConnected), or extenders that can futher exploit the concept of connected component (attaching extenders to ConnCompSet).

5 Supporting the ECO Paradigm with a Pre-Compiler

The introduction of a new paradigm is of little help if not supported by suitable programming tools. Among several possible alternatives, we have chosen to support the ECO paradigm with a pre-compiler[8]. Also, we introduced an extension of the C++ language which we call ECO C++. The pre-compiler generates code where new constructs are replaced by standard C++ code that emulates their semantic. Fig. 5 shows the entire compilation process.

This yields a number of advantages. The C++ language is a well know language with many libraries and tools already available and widely used. A pre-compiler permits to use the new paradigm with many of them. Further, it does not require the programmer to learn a completely new language, and because of the existence of good, portable and freely available compilers, the system works on a wide range of platforms.

[6] Using a set of extender-objects whose size grows with the input size is almost always a misuse. In fact, using extenders implies dynamical update through E-methods, which, in this case, takes linear time. Note that if we really need to update such a set of objects dynamically, extenders largely simplify the work and do not increase the computational complexity.

[7] At least two design choices are possible to deal with operation that disconnect the graph: an exeption can be thrown or the graph can be dynamically reclassified as NotConnected.

[8] For example, a pure methodological approach would force the user to write a lot of code related to the new concepts introduced by ECO, possibly more than the code dedicated to the problem itself.

Further, pure C++ is not powerful enough to permit to support ECO by means of a library. The need of the metalevel can be understood observing the peculiar declaration/definition system of the concept of E-methods.

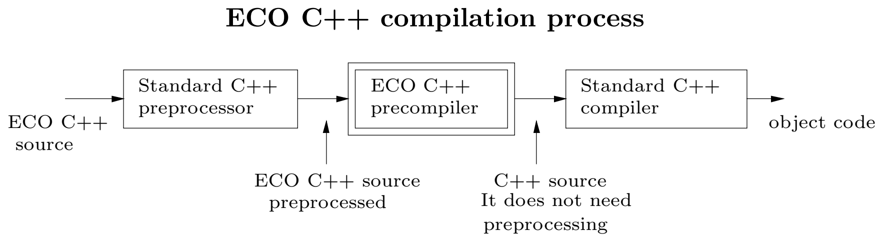

ECO C++ compilation process
ECO C++ source
Standard C++ preprocessor
ECO C++ precompiler
Standard C++ compiler
object code
ECO C++ source preprocessed
C++ source It does not need preprocessing

Fig. 5. The ECO C++ compilation process. A pre-compilation phase has been inserted after the preprocessing phase and before the usual compilation phase.

5.1 The ECO C++ language

ECO C++ supports the concept of extenders and E-methods by means of new syntactic primitives. The programmer declares a support-class and an extender by using a suitable syntax as in the example shown in Fig. 6 and 7 . The keyword **extensible** introduces a declaration of a support-class in which E-methods can be declared. E-methods are denoted by the keyword **extend**. The behavior for such E-methods may be defined for each extender (the default behavior is “do nothing”). The E-methods invocation is performed by using the keyword **call_e_method( )** as shown in the same figure. When the control flow reaches such keyword the execution is dispatched to the behaviors specified in each current extender-object.

The keyword **extend** denotes an extender for a specified class, when placed in the head of a class declaration. Each constructor for the extender shall have, as first parameter, a reference or a pointer to the support-class. The actual parameter will be the support-object of the extender-object that is being creating (see Fig. 7). The precompiler automatically generates code that updates the list of the extender-objects for the given support-object. The behavior for the E-methods may be defined by the programmer as he/she does for usual methods.

The usage of extenders is quite easy. As it can be noticed in Fig. 8, extenders are instantiated as usual classes, but for the special meaning of the first parameter in the constructor. Further, the extender-objects must be destroyed before the support-object. Note that this automatically happens if the storage class of the instance is **auto** (i.e. the allocation is performed on the stack) because the destruction order for the objects declared in a block is the reverse of the declaration order, according to the standard C++ semantic [19,13].

ECO C++ supports the classer concept by means of the keyword **dynamic**. Fig. 9 shows a declaration for a classer. All constructors have to be private because the presence of the classer states that a given property holds. Here, we suggest to introduce a class method (declared **static**) that performs the test and eventually instantiates the classer, if admissible. We call such class method a pseudo-constructor. The programmer does not need to maintain references to classer instances. A special syntax makes easy to access methods of a classer and to test if a classer is instantiated for a given support-object:

```
class Graph: extensible
  {
  public:
    // a (non E-) method
    Vertex Add_New_Vertex();
       {
       call_e_method( Check_Add_New_Vertex() );
       call_e_method( Pre_Add_New_Vertex() );
       ...
       call_e_method( Post_Add_New_Vertex(v) );
       return v;
       }
    // some E-methods
    extend void Check_Add_New_Vertex()
    extend void Pre_Add_New_Vertex()
    extend void Post_Add_New_Vertex(Vertex v)
    ...
  };
```

Fig. 6. An example of declaration of a support-class Graph written in ECO C++.

support_object.**{***classer***}**.*method*(...)	call the method
support_object.**{***classer***}**	return true if the classer is instantiated.

Fig. 10 shows the usage of the classer declared in Fig. 9. It shows how to invoke a pseudo-constructor, how to test if a classer is instantiated, and how to preform a classer method calling.

Similar concepts are implemented in other systems, but they differ from ECO in important aspects. The concepts of *signal* and *slot*, in Qt [16], are a support for dynamic updating comparable to the E-methods system, but they were born in a quite different field (GUI object component) where no needs for complex classification arise. So, signals and slots alone are not useful in solving the exposed classification problems. On the other hand, the Java member classes [8] play a role similar to the extenders in ECO but they lack of dynamic updating capabilities[9]. In some sense, ECO represents a fusion of these two attracting approaches that permits completely new developments.

[9] Instances of a Java inner class are always associated to another object (the "support-object", using ECO terminology), but Java does not provide a mechanism to dynamically maintain the consistency between the two objects. Further, Java inner classes shall be declared with the container class. So, it is impossible to create a new module with a new inner class.

```
class Labeling: extend Graph
  {
  public:
    // 'g' compulsory parameter
    Labeling(Graph& g) {};
    ~Labeling() {}

    // E-methods definitions
    void Post_Add_New_Vertex(Vertex v)
      { /*initialize with an empty label*/ }
    ...
      // other methods
    Set(Vertex, char*){}
    Get(Vertex);
    ...
  };
```

Fig. 7. An example of declaration of an extender of Graph (see Fig. 6) written in ECO C++.

5.2 The run-time support of ECO C++

The run-time support for the ECO C++ language has to maintain low level structures that represent the relationships between each support-object and its current extender-objects. The following operations have to be performed using such structures:

– creation of an extender-object,
– deletion of an extender-object,
– invocation of an E-method.

It is easy to implement a support system that is based on linked lists and performs a creation and a deletion taking $O(1)$ time.

On the contrary, the invocation of an E-method takes $O(k)$ time, where k is the number of the current extender-objects, if the E-method execution takes $O(1)$ for each extender-object, which is the most common situation. However, k is most of the time only dependent on the algorithm, as it arises from the analysis that we made in Section 4.2.

6 Conclusions

Developing libraries of algorithms is a complex task requiring expertise both in algorithmics and in software engineering.

In this paper we have presented techniques that are mainly related to solving graph classification problems in libraries of graph algorithms. The introduced representation primitives allow the designer to model dynamically changing properties of graphs with little effort. They permit to overcome most of the

```
main()
    {
    Graph G;    // A graph
    ...
                      // Two labelings
    Labeling L1(G);                  // automatic allocation
    Labeling* L2= new Labeling(G);   // dynamic allocation
    ... // use 'L1' and 'L2' accessing their methods as usual
    delete L2;    // A dynamically allocated object has to be deleted, as usual.
    ...
    } // here L1 is deleted before G
```

Fig. 8. An example of usage of the extender and the support-class declared in Fig. 8.

limitations that come out with standard object-orientation, and to reach many of its aimed goals, like extensibility (the open-closed principle) and flexibility (reuse of preexistent modules in different contexts). Further, the primitives allow to elegantly (in our opinion) express known concepts like the *observer pattern* [7] and the *data accessor* [18].

The pre-compiler for the ECO C++ language permits to easily apply the new concepts avoiding many of the drawbacks that typically arise when using new paradigms. The pre-compiler is available on the Web at the address below:

`http://www.dia.uniroma3.it/~pizzonia/eco`

The presented ideas have born in the Graph Drawing area and specifically during the development and usage of the GDToolkit [20] system and have already been applied within the same project.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1996.
2. D. Alberts, G. Cattaneo, G. F.Italiano, U. Nanni, and C. D. Zaroliagis. A software library of dynamic graph algorithms. In R. Battiti and A. A. Bertossi, editors, *Proceedings of "Algorithms and Experiments" (ALEX98) Trento, Italy, February 9-11, 1998*, pages 129–136, 1998.
3. Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography, June 1994.
4. Jonathan Berry, Nathaniel Dean, Mark Goldberg, Gregory Shannon, and Steven Skiena. Graph drawing and manipulation with LINK. In G. Di Battista, editor, *Graph Drawing (Proc. GD' 97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 425–437. Springer-Verlag, 1998.
5. Paola Bertolazzi, Giuseppe Di Battista, and Giuseppe Liotta. Parametric graph drawing. *IEEE Transactions on Software Engineering*, 21(8):662–673, August 1995.

```
class Graph: extensible
   {
   ...
   extend void Check_Add_Edge(); // An E-method.
   };

class PlanarGraph: dynamic Graph
   {
   private:
      PlanarGraph( Graph& ) {...}; // Clients cannot call this constructor.

   public:
        // The pseudo-constructor performs the instantiation (if admissible).
      static void Test(Graph& g)
        {
          ....  // planarity test
          new PlanarGraph( *this );
        };

        // Check if planarity is preserved and throw and exception if needed
      void Check_Add_Edge() {...};

      int Get_number_of_faces(); // specific planar graph method
   };
```

Fig. 9. A delcaration of a classer for managing the planarity property.

```
main()
    {
    Graph g;
    ...
    PlanarGraph::Test(g); // planarity test (using pseudo-consructor)

    if ( g.{PlanarGraph} )     // is 'g' recognized as planar?
         {
         cout << "g is planar, # of faces="
              << g.{PlanarGraph}.Get_number_of_faces() // classer method calling
              << endl;
         }
    } // all classers of 'g' are automatically destroyed
```

Fig. 10. An example of usage of the classer declared in Fig. 9.

6. C. Friedrich. The ffGraph library. Manuscript, Lehrstuhl für Informatik, Univ. of Passau, December 1995.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.
8. James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
9. Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.
10. K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.
11. Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., 1988.
12. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice Hall, Englewood Cliffs, 1991.
13. Bjarne Stroustrup. *The C++ Programming Language: Second Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1991.
14. R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, January/February 1988.
15. Roberto Tamassia and Luca Vismara. A case study in algorithm engineering for geometric computing. Technical Report CS-97-18, Department of Computer Science, Brown University, December 1997. Wed, 24 Jun 1998 13:22:55 GMT.
16. Troll Tech. Qt: a GUI Software Toolkit, 1998. `http://www.troll.no/`.
17. Peter Wegner. The object-oriented classification paradigm. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming.* The MIT Press, 1987.
18. Karsten Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 34–48, New York, October5–9 1997. ACM Press.
19. ANSI X3J16. American national standard for information systems — programming language — C++. Approved standard, ANSI.
20. ALCOM-IT – GDToolkit, 1999. Third University of Rome.
`http://www.dia.uniroma3.it/~gdt`.

A Case Study on the Cost of Geometric Computing[⋆]

Stefan Schirra

Max-Planck-Institute for Computer Science, Saarbrücken, Germany
stschirr@mpi-sb.mpg.de

Abstract. We report on experiments on the performance of various geometry kernels for the two-dimensional convex hull problem. We consider how programming techniques and the choice of geometric representation affect performance. In particular we investigate the cost of exact computation. We use C++ as the implementation language. Our experiments are largely based on CGAL.

1 Introduction

In introductory courses and textbooks on computational geometry planar convex hull is often the first problem studied because the problem and the algorithmic approaches to solve it are easy to grasp. Many different important algorithm design principles have been successfully applied to planar convex hull computation, and so planar convex hull is often used to illustrate such techniques.

The convex hull of a set of points is the smallest (with respect to set inclusion) convex set containing the points. The convex hull of a finite set of points $\mathcal{S}$ in the plane is a convex polygon with vertices in $\mathcal{S}$. The task in the planar convex hull problem is to find the hull polygon. A simple representation of this polygon is the (counter)clockwise sequence of its vertices. A point in $\mathcal{S}$ is called extreme (with respect to $\mathcal{S}$), if its removal changes the convex hull, or equivalently, if it is a vertex of the hull polygon of $\mathcal{S}$. We can state the planar convex hull problem as follows: Given a set (or sequence) of input points, report the counterclockwise sequence of extreme points.

The planar convex hull problem is a basic subroutine in algorithms for other problems, e.g. the width of a point set. It is a well studied problem. The lower bound for computing the convex hull of a set of n points is $\Omega(n \log h)$, where h is the number of extreme points [32]. There are algorithms that achieve this bound [17,32]. The history of planar convex hull computation dates back at least to the early 70s when Graham's scan algorithm [25] and Jarvis's gift-wrapping algorithm [30] were published. In the late 70s and early 80s further algorithms and variations on previous methods [45] were published [1,2,3,5,6,7,8,15,20,21,24,27], [28,29,33]. In the early 80s the lower bound was proven and Kirkpatrick and

⋆ This work is partially supported by the ESPRIT IV LTR Projects No. 21957 (CGAL) and 28155 (GALIA)

Seidel presented a theoretically optimal algorithm [32]. There were also experimental studies on the efficiency of the planar convex hull algorithms in practice [5,27,36]. Since the mid 80s, there was no news on planar convex hull front, with the notable exception of Chan's optimal algorithm [17].

When you have to implement a convex hull algorithm, you will probably want to implement an efficient algorithm. Although the theoretical worst-case asymptotic running times don't tell you all about the actual performance of an algorithm for the small or medium size instances you want to solve, they are a good help. But there is a bit more than choosing an efficient algorithm when we actually have to implement an algorithm: For example, we have to address questions like "How do we store the points?", "How do we represent points?", "How should we implement our predicates?", and we would like to know how the options we have affect performance. This is the issue addressed in this case study. The generic CGAL library [16] provides a unique framework for studying such questions. In the next section, we describe our test environment. Section 3 states the results of our experiments.

2 The Test Bed

The convex hull problem as stated above permits a generic communication with the data structures containing the input point set and the output sequence of extreme points using the iterator concept used in the Standard Template Library (STL), see e.g. [35]. We are interested in the performance of predicate evaluation and the influence of point representation and design issues. In our experiments, we therefore used code that is parameterized by the point type on which the algorithms operate and by the predicates used to do the computations. These types, point type and predicate types, were collected in a single parameter called *traits* type. For our experiments, we used implementations of several convex hull algorithms, some of them available as part of the current CGAL release [16]. The declaration of all our convex hull functions looks like

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
convex_hull_points( InputIterator first, InputIterator beyond,
                    OutputIterator result,
                    const Traits& ch_traits);
```

where `InputIterator` and `OutputIterator` are the iterator types used for the communication between data containers and algorithms and the `Traits` parameter is supposed to provide point type `Traits::Point_2` and predicate types, for instance predicates like `Traits::Leftturn(p,q,r)`, where `p`, `q`, and `r` are of type `Traits::Point_2`. The traits parameter can be seen as a geometry kernel, that provides at least the functionality required by the algorithm.

For the experiments we used an environment that let us combine various geometry kernels with various algorithms. A snapshot of the control panel of the test environment is shown in Fig. 1. We had 11 parameterized convex hull algorithms and 31 geometry kernels, plus 4 non-parameterized convex hull algorithms, i.e. in total we had 345 convex hull algorithms at hand for our experiments.

The test code was compiled with `egcs` (release 1.1.1, i.e. `egcs-2.91.60`) and run on a SUN Ultra 10 with a 333 Mhz processor and 128 MB RAM under Solaris operating system. In our tests we used CGAL with one more level of inlining than the default of CGAL. Additional levels of inlining can be easily switched on in CGAL. We compiled with optimization level `-O3`. Without the optimization we observed a significant slow down by a factor of two to six.

We used four different generators for point sets:

(1) uniformly distributed points in a square,
(2) uniformly distributed points in a disk,
(3) non-uniformly distributed points in a disk (more points near the border),
(4) almost cocircular points.

If not stated otherwise, we generated points with integral coordinates of at most 20 bits. In order to get measurable running times on small point sets, we made several iterations of convex hull computation.

Next, we briefly describe the generic implementations of the different algorithms and the different geometry kernels we used. We start with the algorithms.

`Akl-Toussaint` (`AT`) the CGAL-1.2 implementation [16] of the *throw-away*-algorithm by Akl and Toussaint [3]. It starts by computing the 4-gon[1] spanned by the extreme points in x- and y-directions. Points inside this 4-gon are thrown away, the others are assigned to subproblems corresponding to the edges of the 4-gon. To solve a subproblem, points assigned to the subproblem are sorted and subhulls are then computed using Graham's scan. The implementation uses a predicate called `Right_of_line` to check sideness with respect to a line, two predicates to compare points lexicographically (one with preference to x-coordinate, one with preference to y-coordinate) for computing the initial points and sorting points in the subproblems, and a leftturn predicate to check orientation of three points. Internally STL-containers of type `vector<Traits::Point_2>` are used, i.e., points are copied, not pointers to points.[2] The sorting is done using the sort routine from the STL coming with the compiler.

`waste Akl-Toussaint` (`wAT`) like `Akl-Toussaint`, but now 4 arrays are used to maintain pointers to points. Each array allocates space for n pointers, where n is the number of input points. That's why we call this variant waste.

`waste Akl-Toussaint 8` (`wAT8`) like `waste Akl-Toussaint`, but now initially an 8-gon is computed in order to discard points. The additional points are the extreme points in the diagonal directions of the Cartesian coordinate system. For this task, two additional predicates are used.

`Graham-Andrew` (`GAw`) Andrew's variant [7] of the Graham scan [25]. All points are first sorted lexicographically. By a Graham scan over the sorted sequence of points, the subhull below the line segment from smallest to largest point is computed, and then by a scan in reverse order the subhull above that segment. During the scan, the heuristic described in [37] is used.

[1] might be degenerate, i.e. a triangle, a line segment, or even a point (if all input points are equal)

[2] Of course, using an appropriate traits class, you can always let the algorithm maintain pointers instead of points.

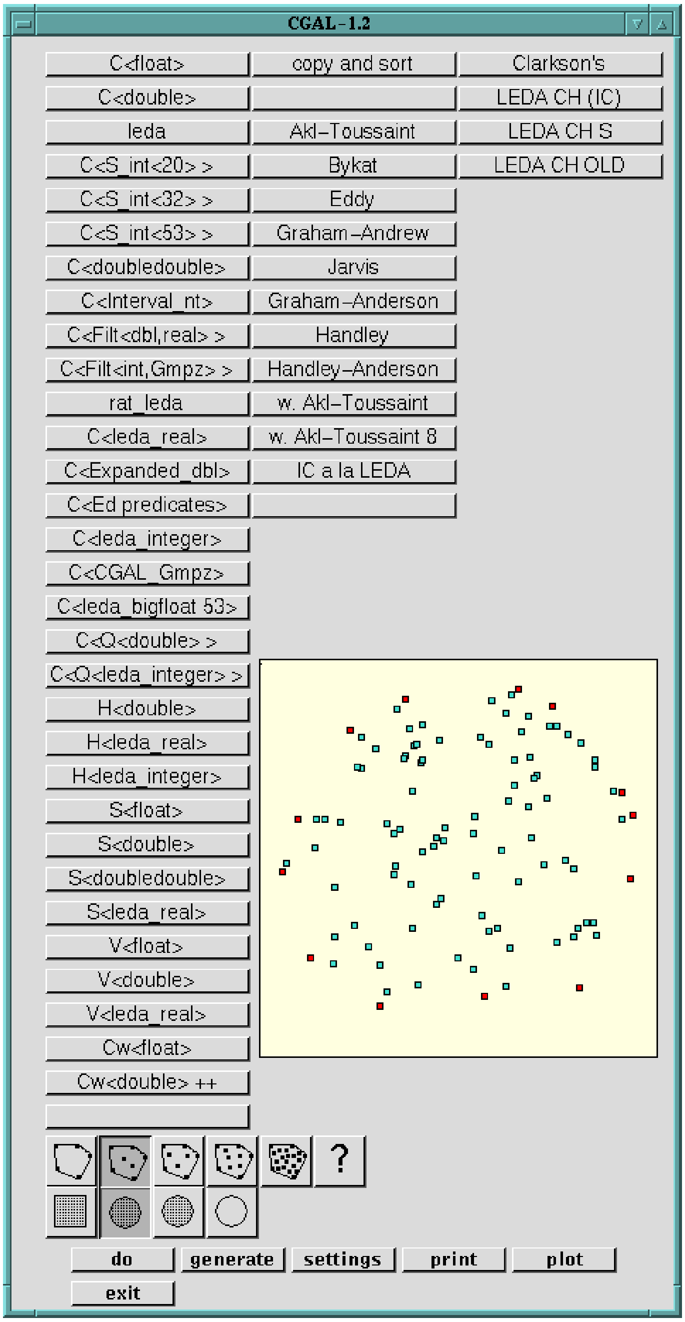


Fig. 1. The control panel of our test environment. In the left column you can select geometry kernels to instantiate the generic algorithms in the middle column. In the right column further algorithms (using Cartesian coordinates of type `double`) can be added.

`Graham-Anderson` (`GAs`) Anderson's variant [6] of the Graham scan. The lexicographically smallest point w is computed. Then all points are sorted in rotation order around w, where ties are resolved by (inverse) distance to w. A predicate is used to do comparison with respect to the rotation order around w. In all tested traits, this predicate was implemented using the orientation test. Finally, a Graham scan over the sorted sequence is done.

`Handley` (`Ha`) as suggested by Handley [28], this algorithm mixes discarding and sorting points. First extreme points in x-direction are computed. The goal of the next step is to compute the lower hull. Incrementally points are inserted in an (unbalanced) binary search tree starting with the extreme points computed initially, where in each step while walking down the tree points are discarded if they lie above the line segment through their current neighbors in the subsequence inspected so far. After processing all points a Graham scan over the sorted subset of points is done. The upper hull is computed analogously. The implementation internally maintains pointers to points.

`Handley-Anderson` (`HAs`) applies the idea of mixing discarding and sorting points to Anderson's variant of the Graham algorithm. Comparison of points is with respect to the rotation order around the lexicographically smallest point.

`Bykat` (`By`) CGAL implementation of Bykat's algorithm [15], which is a two-dimensional quickhull algorithm just like Eddy's algorithm [21,41]. The algorithm initially computes the two extreme points in x-direction and partitions the point set into the set of points above this line and the set of points below it. This gives rise to two subhull problems. In general, a subproblem to be solved in a quickhull step has an associated line segment, whose endpoints p and q are extreme points, and a associated set of points, which are candidates for extreme points between the segment endpoint. A subproblem is processed as follows: If the associated point set is not empty, the point r with maximum distance to the line segment is computed. Points inside the triangle spanned by the line segment pq and r are thrown away. The set of remaining points is partitioned according to the segment between p and r. This way, two new subproblems are formed. In contrast to the implementation of Eddy's algorithm, this version is non-recursive. Internally, STL-vectors are used for stack and point containers. Furthermore, STL-algorithms like `partition` are used to re-arrange points in the containers.

`Eddy` (`Ey`) CGAL implementation of Eddy's algorithm [21]. Analogous to `Bykat`, but subproblems are solved recursively. Internally, STL-lists are used.

`Jarvis` (`Ja`) CGAL implementation of Jarvis's march [30]. Starting with an initial extreme point, subsequent hull points are computed one by one by computing the minimum point with respect to rotation order around the so far last computed hull point. Uses the same predicate for rotation order as `Graham-Anderson`.

`IC_ala_LEDA` (`IC`) Slightly modified and parameterized version of the LEDA [39,38] convex hull algorithm using incremental construction. Besides a test for equality of points it uses an orientation predicate only.

For a more comprehensive description of these algorithms we refer the reader to the standard textbooks on computational geometry and the original papers.

We used the following traits arguments in our experiments:

C<float> uses the Cartesian CGAL kernel with single precision floating point arithmetic. It uses point type CGAL_PointC2<float> and the CGAL predicates corresponding to this point type. The parameterized Cartesian CGAL kernel uses reference counting.

C<double> uses the Cartesian CGAL kernel with double precision floating point arithmetic.

leda uses the two-dimensional floating-point geometry kernel of LEDA. This kernel uses double-precision floating-point arithmetic with Cartesian representation and uses reference counting.

C<CGAL_S_int<20> > uses the Cartesian CGAL kernel with a special number type called CGAL_S_int< N> for N= 20 and specialized predicates. The specialized predicates for number type CGAL_S_int< N> use a static floating-point filter (see e.g. [23]) based on the assumption that the coordinates are integral with at most N bits. Double precision floating-point arithmetic is internally used in number type CGAL_S_int< N>. If the filter fails, computation is re-done with exact arithmetic using leda_integer.

C<CGAL_S_int<32> > as above, with N= 32 (corresponding to size of int).

C<CGAL_S_int<53> > as above, with N= 53 (corresponding to double precision).

C<doubledouble> uses the Cartesian CGAL kernel with extended precision floating-point arithmetic. The number type doubledouble [10] implements a floating point arithmetic with about 30 decimal places.

C<CGAL_Interval_nt_advanced> uses the Cartesian CGAL kernel with a "number type" based on interval arithmetic, cf. CGAL-documentation [16]. Whenever intervals are not disjoint in comparison operations, the represented numbers are considered to be equal. Interval endpoints are of type double.

C<Filtered_exact<double,leda_real> uses the Cartesian CGAL kernel with number type CGAL_Filtered_exact<double,leda_real>. With this number type special predicates are used. Coordinates are stored as doubles. In the predicates, interval arithmetic (with interval endpoints of type double) is used as a filter to speed up computation. If the computation with interval arithmetic does not suffice to make a decision, coordinates are converted to leda_real and the predicate is evaluated with this number type.

C<Filtered_exact<int,CGAL_Gmpz> uses the Cartesian CGAL kernel with number type CGAL_Filtered_exact<int,CGAL_Gmpz>. Coordinates are stored as ints. In the predicates, interval arithmetic (with interval endpoints of type double) is used as a filter to speed up computation. If the computation with interval arithmetic does not suffice to make a decision, coordinates are converted to arbitrary precision integer type CGAL_Gmpz and the predicate is evaluated with this number type. The number type CGAL_Gmpz is a wrapper for the Gnu multiple precision arithmetic gmp [26]. The wrapper layer uses reference counting.

rat_leda uses the two-dimensional rational geometry kernel of LEDA. This kernel maintains homogeneous coordinates as arbitrary precision integers (leda_integer). It uses a semi-dynamic floating-point filter to speed up computation.

C<leda_real> uses the Cartesian CGAL kernel with number type leda_real [12,13,14], which provides exact decisions under the arithmetic operations $+, -, *, /$, and $\sqrt[k]{\ }$. leda_reals use adaptive evaluation and internally maintain expression dags for re-evaluation. A very convenient way to use exact computation.

C<Expanded_double> uses the Cartesian CGAL kernel with a number type which is called CGAL_Expanded_double. In this number type coordinates are maintained as sums of double precision floats. It is based on ideas and partially also on public domain code by Shewchuk [43,44], which extends earlier work by Priest [42] and

Dekker [19]. If neither underflow nor overflow occurs, `CGAL_Expanded_double`s compute exact results under addition, subtraction, and multiplication operations The initial values need not be integral.

`Ed. predicates` uses the Cartesian CGAL kernel with specialized predicates using adaptive evaluation. The specialized predicates use Shewchuk's code [43] for orientation predicates and other predicates in the same style. If the filter steps fail, `CGAL_Expanded_double` is finally used.

`C<leda_integer>` uses the Cartesian CGAL kernel with the arbitrary precision integer type `leda_integer`[3] from LEDA.

`C<CGAL_Gmpz>` uses the Cartesian CGAL kernel with the arbitrary precision integer type `CGAL_Gmpz` which is just a wrapper for the Gnu multiple precision arithmetic `gmp` [26]. The wrapper layer uses reference counting.

`C<leda_bigfloat> 53` uses the Cartesian CGAL kernel with the floating-point number type `leda_bigfloat`. Among other modes this floating-point number type allows one to fix the mantissa length (in bits). Here, the mantissa length was set to 53, corresponding to computation with `double`.

`C<Q< double> >` uses the Cartesian CGAL kernel with number type `CGAL_Quotient< double>`. The type `CGAL_Quotient` is a parameterized and slightly simplified version of the number type `leda_rational`. A number is maintained as a numerator and a denominator.

`C<Q<leda_integer> >` uses CGAL's Cartesian kernel with number type `CGAL_Quotient< leda_integer>`. This number type corresponds to `leda_rational`.

`H<double>` uses the homogeneous kernel of CGAL with the built-in double precision floating-point number type.

`H<leda_real>` uses the homogeneous CGAL kernel with number type `leda_real`.

`H<leda_integer>` uses the homogeneous CGAL kernel with number type `leda_integer`.

`S<float>` uses Cartesian representation without reference counting. Coordinates are of type `float`, predicates are analogous to the Cartesian CGAL kernel.

`S<double>` uses Cartesian representation without reference counting. Coordinates are of type `double`.

`S<doubledouble>` uses Cartesian coordinates of type `doubledouble`. Objects are not reference counted.

`S<leda_real>` uses Cartesian representation without reference counting. The coordinates are of type `leda_real`.

`V<float>` uses a Java-like geometry kernel: Classes have abstract base classes specifying their interfaces, most of the member functions are declared virtual and not declared inline. Cartesian representation is used with coordinates of type `float`.

`V<double>` as above, but with coordinates of type `double`.

`V<leda_real>` As above, with coordinates of type `leda_real`.

`Cw<float>` uses the Cartesian CGAL kernel with an additional wrapper layer that gives Cartesian and homogeneous kernels a common interface. The Cartesian coordinates are stored as single precision floats. It uses point type `CGAL_Point_2< CGAL_Cartesian< float> >`, which wraps `CGAL_PointC2<float>`. Hence it uses reference counting.

`Cw<double> ++` uses the Cartesian CGAL kernel with CGAL's additional wrapper layer. It uses double precision floating point arithmetic and the corresponding predicates, but does a few additional test to increase robustness. For example, in the comparison predicate for counterclockwise rotation order, points to be compared are checked for equality with the rotation center before an orientation predicate is called.

[3] Note that `leda_integer` uses assembler code for the test machine (just like `CGAL_Gmpz`), while they don't do so on other machines.

3 Selected Performance Issues

In this section we give examples on the typical performance with different geometry kernels. In order to verify that the parameterization of our code does not cause inherent overhead and to show the competitiveness of the generic implementations we run our experiments on the following convex hull code as well:

`Clarkson's` Clarkson's nice planar convex hull code that fits on the back of a business card [18]. Modified to output points instead of indices. Uses `qsort()` internally. Reads points from an array and outputs point to an array.

`leda CONVEX_HULL (IC)` LEDA's standard convex hull algorithm for its floating-point geometry kernel. Uses incremental construction. Operates on `leda_lists`.

`leda CONVEX_HULL_S` LEDA's convex hull sweep algorithm for its floating-point geometry kernel. Sorts the points and adds them in sorted order as described in [29], see also [22]. Operates on `leda_lists`.

`leda CONVEX_HULL_OLD` convex hull algorithm for LEDA's floating-point geometry kernel. It has a *throw-away* preprocessing step similar to the Akl-Toussaint algorithm. After the preprocessing step, points are added incrementally as in LEDA's new standard convex hull algorithm. Operates on `leda_lists`.

Figures 2 - 5 show the performance of these algorithms and the parameterized algorithms with geometry kernel `C<double>` for a sets of 1000 points in a square, in a disk, a biased disk, and almost on a circle, respectively. We find that bar charts illustrate the typical behavior much better than lots of columns of running times, since we are interested in the ratio of the running times for the tested combinations rather than precise numerical values. After all, there are always small errors in measurement. With each bar you see an abbreviation for the algorithm used as well as the name of the geometry kernel. The horizontal extension of a bar is proportional to the running time, which is shown on the left. The slowest algorithm has a bar of maximal size. Further comparisons of running times with respect to the algorithms are given in figures 18 and 19.

3.1 Cost of (Exact) Arithmetic

We start with a representative sample of traits classes with different arithmetic plugged into algorithm `Handley`. Fig. 6 shows a bar chart of the running times for a set of 1000 points in a disk. We did not add a kernel using `leda_bigfloat` with precision fixed to 53 to the sample in Fig. 6, because it was so slow, a factor of 10 slower than `leda_integer`.

Arbitrary precision integer arithmetic is well known to be expensive. Not surprisingly we made the same observation. Karasick et al. [31] already report on slow down factors of several orders of magnitude. Of course, the factor depends on the arithmetic demand of the algorithm (which is low in all algorithms of the test set) and the efficiency of the arbitrary precision integer arithmetic. In Fig. 7 we compared two arbitrary precision integer types: `leda_integer` and the CGAL number type `CGAL_Gmpz`, which is a reference counted number type wrapping the Gnu multiple precision integer package `gmp` [26]. The wrapping

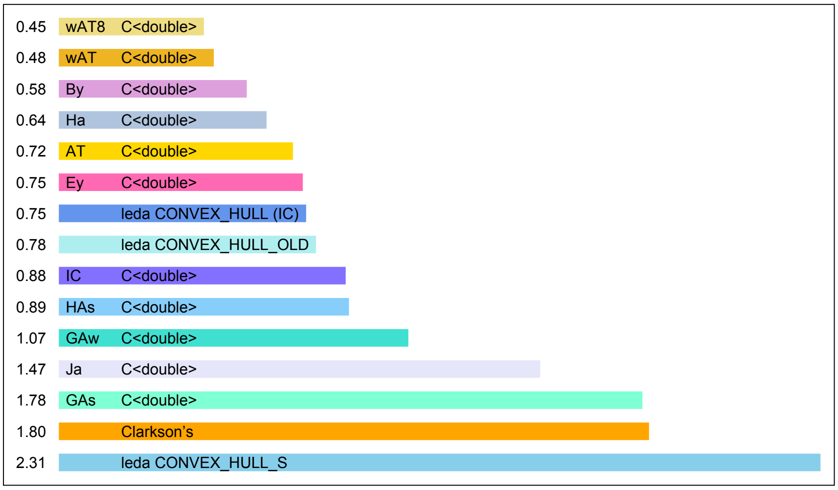


Fig. 2. Running times of the different algorithms with double precision floating point arithmetic. Points were uniformly distributed in a square and had integral coordinates of at most 20 bits. Running times are for 500 iterations.

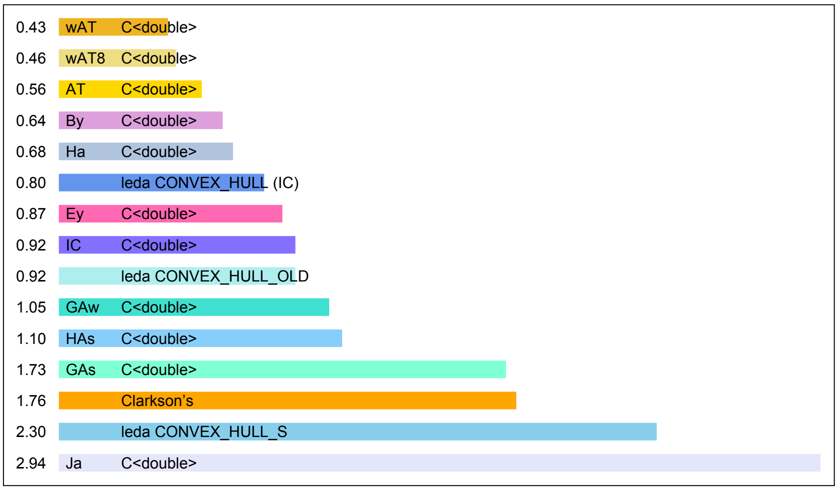


Fig. 3. Running times of the different algorithms with double precision floating point arithmetic. Points were uniformly distributed in a disk and had integral coordinates of at most 20 bits. Running times are for 500 iterations.

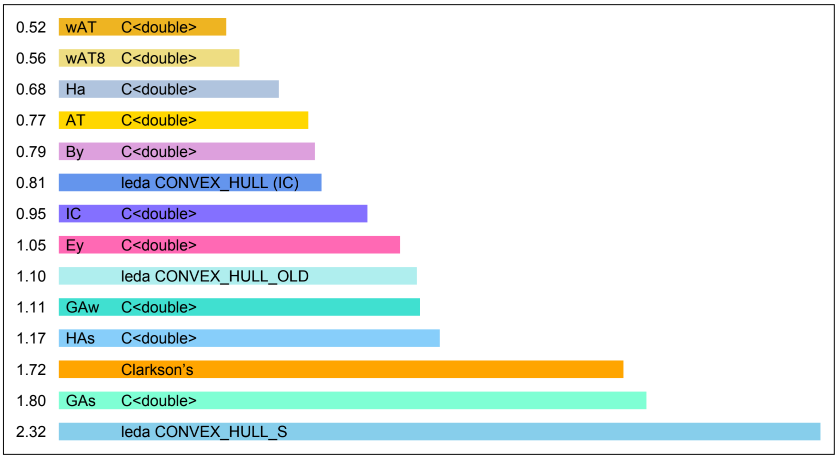


Fig. 4. Running times of the different algorithms with double precision floating point arithmetic. Points had integral coordinates of at most 20 bits and are in a biased disk (with increased probability near the border). Running times are for 500 iterations.

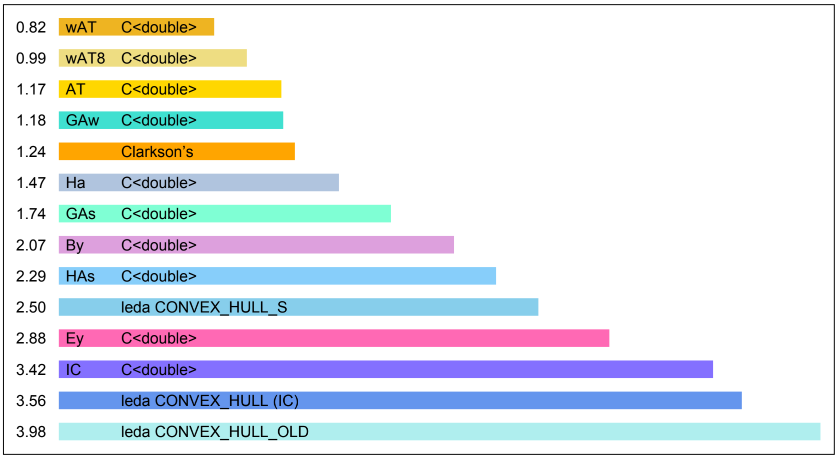


Fig. 5. Running times of the different algorithms with double precision floating point arithmetic. Points had integral coordinates of at most 20 bits and are (almost) on a circle. Running times are for 500 iterations.

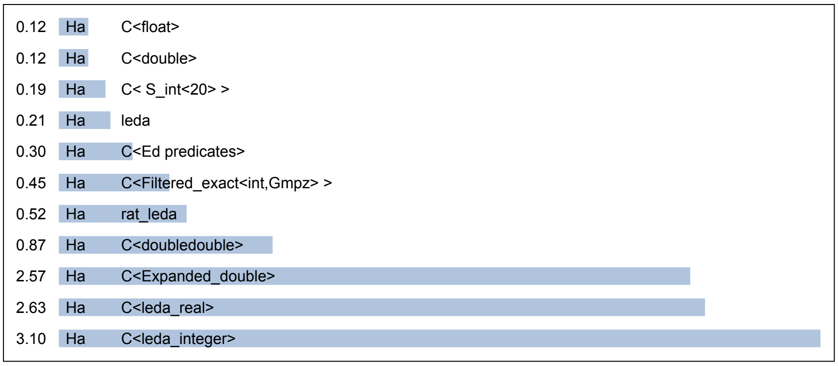


Fig. 6. Running times (for 100 iterations) for `Handley` (`Ha`) on non-uniformly distributed points (with integral coordinates of at most 20 bits) in a disk.

gives the `gmp`, which is written in C, the natural syntax for usage within C++. Fig. 7 shows typical ratio of running times for the two integer types with the Cartesian CGAL kernel. Note that all the numbers arising during computation were very small. Thus, one cannot conclude anything on the performance of these types on larger integers. As can be seen in Fig. 6, the kernels `C<doubledouble>` and `C<Expanded_double>` faster than the arbitrary precision integer types.

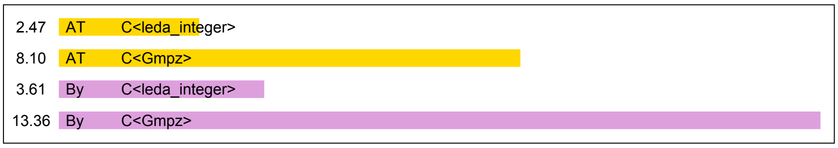


Fig. 7. Performance of `Akl-Toussaint` (`AT`) and `Bykat` (`By`) with different integer types.

The goal of adaptive evaluation is to avoid computation with higher precision if it is not needed for reliable decisions. The simplest form are static floating-point filters. Based on a bound on the size of the (integral) operands error bounds for the values computed in the predicates are computed a priori. A sign test simply compares the absolute computed value to the error bound. Only if the error bound is larger, the actual value of an expression and the computed approximation might have different signs. If the error bound can not verify the sign, the expression is re-evaluated using exact arithmetic. Such a filter is used in `C<CGAL_S_int<N>`, where `N` is a bound on the number of bits of the integral operands in an expression. The quality of such a filter depends on how good the a priori known size bound `N` matches the actual sizes of the operands. Figures 8 and 9 illustrate the dependence on the number of bits in the input coordinates. Static filters fail more often and cause exact re-calculation, if the size bound is not tight.

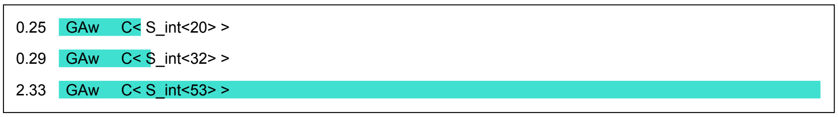


Fig. 8. `C<CGAL_S_int<20> >` vs. `C<CGAL_S_int<32> >` and `C<CGAL_S_int<53> >` on input data with coordinates of at most 20 bits.

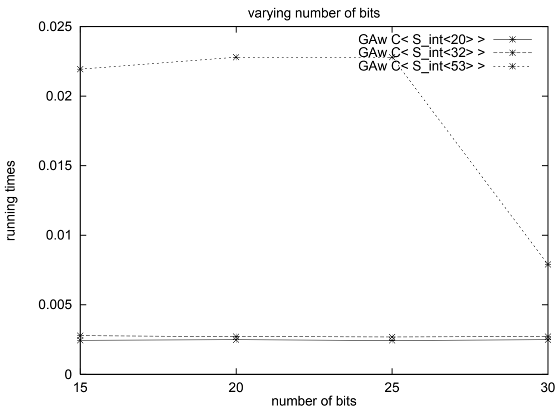


Fig. 9. Dependence on the number of bits. Note that `C<CGAL_S_int<20> >` does not guarantee exact decisions for coordinates with more than 20 bits.

With a semi-dynamic filter, bounds on the sizes of the operands are computed at run time. Some factors in the error bound, that depend on the expression only, are still computed a priori. Such a filter is used in `rat_leda`, the rational geometry kernel of LEDA. Semi-dynamic filters give better error bounds.

A fully-dynamic filter is used in the `leda_reals`. The error bound is completely determined at run time. `leda_real` provide a very general and easily applicable way of exact computation. There is no need to derive error bounds while implementing the predicates. The convenience has its price, but the adaptive evaluation keeps the cost in a reasonable range in our case.

Interval arithmetic [4,40,11] can also be used as a fully dynamic filter: First, all computations in a predicate are done with interval arithmetic. Only if the

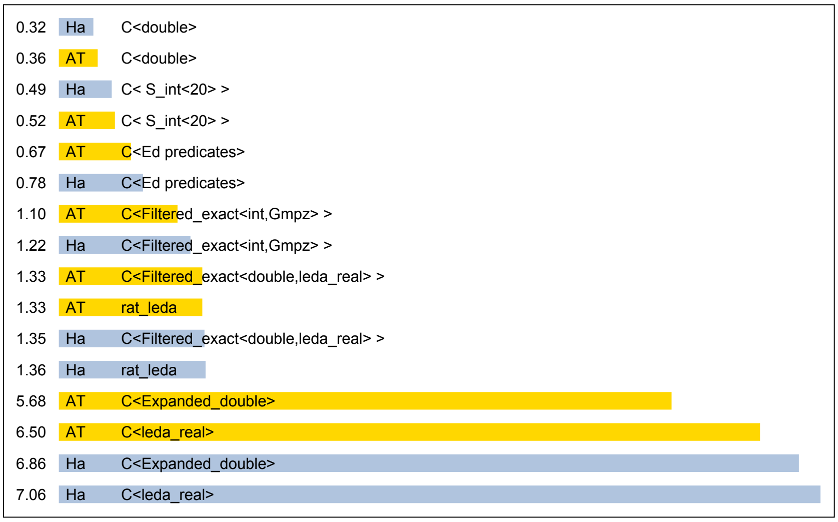


Fig. 10. Running times for different kernels supporting exact computation and for `C<double>` for 1000 points in a disk with algorithms `Handley` (`Ha`) and Akl-Toussaint (`AT`).

computed intervals do not allow for reliable comparison (for instance, if the interval enclosing a numerical value, whose sign has to be determined, contains zero), the computation is re-done in a second step using exact computation. In CGAL, this technique is used in the predicates for number type `CGAL_Filtered_exact< NumberType1, NumberType2 >`. The number type `NumberType1` is used to store the coordinates. In a predicate, the numerical data are first converted into intervals with endpoints of type `double`. Then the predicate is evaluated with interval arithmetic. If during this evaluation, a comparison operation cannot make a reliable decision because the intervals to be compared do overlap, an exception[4] is thrown which is then caught by a code block that re-does the computation. In this latter step the data are converted to number type `NumberType2` and the predicate is now evaluated with the arithmetic of this type.

Shewchuk [43] provides exact predicates for points with Cartesian double coordinates (not necessarily integral). Before a filter he uses further tests to speed up reliable decision making with floating-point arithmetic. As one can see in Fig. 10, the tests are quite effective.

The above geometry kernels supporting exact computation have quite different capabilities. For `CGAL_S_int< >` one needs a tight a priori known bound on the size of the integral input data. No such bound is needed for using `CGAL_Filtered_exact<int,CGAL_Gmpz>`. The kernel with `CGAL_Filtered_exact <double,leda_real>` and the kernel based on Shewchuk's code can handle non-

[4] using the exception handling mechanism of C++

integral data as well. The `rat_leda` kernel supports rational arithmetic and can even be used in cascaded computations. Besides rational arithmetic, `leda_reals` support root operations, too. Using `leda_real` in CGAL's kernels gives an easy to use and general way of exact geometric computation, but it is also more expensive than the more specialized approaches.

3.2 Arithmetic Demand and Number Types

It can be seen in Fig. 11, that `Bykat` is faster than `Graham-Andrew` for the built-in floating-point types while `Bykat` is slower for the expensive number types. The reason is the higher arithmetic demand [9,34] of the `Bykat` algorithm. In the worst-case, the demand is the same for both algorithms, both have degree 2. While all predicates in `Bykat` have degree 2, the comparisons in the sorting step of `Graham-Andrew` have degree 1 only. This pays off for expensive number types. For such reasons, the ranking of the algorithms with slower arithmetic can be quite different from the ranking with fast arithmetic.

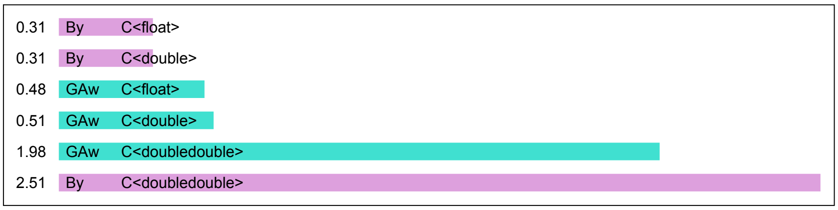


Fig. 11. Performance of `Bykat` and `Graham-Andrew` on different number types. Which one is faster depends on the cost of the arithmetic.

3.3 Coordinate Representation

In CGAL, a user can choose between Cartesian and homogeneous coordinates. Homogeneous representation has an additional coordinate that can be seen as a common denominator for Cartesian coordinates. A point p with homogeneous coordinates $(p_{\mathrm{hx}}, p_{\mathrm{hy}}, p_{\mathrm{hw}})$ with $p_{\mathrm{hw}} \neq 0$ has Cartesian coordinates $(p_{\mathrm{hx}}/p_{\mathrm{hw}}, p_{\mathrm{hy}}/p_{\mathrm{hw}})$. In homogeneous geometry kernels division can be avoided. In figures 12 and 13 we compare kernels with homogeneous and Cartesian coordinates of the same number type. Furthermore, we compare them to a Cartesian kernel, whose coordinates are quotients of this number type. Not surprisingly, the experiments show that the homogeneous kernels are slower than the Cartesian kernels with the same number type. If we use a parameterized quotient type with the same number type in the Cartesian kernel, it turns out that it is slower than the corresponding homogeneous kernel, as expected.

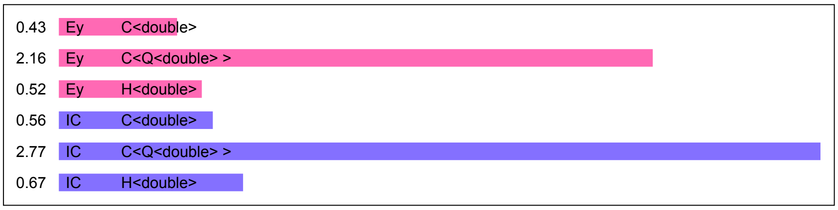


Fig. 12. Performance of homogeneous and Cartesian kernels with number type `double`.

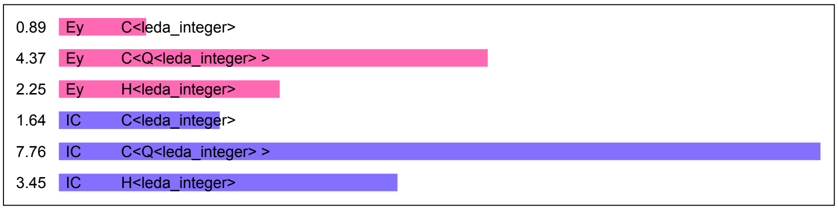


Fig. 13. Performance of homogeneous and Cartesian kernels with number type `leda_integer`.

3.4 Influence of Reference Counting

To investigate the cost of reference counting we used a second kernel template with Cartesian representation. Use of reference counting slows down coordinate access, since objects refer to the shared representation via a pointer. Copying redirects a pointer and updates reference counters. Depending on the size of the objects, the number of copying operations performed, and the number of coordinate access operations, this strategy may or may not pay off. The `C< >` kernels use reference counted points, the `S< >` kernels always copy their coordinates. In figures 14 and 15 we used algorithms `Bykat` and `IC_ala_LEDA`. Furthermore, we used an algorithm that copies the points, sorts them lexicographically, and report as many points as there are hull points. We call it `copy and sort` (`c&s`). The figures show, that the reference counting doesn't help for small number types. If a number type needs more space, e.g. a `leda_real` has twice the size of a `double`, the reference counting leads to slightly better performance. It has to be added, that the behavior under reference counting and the answer to the question when is it worth to use it also depend on caching and memory effects.

3.5 Library Design and Style Issues

The generic-programming style with templates supports inlining which in turn allows the compiler to a better job in optimizing the code. We compared the CGAL kernels (with one more level of inlining) with a different more Java-like

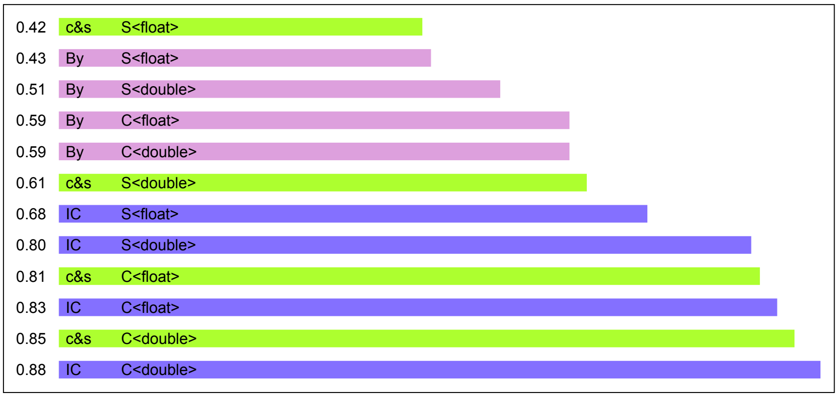


Fig. 14. Reference counted (`C<>`) versus non-reference counted (`S<>`) points for small size number types (1000 random points in a square, 500 iterations).

design, where we made all access member functions virtual and did not inline them. This slows down the algorithms significantly, even more than adaptive exact computation does, see Fig. 16.

The additional wrapper layer for unifying the interface of Cartesian and homogeneous kernel in CGAL does not cause any slow down. It is optimized away by the compiler. Both `C<float>` and `Cw<float>` showed the same performance in our experiments. Because of the additional tests for increasing robustness `Cw<double> ++` was slightly slower than `C<double>`.

Finally, we compared running times for the LEDA floating point kernel `leda` and `C<double>` in Fig. 17. Most of the predicates in both kernels are very similar. However, some of the predicates in `leda` are not inlined, but calls of library functions. The crucial predicates in `Eddy` are inlined, and hence the difference in running times is less significant for this algorithm.

4 Conclusions and Future Work

Most of the experiments we did confirmed our expectation. The most notable observation might be that adaptive exact computation in a design that supports optimization by the compiler very well is less expensive than a Java-style design that hinders optimization.

In the future we will add experiments with cascaded computation, e.g. computing the convex hull of the intersection points of a set of line segments. For such computations Cartesian kernels with integral number types will not suffice anymore. We will also further complete the algorithmic component by adding a divide & conquer algorithm to the set of test algorithms, the optimal *marriage-before-conquest* algorithm [32], which is, however, unlikely to be competitive for small size problems, see [36], and Chan's algorithm [17]. A first experiments with

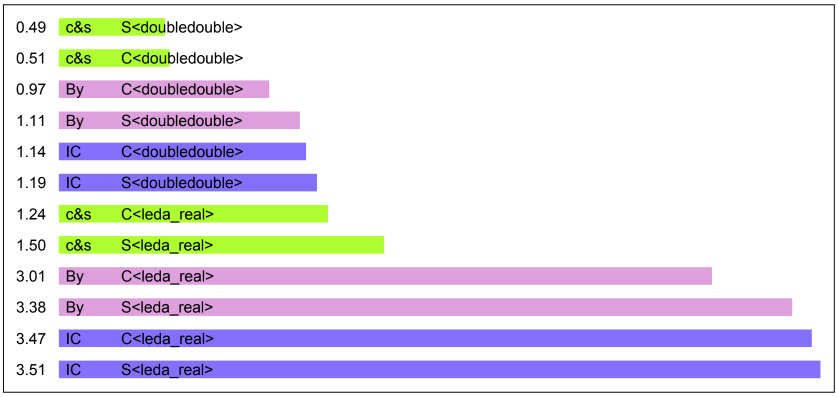


Fig. 15. Reference counted (`C<>`) versus non-reference counted (`S<>`) points for larger number types (1000 random points in a square, 100 iterations).

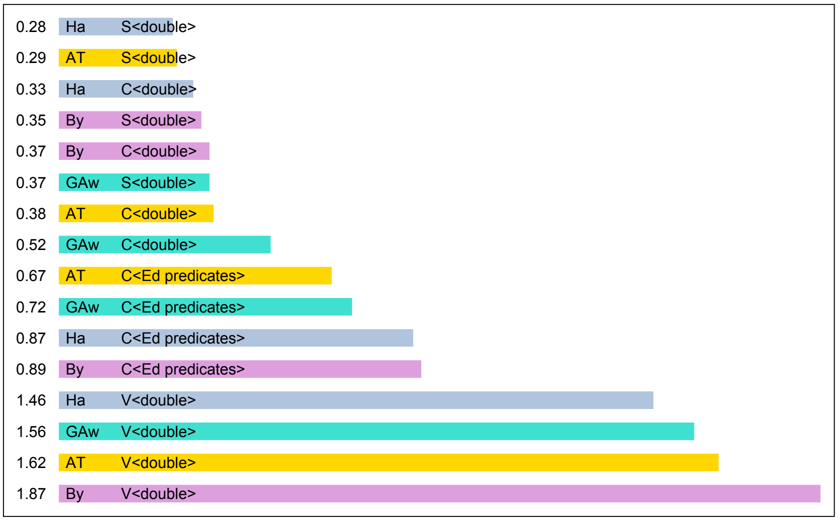


Fig. 16. Cartesian CGAL kernel with inlining and non-virtual member functions (`C<>`) versus a kernel with non-inlined virtual member functions (`V<>`). Moreover, the Cartesian kernel with `Ed. predicates` was run on the same point set.

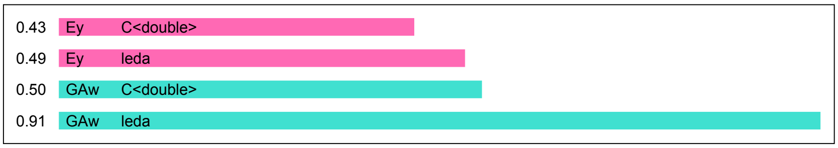


Fig. 17. Running times (for 250 iterations) for `Eddy` (`Ey`) and `Graham-Andrew` (`GAw`) on uniformly distributed points in a disk.

an implementation of Chan's algorithm, it was not competitive for the problem size we looked at. We also might add variations on quickhull, where new extreme points are computed such that subproblem sizes are balanced. This would give further worst-case optimal algorithms.

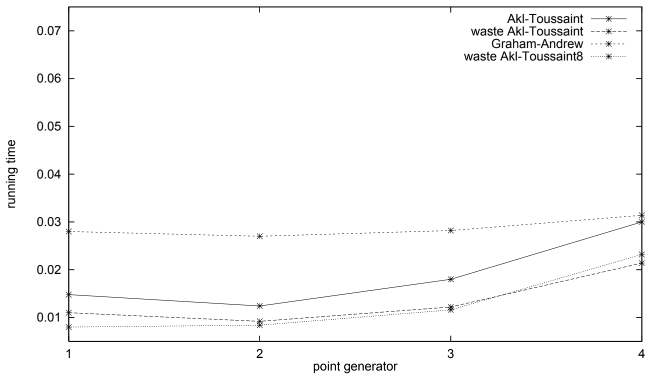


Fig. 18. Dependence on point distribution and hull size. Running times at x-coordinate k are for 10^4 points created by generator (k).

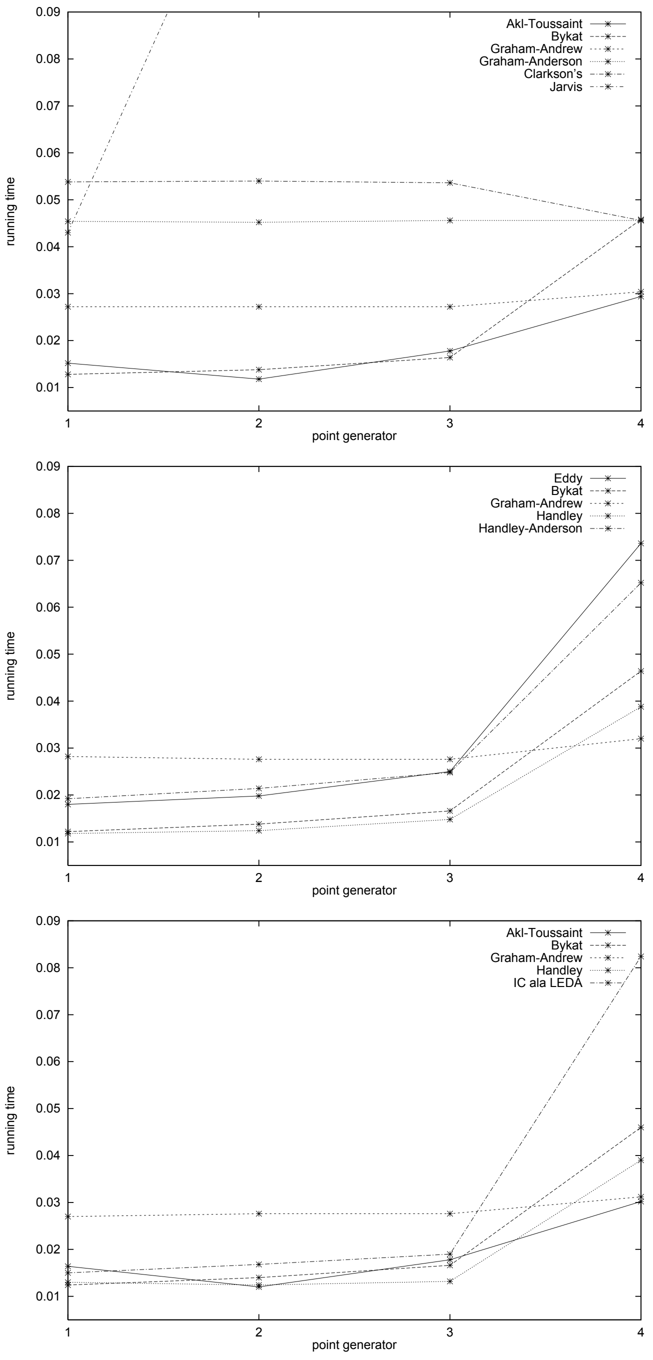


Fig. 19. Dependence on point distribution and hull size. Running times at x-coordinate k are for 10^4 points created by generator (k).

References

1. S. G. Akl. Two remarks on a convex hull algorithm. *Inform. Process. Lett.*, 8(2):108–109, 1979.
2. S. G. Akl. Corrigendum on convex hull algorithms. *Inform. Process. Lett.*, 10(3):168, 1980.
3. S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Inform. Process. Lett.*, 7(5):219–222, 1978.
4. G. Alefeld and J. Herzberger. *Introduction to Interval Computation.* Academic Press, New York, 1983.
5. D. C. S. Allison and M. T. Noga. Some performance tests of convex hull algorithms. *BIT*, 24:2–13, 1984.
6. K. R. Anderson. A reevaluation of an efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 7(1):53–55, 1978.
7. A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9(5):216–219, 1979.
8. B. K. Bhattacharya and G. T. Toussaint. Time- and storage-efficient implementation of an optimal convex hull algorithm. *Image Vision Comput.*, 1:140–144, 1983.
9. J.-D. Boissonnat and F. Preparata. Robust plane sweep for intersecting segments. Technical Report 3270, INRIA, Sophia-Antipolis, France, September 1997.
10. K. Briggs. The doubledouble home page. `http://epidem13.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`.
11. H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.
12. C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proc. of the 8th ACM-SIAM Symp. on Discrete Algorithms*, pages 702–709, 1997.
13. C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proceedings of the 11th ACM Symposium on Computational Geometry*, pages C18–C19, 1995.
14. C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class `real` number. Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, 1996.
15. A. Bykat. Convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 7:296–298, 1978.
16. CGAL project. `http://www.cs.uu.nl/CGAL`
17. T. M. Y. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 10–19, 1995.
18. K. Clarkson. A short, complete planar convex hull code. `http://cm.bell-labs.com/who/clarkson/2dch.c`.
19. T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224 – 242, 1971.
20. F. Dévai and T. Szendrényi. Comments on convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 9:141–142, 1979.
21. W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3:398–403 and 411–412, 1977.
22. H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer Verlag, 1986.
23. S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.
24. A. Fournier. Comments on convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 8:173, 1979.

25. R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.
26. T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.
27. P. J. Green and B. W. Silverman. Constructing the convex hull of a set of points in the plane. *Comput. J.*, 22:262–266, 1979.
28. C. C. Handley. Efficient planar convex hull algorithm. *Image Vision Comput.*, 3:29–35, 1985.
29. S. Hertel. An almost trivial convex hull algorithm for a presorted point set in the plane. Report A83/08, Fachber. Inform., Univ. Saarlandes, Saarbrücken, West Germany, 1983.
30. R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2:18–21, 1973.
31. M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.
32. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.
33. J. Koplowitz and D. Jouppi. A more efficient convex hull algorithm. *Inform. Process. Lett.*, 7:56–57, 1978.
34. G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.
35. S. B. Lippman and J. Lajoie. *C++ primer*. Addison-Wesley, Reading, MA, 3rd ed., 1998.
36. M. M. McQueen and G. T. Toussaint. On the ultimate convex hull algorithm in practice. *Pattern Recogn. Lett.*, 3:29–34, 1985.
37. K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, Heidelberg, Germany, 1984.
38. K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.
39. K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User manual*, 3.7 edition, 1998. see `http://www.mpi-sb.mpg.de/LEDA/leda.html`.
40. R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, 1979.
41. M. H. Overmars and J. van Leeuwen. Further comments on Bykat's convex hull algorithm. *Inform. Process. Lett.*, 10:209–212, 1980.
42. D. M. Priest. *On Properties of Floating-Point Arithmetic: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, Department of Mathematics, University of California at Berkeley, 1992.
43. J. R. Shewchuk. C code for the 2d and 3d orientation and incircle tests, and for arbitrary precision floating-point addition and multiplication. Available from `http://www.cs.cmu.edu/~quake/robust.html`.
44. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, 1996.
45. G. T. Toussaint. A historical note on convex hull finding algorithms. *Pattern Recogn. Lett.*, 3:21–28, 1985.

Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning[⋆]

Andrew E. Caldwell, Andrew B. Kahng and Igor L. Markov

UCLA Computer Science Dept., Los Angeles, CA 90095-1596

Abstract. We discuss the implementation and evaluation of move-based hypergraph partitioning heuristics in the context of VLSI design applications. Our first contribution is a detailed software architecture, consisting of seven reusable components, that allows flexible, efficient and accurate assessment of the practical implications of new move-based algorithms and partitioning formulations. Our second contribution is an assessment of the modern context for hypergraph partitioning research for VLSI design applications. In particular, we discuss the current level of sophistication in implementation know-how and experimental evaluation, and we note how requirements for real-world partitioners – if used as motivation for research – should affect the evaluation of prospective contributions. We then use two "implicit decisions" in the implementation of the Fiduccia-Mattheyses [20] heuristic to illustrate the difficulty of achieving meaningful experimental evaluation of new algorithmic ideas.

1 Introduction: Hypergraph Partitioning in VLSI Design

Given a hyperedge- and vertex-weighted hypergraph $H = (V, E)$, a *k-way partitioning* of V assigns the vertices to k disjoint nonempty partitions. The *k-way partitioning problem* seeks to minimize a given cost function $c(P^k)$ whose arguments are partitionings. A standard cost function is *net cut*,[1] which is the sum of weights of hyperedges that are cut by the partitioning (a hyperedge is *cut* exactly when not all of its vertices are in one partition).

Constraints are typically imposed on the partitioning solution, and make the problem difficult. For example, certain vertices can be fixed in particular partitions (*fixed constraints*). Or, the total vertex weight in each partition may be limited (*balance constraints*), which results in an NP-hard formulation [21]. Thus, the cost function $c(P^k)$ is minimized over the set of *feasible solutions* S_f, which is a subset of the set of all possible k-way partitionings. Effective move-based heuristics for k-way hypergraph partitioning have been pioneered in such works as [36], [20], [9], with refinements given by [38], [42], [26], [40], [18], [4], [12], [25], [34], [19] and many others. A comprehensive survey of partitioning formulations and algorithms, centered on VLSI applications and covering move-based,

[⋆] This research was supported by a grant from Cadence Design Systems, Inc. Author contact e-mail: {caldwell,abk,imarkov}@cs.ucla.edu.

[1] Or simply *cut*, as in *minimum cut partitioning*. Note that in the VLSI context, a circuit hypergraph is called a *netlist*; a hyperedge corresponds to a *signal net*, or *net*; and a vertex corresponds to a *module*.

spectral, flow-based, mathematical programming-based, etc. approaches, is given in [5]. A recent update on balanced partitioning in VLSI physical design is provided by [31].

1.1 The VLSI Design Context

VLSI design has long provided driving applications and ideas for hypergraph partitioning heuristics. For example, the methods of Kernighan-Lin [36] and Fiduccia-Mattheyses [20] form the basis of today's move-based approaches. The method of Goldberg-Burstein [24] presaged the multilevel approaches recently popularized in the parallel simulation [22], [27], [32] and VLSI [3],[34],[4] communities. As noted in [5], applications in VLSI design include test; simulation and emulation; design of systems with multiple field-programmable devices; technology migration and repackaging; and top-down floorplanning and placement.

Depending on the specific VLSI design application, a partitioning instance may have directed or undirected hyperedges, weighted or unweighted vertices, etc. However, in all contexts the instance represents – at the transistor-level, gate-level, cell-level, block-level, chip-level, or behavioral description module level – a human-designed system. Such instances are highly non-random. Many efforts (e.g., [30], [14], [23], [8]) have used statistical attributes of real-world circuit hypergraphs (based on Rent's parameter [39], shape, depth, fanout distribution, etc.) to generate random hypergraphs believed relevant to evaluation of heuristics. These efforts have not yet met with wide acceptance in the VLSI community, mostly because generated instances do not guarantee "realism". Hence, the current practice remains to evaluate new algorithmic ideas against suites of benchmark instances.

In the VLSI partitioning community, performance of algorithms is typically evaluated on the ACM/SIGDA benchmarks now maintained by the Collaborative Benchmarking Laboratory at North Carolina State University `http:www.cbl.ncsu.edu/benchmarks`.[2] Alpert [2] noted that many of these circuits no longer reflect the complexity of modern partitioning instances, particularly in VLSI physical design; this motivated the release of eighteen larger benchmarks produced from internal designs at IBM [1].[3]

Salient features of benchmark (real-world) circuit hypergraphs include

- size: number of vertices can be up to one million or more (instances of all sizes are equally important).
- sparsity: average vertex degrees are typically between 3 and 5 for cell-, gate- and device-level instances; higher average vertex degrees occur in block-level design.
- number of hyperedges (nets) typically between 0.8x and 1.5x of the number of vertices (each module typically has only one or two outputs, each of which represents the source of a new signal net).
- average net sizes are typically between 3 to 5.
- a small number of very large nets (e.g., clock, reset, test) connect hundreds or thousands of vertices.

[2] These benchmarks are typically released by industry or academic designers at various workshops and conferences (e.g., LayoutSynth90, LayoutSynth92, Partitioning93, PDWorkshop93, ...).

[3] While those benchmarks are now used in most partitioning papers, we would like to stress that they present considerably harder partitioning problems than earlier available benchmarks available from `http://www.cbl.ncsu.edu/benchmarks`, primarily due to more esoteric distributions of node degrees and weights. See, e.g., Tables 6, 4 and 5.

Statistics for circuit benchmarks used in our work are given in Tables 4, 5 and 6.

Partitioning heuristics must be highly efficient in order to be useful in VLSI design.[4] As a result – and also because of their flexibility in addressing variant objective functions – fast and high-quality iterative move-based partitioners based on the approach of Fiduccia-Mattheyses [20] have dominated recent practice.

1.2 The Fiduccia-Mattheyses Approach

The Fiduccia-Mattheyses (FM) heuristic for bipartitioning circuit hypergraphs [20] is an iterative improvement algorithm. Its neighborhood structure is induced by single-vertex, partition-to-partition moves.[5] FM starts with a possibly random solution and changes the solution by a sequence of moves which are organized as *passes*. At the beginning of a pass, all vertices are free to move (*unlocked*), and each possible move is labeled with the immediate change in total cost it would cause; this is called the *gain* of the move (positive gains reduce solution cost, while negative gains increase it). Iteratively, a move with highest gain is selected and executed, and the moving vertex is *locked*, i.e., is not allowed to move again during that pass. Since moving a vertex can change gains of adjacent vertices, after a move is executed all affected gains are updated. Selection and execution of a best-gain move, followed by gain update, are repeated until every vertex is locked. Then, the best solution seen during the pass is adopted as the starting solution of the next pass. The algorithm terminates when a pass fails to improve solution quality.

The FM algorithm can be easily seen to have three main operations: (1) the computation of initial gain values at the beginning of a pass; (2) the retrieval of the best-gain (feasible) move; and (3) the update of all affected gain values after a move is made. The contribution of Fiduccia and Mattheyses lies in observing that circuit hypergraphs are sparse, so that any move gain is bounded between two and negative two times the maximal vertex degree in the hypergraph (times the maximal edge weight, if edge weights are used). This allows hashing of moves by their gains: all affected gains can be updated in linear time, yielding overall linear complexity per pass. In [20], all moves with the same gain are stored in a linked list representing a "gain bucket".

1.3 Contributions of This Paper

In this paper, we discuss the implementation and evaluation of move-based hypergraph partitioning heuristics, notably the FM heuristic, in the context of VLSI design applications. Our first contribution is a detailed software architecture, consisting of seven reusable components, that allows flexible, efficient and accurate assessment of the practical implications of new move-based algorithms and partitioning formulations. Our second contribution is an assessment of the modern context for hypergraph partitioning research for

[4] For example, a modern top-down standard-cell placement tool might perform timing- and routing congestion-driven recursive min-cut bisection of a cell-level netlist to obtain a "coarse placement", which is then refined into a "detailed placement" by stochastic hill-climbing search. The *entire* placement process in currently released tools (from companies like Avant!, Cadence, CLK CAD, Gambit, etc.) takes approximately 1 CPU minute per 6000 cells on a 300MHz Sun Ultra-2 uniprocessor workstation with adequate RAM. The implied partitioning runtimes are on the order of 1 CPU second for netlists of size 25,000 cells, and 30 CPU seconds for netlists of size 750,000 cells [16]. Of course, we do not advocate performance tuning to match industrial-strength runtimes. However, absent other justifications, "experimental validation" of heuristics in the wrong runtime regimes (say, hundreds of CPU seconds for a 5000-cell benchmark) has no practical relevance.

[5] By contrast, the stronger Kernighan-Lin (KL) heuristic [36] uses a pair-swap neighborhood structure.

VLSI design applications. In particular, we discuss the current level of sophistication in implementation know-how and experimental evaluation, and we note how requirements for real-world partitioners – if used as motivation for research – should affect the evaluation of prospective contributions. We then use two "implicit decisions" in the implementation of the FM heuristic to illustrate the difficulty of achieving meaningful experimental evaluation of new algorithmic ideas. Finally, we provide brief anecdotal evidence that our proposed software architecture is conducive to algorithm innovation and leading-edge quality of results.

2 Architecture of a Move-Based Partitioning Testbench

In this section, we describe a seven-component software architecture for implementation of move-based partitioning heuristics, particularly those based on the FM approach. By way of example, we reword the Fiduccia-Mattheyses algorithm in terms of these seven software components. By carefully dividing responsibilities among components we attempt to provide the implementation flexibility and runtime efficiency that is needed to evaluate the practical impact of new algorithmic ideas and partitioning formulations.

2.1 Main Components

Common Partitioner Interface. Formally describes the input and output to partitioners without mentioning internal structure and implementation details. All partitioner implementations then conform to this input/output specification.

Initial Solution Generator. Generates partitionings that satisfy given constraints, typically using randomization in the construction.

Incremental Cost Evaluator. Evaluates the cost function for a given partitioning and dynamically maintains cost values when the partitioning is changed by applying moves. Updates typically should be performed in constant time.

Legality Checker. Verifies whether a partitioning satisfies a given constraint. The Legality Checker is used to determine the legality of a move. Multiple constraints may be handled with multiple legality checkers.

Gain Container. A general container for moves, optimized for efficient allocation, retrieval and queueing of available moves by their gains. Moves can be retrieved by, e.g., the index of the vertex being moved, and/or the source or destination partition. The Gain Container supports quick updates of the gain for a move, and fast retrieval of a move with the highest gain. The Gain Container is also independent of the incremental cost evaluator and legality checker; it is populated and otherwise managed by the Move Manager.

Move Manager. Responsible for choosing and applying one move at a time. It may rely on a Gain Container to choose the best move, or randomly generate moves. It can undo moves on request. If used in pass-based partitioners, it incrementally computes the change in gains due to a move, and updates the Gain Container.

The Move Manager maintains "status information", such as the current cost and how each partition is filled. It may be controlled by the caller via parameter updates before every move selection (e.g. a temperature parameter in simulated annealing).

Pass-Based Partitioner (proper). Solves "partitioning problems" by applying incrementally improving passes to initial solutions. A pass consists of legal moves, chosen and applied by the move manager. Within a pass, a partitioner can request that the Move Manager *undo* some of the moves, i.e. perform inverse moves. The Pass-Based Partitioner is an implementation of the *Common Partitioning Interface*.

This modularity allows for separate benchmarking and optimization of most components. It also provides flexibility to use multiple alternative implementations relevant to special cases.[6] A fundamental facility enabling such modularity is a common efficient hypergraph implementation.[7]

2.2 Component Descriptions

We now give somewhat more detailed component descriptions, omitting three components for which implementation choices are less critical.

Incremental Cost Evaluator Initialized with a hypergraph, the Incremental Cost Evaluator is responsible for evaluating the cost function for a given partitioning, and incrementally maintaining this value when the partitioning changes (i.e., a vertex is moved). When the cost function is computed as sum of hyperedge costs, those costs should also be maintained and available.

Efficient implementations typically maintain an internal state, e.g. relevant statistics, for each hyperedge. This facilitates efficient constant-time cost updates when single moves are performed. An Evaluator whose values are guaranteed to be from a small (esp. finite) range should be able to exploit this range to enable faster implementations of the Gain Container (e.g. buckets *versus* priority queues).

Interface:

- Initialize (internal structures) with a hypergraph and a partitioning solution.
- Report current cost (total or of one net) without changing internal state.
- Complete change of internal state (re-initialization) for all vertices and nets.
- Incremental change of internal state (for all nets whose cost is affected or for a given net) due to one elementary move without updating the costs.[8]

[6] For example, many optimizations for 2-way partitioning from the general k-way case can be encapsulated in the evaluator. On the other hand, in our experience optimizing the Gain Container for 2-way is barely worth maintaining separate pieces of code.

[7] A generic hypergraph implementation must support I/O, statistics, various traversals and optimization algorithms. However, no such implementation will be optimal for all conceivable uses.

In particular, the excellent LEDA library is bound to have certain inefficiencies related to hypergraph construction and memory management. We decided to implement our own reusable components based on the Standard Template Library and optimize them for our use models.

Features directly supported by the hypergraph component include memory management options, conversions, I/O, various construction options such as ignoring hyperedges of size less than 2 or bigger than a certain threshold, lazily executed calls for sorting nodes or edges in various orders etc. Many trade-off had to be made, e.g. the hypergraph objects used in critical pieces of code have to be unchangeable after their initial construction so as to allow for very efficient internal data structures.

None of the many existing generic implementations we reviewed was sufficiently malleable to meet our requirements without overwhelming their source code by numerous compiler `#defines` for adapting the code to a given use model. Having complete control over the source code and internal interfaces also allows for maximal code reuse in implementing related functionalities.

[8] Changing the state of one net will, in general, make the overall state of the evaluator inconsistent. This can be useful, however, for "what-if" cost lookups when a chain of incremental changes can return to the original state.

Gain Container The Gain Container stores all moves currently available to the partitioner (these may not all be legal at a given time) and prioritizes them by their gains (i.e., the immediate effect each would have on the total cost). A container of move/gain pairs is defined by the interface below, which allows quick updates to each and any move/gain pair after a single move.

A Gain Container should be able to find the move with highest gain quickly, possibly subject to various constraints such as a given source or destination partition, and may provide support for various *tie-breaking schemes* in order to choose the best move among the moves with highest gain. A Gain Container does not initiate gain updates by itself and is not aware of Cost Evaluators, the Move Manager, or how the gains are interpreted. Gain Containers do not need to determine the legality of moves. This makes them reusable for a range of constrained partitioning problems. Faster implementations (e.g. with buckets) may require that the maximal possible gain be known.
Interface:

– Add a move to the Container, given the gain.
– Get the gain for a move.
– Set the gain for a move (e.g. to update).
– Remove a move from the Container.
– Find a move of highest gain.
– Invalidate current highest gain move, in order to request the next highest gain move.[9] Typically applied if the current highest gain move appears illegal.
– Invalidate current highest gain bucket to access the next highest gain bucket.

The primary constituents of a Gain Container are a *repository* and *prioritizers*.

Repository for gain/move pairs handles allocation and deallocation of move/gain pairs, and supports fast gain lookups given a move.
Prioritizer finds a move with highest gain. In addition, may be able to choose choose best-gain moves among moves with certain properties, such as a particular destination or source partition. Updates gains and maintains them queued, in particular, is responsible for tie-breaking schemes.

We say that some moves stored in the repository are *prioritized* when they participate in the prioritizer's data structures. Not prioritizing the moves affecting a given vertex corresponds to "locking" the vertex, as it will never be chosen as the highest-gain move. The standard FM heuristic locks a given cell as soon as it is moved in a pass; however, variant approaches to locking have been proposed [15].

Move Manager A Move Manager handles the problem's move structure by choosing and applying the best move (typically, the best *legal* move), and incrementally updates the Gain Container that is used to choose the best move. The Move Manager reports relevant "status information" after each move, e.g. current cost and partition balance, which allows the caller to determine the best solution seen during the pass. In order to return to such best solution, the move manager must perform undo operations on request.

[9] Note that this does not remove the move from the Gain Container.

Interface:

– Choose one move (e.g., the best feasible) and apply it. Ensure all necessary updates (gain container, incremental evaluator).
– Return new "status info", e.g., total cost, partition balances, etc.
– Undo a given number of moves (each move applied must be logged to support this).

Pass-Based Partitioner (Proper) Recall that a Pass-Based Partitioner applies incrementally improving passes to initial solutions and returns best solutions seen during such passes.[10] A pass consists of moves, chosen and applied by move manager. After a pass, a Partitioner can request that the Move Manager perform undo operations to return to the best solution seen in that pass. A Partitioner decides when to stop a pass, and what intermediate solution within the pass to return to, on the basis of its control parameters and status information returned by the Move Manager after each move. The Partitioner can have access to multiple combinations of Incremental Cost Evaluators, Move Managers and Gain Containers, and can use them flexibly at different passes to solve a given partitioning problem. Note that a Partitioner is not necessarily aware of the move structure used: this is a responsibility of Move Managers.

Interface:

– Takes a "partitioning problem" and operational parameters on input.
– Returns all solutions produced, with the best solution marked.

2.3 A Generic Component-based FM Algorithm

A "partitioning problem" consists of

– hypergraph
– solution placeholders ("buffers") with or without initial solutions
– information representing relevant constraints, e.g., fixed assignments of vertices to partitions and maximum total vertex area in each partition
– additional information required to evaluate the cost function, e.g., geometry of partitions for wirelength-driven partitioning in the top-down placement context.

The Partitioner goes over relevant places in solution buffers and eventually writes good partitioning solutions into them. An existing solution may thus be improved, but if a place is empty, an initial solution generator will be called. A relevant Move Manager must be instantiated and initialized; this includes instantiation of the constituent Evaluator and Gain Container. The Partitioner then performs successive passes as long as the solution can be improved.

At each pass, the Partitioner repetitively requests the Move Manager to pick one [best] move and apply it, and processes information about the new solutions thus obtained. Since no vertex can be moved twice in a pass, no moves will be available beyond a certain point (*end of a pass*). Some best-gain moves may increase the solution cost, and typically the solution at the end of the pass is not as good as the best solutions seen during the pass. The Partitioner then requests that the Move Manager undo a given number of moves to yield a solution with best cost.

[10] While every pass as a whole must not worsen current solution, individual moves within a pass may do so.

While the reinitialization of the Move Manager at the beginning of each pass seems almost straightforward, picking and applying one move is subtle. For example, note that the Move Manager requests the best move from the gain container and can keep on requesting more moves until a move passes legality check(s). As the Move Manager applies the chosen move and locks the vertex, gains of adjacent vertices may need to be updated.

In performing "generic" gain update, the Move Manager walks all nets incident to the moving vertex and for each net computes gain updates (*delta gains*) for each of its vertices due to this net (these are combinations of the given net's cost under four distinct partition assignments for the moving and affected vertices; see Section 3.4). These partial gain updates are immediately applied through Gain Container calls, and moves of affected vertices may have their priority within the Gain Container changed. Even if the delta gain for a given move is zero, removing and inserting it into the gain container will typically change tie-breaking among moves with the same gain.

In most implementations the gain update is the main bottleneck, followed by the Gain Container construction. Numerous optimizations of generic algorithms exist for specific cost functions, netcut being particularly amenable to such optimizations.

3 Evaluating Prospective Advances in Partitioning

3.1 Formulations and Metrics for VLSI Partitioning

VLSI design presents many different flavors of hypergraph partitioning. Objective functions such as ratio-cut [45], scaled cost [11], absorption cut [44] sum of degrees, number of vertices on the cut line [28], etc. have been applied for purposes ranging from routability-driven clustering to multilevel annealing placement. In top-down coarse placement, partitioning involves fixed or "propagated" terminals [17, 43], tight partition balance constraints (and non-uniform vertex weights), and an estimated-wirelength objective (e.g., sum of half-perimeters of net bounding boxes). By contrast, for logic emulation the partitioning might have all terminals unfixed, loose balance constraints (with uniform vertex weights), and a pure min-cut objective. The partitioning can also be multi-way instead of 2-way [43, 42, 29], "multi-dimensional" (e.g., simultaneous balancing of power dissipation and module area among the partitions), timing-driven, etc. With this in mind, partitioners are best viewed as "engines" that plug into many different phases of VLSI design. Any prospective advance in partitioning technology should be evaluated in a range of contexts.

In recent VLSI CAD partitioning literature, comparisons to previous work are made using as wide a selection of benchmark instances as practically possible; using uniform vs. non-uniform vertex weights; and using tight vs. loose partition balance constraints (typically 49-51% and 45-55% constraints for bipartitioning).[11] Until recently, heuristics have typically been evaluated according to solution quality and runtime. Even though the quality-runtime tradeoff is unpredictable given widely varying problem sizes, constraints and hypergraph topologies, most papers report average and best solution quality obtained over some fixed number of independent runs (e.g., 20 or 100 runs). This reporting style can obscure the quality-runtime tradeoff, notably for small runtimes, and is a failing of the

[11] VLSI CAD researchers also routinely document whether large nets were thresholded, the details of hypergraph-to-graph conversions (e.g., when applying spectral methods), and other details necessary for others to reproduce the experiments. The reader is referred to [5, 2] for discussions of reporting methodology.

VLSI CAD community relative to the more mature metaheuristics/INFORMS communities.[12] Statistical analyses (e.g., significance tests) are not particularly popular yet, but are recognized as necessary to evaluate the significance of solution quality variation in diverse circumstances [8].

3.2 Need For "Canonical" Testbench

The components described in Section 2 yield a testbench, or "framework", that can be recombined and reused in many ways to enable experiments with

- multiple objective functions, e.g., ratio cut [45], absorption [44], the number of boundary vertices [28], the "$k-1$ objective" [13] etc.
- multiple constraint types ([35])
- variant formulations, e.g., multi-way [42,33,15], replication-based [37] etc.
- new partitioning algorithms and variations

The component-based framework allows seamless replacement of old algorithms by improved ones in containing applications. Even more important, a solid testbench is *absolutely essential* to identify algorithmic improvements "at the leading edge" of heuristic technology. I.e., it is critical to evaluate proposed algorithm improvements not only against the best available implementations, but also *using a competent implementation*. This is the main point we wish to make.

In our experience, new "improvements" often look good if applied to weak algorithms, but may actually worsen strong algorithms. Only after an improvement has been thoroughly analyzed, implemented and confirmed empirically, can it be turned on by default and be applied to all evaluations of all subsequent proposed improvements. On the other hand, one often encounters pairs of conflicting improvements of which one, if applied by itself, dominates the other while the combination of the two is the worst. Therefore, interacting improvements must be implemented as options, and tested in all possible combinations.

In the following, we focus on the very pernicious danger of reporting "experimental results" that are irreproducible and possibly meaningless due to a poorly implemented partitioning testbench. We demonstrate that a fundamental cause of a poor testbench is failure to understand the "implicit implementation decisions" that dominate quality/runtime tradeoffs. A corollary is that researchers must clearly report such "implicit decisions" in order for results to be reproducible.

3.3 Algorithm and Implementation Improvements

In this subsection we note the existence of several types of implementation decisions for optimization metaheuristics, and illustrate such decisions for the Fiduccia-Mattheyses heuristic [20] and its improvements.

Of particular interest are *implicit decisions* – underspecified features and ambiguities in the original algorithm description that need to be resolved in any particular implementation. Examples for the Fiduccia-Mattheyses heuristic include:

[12] See, e.g., Barr et al. [6]. Separate work of ours has addressed this gap in reporting methodology within the VLSI CAD community [10].

- *tie-breaking* in choosing highest gain bucket (see Subsection 3.4)
- *tie-breaking* on where to attach new element in gain bucket, i.e., LIFO versus FIFO versus random [26][13]
- whether to update, or skip the updating, when the delta gain of a move is zero (see Subsection 3.4)
- breaking ties when selecting the best solution during the pass — choose the first or last one encountered, or the one that is furthest from violating constraints.

Below, we show that the effects of such implicit decisions can far outweigh claimed improvements in solution quality due to algorithm innovation. Other types of implementation decisions, details of which are beyond our present scope, include:

- *Modifications of the algorithm*: important changes to steps or the flow of the original algorithm as well as new steps and features. Among the more prominent examples are "lookahead" tie-breaking [38] among same-gain moves; the *multiple unlocking* heuristic of [15] which allows vertices to move more than once during a pass; and the CLIP heuristic of [18] which chooses moves according to "updated" gains (i.e., the actual gain minus the gain at the beginning of the pass) instead of actual gains.
- *Tuning that can change the result*: minor algorithm or implementation changes, typically to avoid particularly bad special cases or pursue only "promising computations". Examples include thresholding large nets from the input to reduce run time; "loose net" removal [12] where gain updates are performed only for [loose] nets that are likely to be uncut; and allowing of illegal solutions during a pass (to improve hill-climbing ability of the algorithm) [19].
- *Tuning that can not change the result*: minor algorithm or implementation changes to simplify computations in critical or statistically significant special cases. Examples include skipping nets which cannot have non-zero delta gains (updates); code optimizations that are specific to the netcut objective; and code optimizations that are specific to 2-way partitioning.

3.4 An Empirical Illustration

We now illustrate how "implicit implementation decisions" can severely distort the experimental assessment of new algorithmic ideas. Uncertainties in the description of the Fiduccia-Mattheyses algorithm have been previously analyzed, notably in [26], where the authors show that inserting moves into gain buckets in LIFO order is much preferable to doing so in FIFO order (also a constant-time insertion) or at random. Since the work of [26], all FM implementations that we are aware of use LIFO insertion.[14] In our experiments, we consider the following two implicit implementation decisions:

- **Zero delta gain update.** Recall that when a vertex x is moved, the gains for all vertices y on nets incident to x must potentially be updated. In all FM implementations, this is done by going through the incident nets one at a time, and computing the

[13] In other words, gain buckets can be implented as stacks, queues or random priority queues where the chances of all elements to be selected are equal at all times. [26] demonstrated that stack-based gain containers (i.e. LIFO) are superior.

[14] A series of works in the mid-1990s retrospectively show that the LIFO order allows vertices in "natural clusters" to move together across the cutline. The CLIP variant of [18] is a more direct way of moving clusters.

changes in gain for vertices y on these nets. A straightforward implementation computes the change in gain ("delta gain") for y by adding and subtracting four cut values for the net under consideration,[15] and immediately updating y's position in the gain container.
Notice that sometimes the delta gain can be zero. An implicit implementation decision is whether to reinsert a vertex y when it experiences a zero delta gain move (**"AllΔ_{gain}"**), or whether to skip the gain update (**"Nonzero"**). The former will shift the position of y within the same gain bucket; the latter will leave y's position unchanged. The effect of zero delta gain updating is not immediately obvious.[16]

– **Tie-breaking between two highest-gain buckets in move selection.** When the gain container is implemented such that available moves are segregated, typically by source or destination partition, there can be more than one nonempty highest-gain bucket. Notice that when the balance constraint is anything other than "exact bisection", it is possible for all the moves at the heads of the highest-gain buckets to be legal. The FM implementer must choose a method for dealing with this situation. In our experiments, we contrast three approaches:[17] (i) choose the move that is not from the same partition as the last vertex moved (**"away"**); (ii) choose the move in partition 0 (**"part0"**); and (iii) choose the move from the same partition as the last vertex moved (**"toward"**).

Our experimental testbench allows us to test an FM variant in the context of flat LIFO (as described in [26]), flat CLIP (as described in [18]), and multilevel LIFO and multilevel CLIP (as described in [4]). Our implementations are in C++ with heavy use of STL3.0; we currently run in the Sun Solaris 2.6 and Sun CC4.2 environment. We use standard VLSI benchmark instances available on the Web at [1] and several older benchmarks from `http://www.cbl.ncsu.edu/benchmarks`. Node and hyperedge statistics for the benchmarks are presented in Tables 4, 5 and 6. Our tests are for bipartitioning only. We evaluate all partitioning variants using actual vertex areas and unit vertex areas, incorporating the standard protocols for treating pad areas described in [2]. We also evaluate all partitioning variants using both a 10% balance constraint (i.e., each partition must have between 45% and 55% of the total vertex area) as well as a 2% balance constraint (results are qualitatively similar; we therefore report only results for 2% balance constraints). All experiments were run on Sun Ultra workstations, with runtimes normalized to Sun Ultra-1 (140MHz) CPU seconds. Each result represents a set of 100 independent runs with random initial starting solutions; Tables 1 and 2 report triples of form "**average cut** (average CPU sec)". From the data, we make the following observations.

– The average cutsize for a flat partitioner can increase by rather stunning percentages if the worst combination of choices is used instead of the best combination. Such effects far outweigh the typical solution quality improvements reported for new algorithm ideas in the partitioning literature.

[15] These four cut values correspond to: (a) x, y in their original partitions; (b) x in original partition, y moved; (c) x moved, y in original partition; and (d) x and y both moved. (a) - (b) is the original gain for y due to the net under consideration; (c) - (d) is the new gain for y due to the same net. The difference ((a)-(b)) - ((c)-(d)) is the delta gain. See [29] for a discussion.

[16] The gain update method presented in [20] has the *side effect* of skipping all zero delta gain updates. However, this method is both netcut- and two-way specific; it is not certain that a first-time experimenter with FM will find analogous solutions for k-way partitioning with a general objective.

[17] These approaches are described for the case of bipartitioning. Other approaches can be devised for k-way partitioning.

ALGORITHM		TESTCASES with **unit** areas and **2%** balance					
Updates	Bias	**primary1**	**primary2**	**biomed**	**ibm01**	**ibm02**	**ibm03**
				Flat LIFO FM			
All Δ_{gain}	Away	**102**(0.247)	**486**(1.6)	**459**(29.3)	**1778**(16.4)	**1810**(36.5)	**4175**(34.1)
All Δ_{gain}	Part0	**102**(0.27)	**465**(2.06)	**422**(37.8)	**1673**(19.8)	**1570**(43.3)	**4064**(35)
All Δ_{gain}	Toward	**102**(0.264)	**374**(1.94)	**316**(36.5)	**1030**(15.4)	**931**(30)	**3323**(39.6)
Nonzero	Away	**80.5**(0.222)	**285**(1.31)	**166**(22.8)	**543**(10.2)	**549**(18)	**2304**(29.2)
Nonzero	Part0	**80**(0.236)	**289**(1.47)	**150**(26.3)	**551**(11.8)	**549**(18.6)	**2383**(31.1)
Nonzero	Toward	**78.8**(0.219)	**291**(1.41)	**143**(24.8)	**508**(10.9)	**551**(19.3)	**2285**(33.1)
				Flat CLIP FM			
All Δ_{gain}	Away	**69.8**(0.351)	**246**(2.22)	**149**(53.9)	**555**(21.1)	**707**(46.2)	**1747**(45.5)
All Δ_{gain}	Part0	**68**(0.319)	**242**(2.3)	**138**(52)	**562**(21.3)	**636**(45.5)	**1686**(51.7)
All Δ_{gain}	Toward	**68.4**(0.35)	**236**(2.14)	**121**(44)	**561**(19.3)	**619**(44)	**1664**(43.2)
Nonzero	Away	**66.3**(0.284)	**231**(2.02)	**130**(29.1)	**488**(17.5)	**534**(46.8)	**1650**(41.9)
Nonzero	Part0	**66.4**(0.305)	**226**(2.2)	**124**(33.7)	**467**(17.1)	**507**(37.4)	**1618**(41.2)
Nonzero	Toward	**64.6**(0.304)	**223**(2.13)	**125**(31.4)	**474**(15.8)	**559**(36.9)	**1551**(41.9)
				ML LIFO FM			
All Δ_{gain}	Away	**64.2**(0.818)	**183**(4.95)	**147**(36.8)	**280**(29.7)	**399**(70.6)	**1043**(138)
All Δ_{gain}	Part0	**63.8**(0.871)	**182**(5.21)	**145**(33.5)	**282**(31.6)	**406**(66.8)	**999**(113)
All Δ_{gain}	Toward	**63.2**(0.836)	**180**(4.74)	**145**(33.6)	**272**(30.8)	**421**(66.1)	**1035**(118)
Nonzero	Away	**62.9**(0.807)	**182**(4.69)	**142**(28.1)	**274**(29.7)	**396**(60.4)	**1037**(123)
Nonzero	Part0	**61.8**(0.849)	**176**(5.15)	**144**(26.8)	**270**(28.8)	**403**(60.2)	**1048**(121)
Nonzero	Toward	**61.4**(0.879)	**181**(4.91)	**143**(24.8)	**271**(29.1)	**415**(56)	**1015**(119)
				ML CLIP FM			
All Δ_{gain}	Away	**63.4**(0.912)	**180**(5.27)	**146**(32.6)	**276**(30.2)	**400**(68.7)	**1042**(138)
All Δ_{gain}	Part0	**62.8**(0.849)	**180**(5.18)	**145**(39.1)	**278**(29.1)	**408**(67.8)	**1022**(129)
All Δ_{gain}	Toward	**63.3**(0.871)	**178**(5.36)	**141**(36.7)	**270**(29.4)	**425**(64.4)	**1032**(101)
Nonzero	Away	**61.2**(0.887)	**178**(5.12)	**142**(26.8)	**268**(28.4)	**392**(57)	**1037**(116)
Nonzero	Part0	**62.8**(0.878)	**176**(5.14)	**142**(29.3)	**275**(28.6)	**409**(55.6)	**1026**(111)
Nonzero	Toward	**63**(0.924)	**179**(4.86)	**142**(28.5)	**271**(28.2)	**409**(52.9)	**1034**(115)

Table 1. Average cuts in partitioning with **unit** areas and **2%** balance tolerance, over 100 independent runs. Average CPU time per run in Sun Ultra-1 (140MHz) seconds is given in parentheses.

– Moreover, we see that one wrong implementation decision can lead to misleading conclusions with respect to other implementation decisions. For example, when zero delta gain updates are made (a wrong decision), the “part0” biasing choice appears significantly worse than the “toward” choice. However, when zero delta gain updates are skipped, “part0” is as good as or even slightly better than “toward”.[18]

– Stronger optimization engines (order of strength: ML CLIP $>$ ML LIFO $>$ flat CLIP $>$ flat LIFO) can tend to decrease the “dynamic range” for the effects of implementation choices. This is actually a danger: e.g., developing a multilevel FM package may hide the fact that the underlying flat engines are badly implemented. At the same time, the effects of a bad implementation choice are still apparent even when that choice is wrapped within a strong optimization technique (e.g., ML CLIP).

ALGORITHM		TESTCASES with **actual** areas and **2%** balance					
Updates	Bias	**primary1**	**primary2**	**biomed**	**ibm01**	**ibm02**	**ibm03**
				Flat LIFO FM			
All Δ_{gain}	Away	**105**(0.283)	**481**(2.05)	**446**(27.8)	**1885**(11.1)	**3256**(34.8)	**4389**(25.4)
All Δ_{gain}	Part0	**103**(0.311)	**467**(2.4)	**428**(34.9)	**1909**(12.3)	**2440**(33.5)	**4166**(27.1)
All Δ_{gain}	Toward	**96.8**(0.285)	**380**(2.15)	**408**(30.7)	**1023**(11.6)	**1274**(22.7)	**3939**(22.3)
Nonzero	Away	**80.5**(0.258)	**291**(1.78)	**164**(27.3)	**639**(8.86)	**551**(14.9)	**2838**(25.2)
Nonzero	Part0	**79**(0.271)	**294**(1.74)	**157**(29.2)	**660**(7.87)	**573**(17)	**2938**(24)
Nonzero	Toward	**79.7**(0.25)	**280**(1.79)	**153**(24.7)	**607**(7.62)	**543**(15.8)	**2843**(25.4)
				Flat CLIP FM			
All Δ_{gain}	Away	**66.2**(0.361)	**244**(2.71)	**148**(57)	**842**(15.3)	**1841**(27.5)	**3623**(23.4)
All Δ_{gain}	Part0	**66.3**(0.395)	**242**(2.81)	**141**(54.6)	**772**(14.9)	**1499**(32.4)	**3543**(29.3)
All Δ_{gain}	Toward	**65.9**(0.393)	**237**(2.7)	**138**(58)	**615**(13)	**945**(21.5)	**3066**(25.6)
Nonzero	Away	**64.5**(0.351)	**231**(2.66)	**130**(33.3)	**542**(12.1)	**574**(18.5)	**2689**(22.8)
Nonzero	Part0	**62**(0.371)	**242**(2.49)	**123**(35)	**556**(12.4)	**582**(17.8)	**2732**(23.1)
Nonzero	Toward	**63.9**(0.377)	**233**(2.31)	**124**(34.9)	**528**(11.8)	**562**(15.3)	**2504**(23.2)
				ML LIFO FM			
All Δ_{gain}	Away	**62.8**(0.905)	**163**(5.27)	**145**(34.9)	**289**(27.9)	**433**(42.9)	**958**(59.4)
All Δ_{gain}	Part0	**62.5**(0.954)	**166**(5.03)	**144**(38.7)	**289**(27.2)	**429**(44.6)	**957**(56.3)
All Δ_{gain}	Toward	**61.1**(0.974)	**161**(5.28)	**143**(36.3)	**289**(27.7)	**423**(47)	**971**(58.7)
Nonzero	Away	**60.4**(0.914)	**158**(4.52)	**142**(29.9)	**287**(22.7)	**432**(39.6)	**969**(52.2)
Nonzero	Part0	**59.9**(0.882)	**158**(4.36)	**142**(29.3)	**282**(25.3)	**421**(44)	**952**(50.6)
Nonzero	Toward	**60.5**(0.9)	**159**(4.5)	**142**(27.5)	**276**(25.4)	**419**(43.2)	**959**(52.5)
				ML CLIP FM			
All Δ_{gain}	Away	**63.5**(0.88)	**163**(5)	**144**(35.9)	**283**(24.5)	**428**(41.5)	**960**(59.3)
All Δ_{gain}	Part0	**62.5**(0.891)	**161**(4.24)	**143**(37.1)	**289**(25.3)	**441**(46.5)	**969**(63.6)
All Δ_{gain}	Toward	**61.9**(0.927)	**162**(4.83)	**141**(37.8)	**284**(24.9)	**425**(44.2)	**953**(62.1)
Nonzero	Away	**60.2**(0.939)	**160**(4.66)	**144**(31.8)	**283**(23)	**414**(48.7)	**957**(50.4)
Nonzero	Part0	**61**(0.895)	**161**(4.89)	**144**(28.1)	**285**(24.7)	**447**(41.8)	**934**(53)
Nonzero	Toward	**60.9**(0.864)	**155**(4.7)	**144**(29.6)	**282**(22.4)	**433**(42.6)	**959**(50.6)

Table 2. Average cuts in partitioning with **actual** areas and **2%** balance tolerance, over 100 independent runs. Average CPU time per run in Sun Ultra-1 (140MHz) seconds is given in parentheses.

[18] We have observed other similar reversals, e.g., in our experience multiple unlocking is less valuable than reported in [15].

4 Conclusions

The results reported in the previous section are for a "detuned" or "generalized" version of our testbench, where we deliberately re-enabled the zero delta gain update and gain bucket choice as options. In our current testbench, these are not options, i.e., our FM-based engines always skip zero delta gain updates and always choose the "toward" gain bucket in case of ties. Our current testbench is also able to invoke several speedups that exploit the nature of the netcut objective and the two-way partitioning context. Comparison of the results in 3 against the best netcut values ever recorded in the literature [2] shows that our testbench is indeed at the leading edge of solution quality.) We emphasize that our testbench is general: we flexibly address new objectives, neighborhood structures, and constraint types. We therefore incur some runtime overhead due to templating, object-oriented code structure, conditional tests, etc. At the same time, our testbench is sufficiently fast that we can accurately assess quality-runtime tradeoff implications of new algorithm ideas.

Configuration	primary1	primary2	biomed	ibm01	ibm02	ibm03
2% unit area	57.1(0.418)	164.7(1.3)	132.8(2.24)	275.0(5.7)	372.1(10.6)	1031.4(11.6)
10% unit area	52.1(0.412)	143.6(1.3)	117.0(2.16)	247.9(5.82)	268.8(10.3)	820.9(11.8)
2% actual area	61.3(0.389)	193.1(1.41)	137.2(2.07)	284.5(6.63)	421.3(14.6)	1079.8(15.8)
10% actual area	54.1(0.421)	177.8(1.07)	125.0(2.23)	244.4(5.36)	275.7(14.7)	1062.1(16.4)

Table 3. Results of applying our optimized multilevel partitioner on 6 test-cases. Solutions are constrained to be within 2% or 10% of bisection. Data expressed as (average cut / average CPU time) over 100 runs, with the latter normalized to CPU seconds on a 140MHz Sun Ultra-1.

In conclusion, we have noted that replicating reported results is a well-known problem in the VLSI partitioning community [5]. Disregarding the issues of experimental protocols, statistical significance tests, data reporting methodology, need to compare with previous results, and so on [6, 8], we still find that implementations reported in the literature are almost never described in sufficient detail for others to reproduce results. In this paper, we have illustrated the level of detail necessary in reporting; our illustration also shows how even expert programmers may fail to write a useful testbench for research "at the leading edge". Our main contributions have been the description of a software architecture for partitioning research, a review of the modern context for such research in the VLSI CAD domain, and a sampling of hidden implementation decisions that are crucial to obtaining a useful testbench.

5 Acknowledgments

We thank Max Moroz for design, implementation and support of an efficient STL-based hypergraph package used in the paper in lieu of the LEDA library.

References

1. C. J. Alpert, "Partitioning Benchmarks for VLSI CAD Community", Web page, `http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html`
2. C. J. Alpert, "The ISPD-98 Circuit Benchmark Suite", *Proc. ACM/IEEE International Symposium on Physical Design*, April 98, pp. 80-85. See errata at `http://vlsicad.cs.ucla.edu/~cheese/errata.html`

Node degree statistics for testcases					
primary1(833)	primary2(3014)	biomed(6514)	ibm01(12752)	ibm02(19601)	ibm03(23136)
deg: #	deg: #	deg: #	deg: #	deg: #	deg: #
Avg: 3.49	Avg: 3.72	Avg: 3.230	Avg: 3.965	Avg: 4.143	Avg: 4.044
1: 48	1: 43	1: 97	1: 781	1: 1591	1: 363
2: 145	2: 453	2: 792	2: 3722	2: 4448	2: 7093
3: 205	3: 1266	3: 4492	3: 2016	3: 2318	3: 4984
4: 273	4: 519	4: 440	4: 1430	4: 1714	4: 4357
5: 234	5: 402	5: 35	5: 1664	5: 3406	5: 2778
6: 5	6: 23	6: 658	6: 1761	6: 4435	6: 1252
7: 22	7: 260		7: 542	7: 1055	7: 300
9: 1	8: 4		8: 194	8: 303	8: 689
	9: 44		9: 368	9: 319	9: 708
			10: 3	29: 2	10: 50
			13: 220	32: 1	11: 25
			39: 1	51: 4	12: 192
				53: 3	13: 47
				60: 1	14: 7
				69: 1	16: 10
					17: 7
					18-19: 4
					20: 16
					21-23: 8
					24: 136
					25: 101
					84-100: 9

Table 4. Hypergraph node degree statistics. The numbers of nodes in degree ranges are given for each testcase together with the total nodes and average node degree.

3. C. J. Alpert and L. W. Hagen and A. B. Kahng, "A Hybrid Multilevel/Genetic Approach for Circuit Partitioning", *Proc. IEEE Asia Pacific Conference on Circuits and Systems*, 1996, pp. 298-301.
4. C. J. Alpert, J.-H. Huang and A. B. Kahng,"Multilevel Circuit Partitioning", *ACM/IEEE Design Automation Conference*, pp. 530-533.
   `http://vlsicad.cs.ucla.edu/papers/conference/c68.ps`
5. C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", *Integration*, 19(1995) 1-81.
6. R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende and W. R. Stewart, "Designing and Reporting on Computational Experiments with Heuristic Methods", *technical report* (extended version of *J. Heuristics* paper), June 27, 1995.
7. F. Brglez, "ACM/SIGDA Design Automation Benchmarks: Catalyst or Anathema?", *IEEE Design and Test*, 10(3) (1993), pp. 87-91.
8. F. Brglez, "Design of Experiments to Evaluate CAD Algorithms: Which Improvements Are Due to Improved Heuristic and Which are Merely Due to Chance?", *technical report* CBL-04-Brglez, NCSU Collaborative Benchmarking Laboratory, April 1998.
9. T. Bui, S. Chaudhuri, T. Leighton and M. Sipser, "Graph Bisection Algorithms with Good Average Behavior", *Combinatorica* 7(2), 1987, pp. 171-191.
10. A. E. Caldwell, A. B. Kahng and I. L. Markov, *manuscript*, 1998.
11. P. K. Chan and M. D. F. Schlag and J. Y. Zien, "Spectral *K*-Way Ratio-Cut Partitioning and Clustering", *IEEE Transactions on Computer-Aided Design*, vol. 13 (8), pp. 1088-1096.
12. J. Cong, H. P. Li, S. K. Lim, T. Shibuya and D. Xu, "Large Scale Circuit Partitioning with Loose/Stable Net Removal and Signal Flow Based Clustering", *Proc. IEEE International Conference on Computer-Aided Design*, 1997, pp. 441-446.
13. J. Cong and S. K. Lim, "Multiway Partitioning with Pairwise Movement", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1998, to appear.
14. J. Darnauer and W. Dai, "A Method for Generating Random Circuits and Its Applications to Routability Measurement", *Proc. ACM/SIGDA International Symposium on FPGAs*, 1996, pp. 66-72.
15. A. Dasdan and C. Aykanat, "Two Novel Multiway Circuit Partitioning Algorithms Using Relaxed Locking", *IEEE Transactions on Computer-Aided Design* 16(2) (1997), pp. 169-178.
16. W. Deng, *personal communication*, July 1998.
17. A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard Cell VLSI Circuits", *IEEE Transactions on Computer-Aided Design* 4(1) (1985), pp. 92-98
18. S. Dutt and W. Deng, "VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques", *Proc. IEEE International Conference on Computer-Aided Design*, 1996, pp. 194-200
19. S. Dutt and H. Theny, "Partitioning Using Second-Order Information and Stochastic Gain Function", *Proc. IEEE/ACM International Symposium on Physical Design*, 1998, pp. 112-117
20. C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions", *Proc. ACM/IEEE Design Automation Conference*, 1982, pp. 175-181.

Node weight statistics for testcases						
weight range	# nodes					
	primary1	primary2	biomed	ibm01	ibm02	ibm03
1	81	107	97	12749	19589	23126
2				2	3	
3					3	
4	89	367				
5			723			7
6	142	715			1	
7	151	494	2818			1
8	73	398				
9			1323		1	
10						
11	27	278			11	
12			35			
13						
14			3			
15	7	550				
16						
17	1	52				
18			860			
19						
20	262	53	655	1	1	2

Table 5. Hypergraph node weight statistics. The interval between the smallest and largest node weight has been divided into 20 equal ranges for each testcase. For each such range we report the number of nodes with weight in this range.

21. 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, pp. 223
22. M. Ghose, M. Zubair and C. E. Grosch, "Parallel Partitioning of Sparse Matrices", *Computer Systems Science & Engineering* (1995) 1, pp. 33-40.
23. D. Ghosh, "Synthesis of Equivalence Class Circuit Mutants and Applications to Benchmarking", *summary of presentation at DAC-98 Ph.D. Forum*, June 1998.
24. M. K. Goldberg and M. Burstein, "Heuristic Improvement Technique for Bisection of VLSI Networks", *IEEE Transactions on Computer-Aided Design*, 1983, pp. 122-125.
25. S. Hauck and G. Borriello, "An Evaluation of Bipartitioning Techniques", *IEEE Transactions on Computer-Aided Design* 16(8) (1997), pp. 849-866.
26. L. W. Hagen, D. J. Huang and A. B. Kahng, "On Implementation Choices for Iterative Improvement Partitioning Methods", *Proc. European Design Automation Conference*, 1995, pp. 144-149.
27. B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs", draft, 1995.
28. B. Hendrickson and T. G. Kolda, "Partitioning Rectangular and Structurally Nonsymmetric Sparse Matrices for Parallel Processing", *manuscript*, 1998 (extended version of PARA98 workshop proceedings paper).
29. D. J. Huang and A. B. Kahng, "Partitioning-Based Standard Cell Global Placement with an Exact Objective", *Proc. ACM/IEEE International Symposium on Physical Design*, 1997, pp. 18-25.
`http://vlsicad.cs.ucla.edu/papers/conference/c66.ps`
30. M. Hutton, J. P. Grossman, J. Rose and D. Corneil, "Characterization and Parameterized Random Generation of Digital Circuits", *In Proc. IEEE/ACM Design Automation Conference*, 1996, pp. 94-99.
31. A. B. Kahng, "Futures for Partitioning in Physical design", *Proc. IEEE/ACM International Symposium on Physical Design*, April 1998, pp. 190-193.
`http://vlsicad.cs.ucla.edu/papers/conference/c77.ps`
32. G. Karypis and V. Kumar, "Analysis of Multilevel Graph Partitioning", draft, 1995
33. G. Karypis and V. Kumar, "Multilevel *k*-way Partitioning Scheme For Irregular Graphs", Technical Report 95-064, University of Minnesota, Computer Science Department.
34. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design", *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 526-529.
Additional publications and benchmark results for hMetis-1.5 are avilable at
`http://www-users.cs.umn.edu/ ~karypis/metis/hmetis/main.html`
35. G. Karypis and V. Kumar, "Multilevel Algorithms for Multi-Constraint Graph Partitioning", Technical Report 98-019, University of Minnesota, Department of Computer Science.
36. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Tech. Journal* 49 (1970), pp. 291-307.
37. C. Kring and A. R. Newton, "A Cell-Replicating Approach to Mincut-Based Circuit Partitioning", *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1991, pp. 2-5.

Hyperedge statistics for testcases					
primary1(902)	primary2(3029)	biomed(5742)	ibm01(14111)	ibm02(19568)	ibm03(27401)
deg: #	deg: #	deg: #	deg: #	deg: #	deg: #
Avg: 3.22	Avg: 3.70	Avg: 3.664	Avg: 3.583	Avg: 4.146	Avg: 3.415
2: 494	2: 1835	2: 3998	2: 8341	2: 10692	2: 17619
3: 236	3: 365	3: 870	3: 2082	3: 1934	3: 3084
4: 62	4: 203	4: 427	4: 1044	4: 1951	4: 2155
5: 26	5: 192	5: 184	5: 737	5: 1946	5: 1050
6: 25	6: 120	6: 13	6: 407	6: 376	6: 790
7: 13	7: 52	7: 11	7: 235	7: 332	7: 436
8: 2	8: 14	8: 28	8: 188	8: 256	8: 342
9: 9	9: 83	9: 7	9: 192	9: 424	9: 501
10: 1	10: 14	10: 4	10: 194	10: 431	10: 235
11: 6	11: 35	11: 5	11: 147	11: 498	11: 198
12: 9	12: 5	12: 5	12: 91	12: 46	12: 162
13: 1	13: 3	13: 1	13: 133	13: 52	13: 195
14: 3	14: 10	14: 2	14: 54	14: 52	14: 112
16: 1	15: 3	15: 41	15: 34	15: 85	15: 79
17: 11	16: 1	17: 21	16: 54	16: 94	16: 100
18: 3	17: 72	18: 1	17: 31	17: 143	17: 119
	18: 1	20: 2	18: 17	18: 100	18: 81
	23: 1	21: 65	19: 12	19: 44	19: 41
	26: 1	22: 34	20: 21	20: 15	20: 24
	29: 1	23: 6	21: 18	21: 11	21: 12
	30: 1	24: 6	22: 31	22: 5	22: 16
	31: 1	43: 6	23: 18	24-29: 10	23: 9
	33: 14	656: 4	25: 2	30: 4	24: 3
	34: 1	861: 1	28: 1	31: 11	25: 6
	37: 1		30: 2	32: 4	26: 3
			31: 2	33: 2	27: 2
			32: 5	34: 1	28: 2
			33: 6	35: 5	29: 6
			34: 1	36: 3	30: 1
			35: 7	37: 2	31: 2
			38: 1	38: 2	31: 3
			39: 2	39: 1	32: 3
			42: 1	40-97: 51	33: 5
				107: 1	34: 3
				134: 1	37-55: 5

Table 6. Hyperedge statistics. The numbers of hyperedges in ranges are given for each testcase together with total hyperedges and average hyperedge degree.

38. B. Krishnamurthy, “An Improved Min-cut Algorithm for Partitioning VLSI Networks”, *IEEE Transactions on Computers*, vol. C-33, May 1984, pp. 438-446.
39. B. Landman and R. Russo, “On a Pin Versus Block Relationship for Partitioning of Logic Graphs”, *IEEE Transactions on Computers* C-20(12) (1971), pp. 1469-1479.
40. L. T. Liu, M. T. Kuo, S. C. Huang and C. K. Cheng, “A Gradient Method on the Initial Partition of Fiduccia-Mattheyses Algorithm”, *Proc. IEEE International Conference on Computer-Aided Design*, 1995, pp. 229-234.
41. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Teubner, 1990.
42. L. Sanchis, “Multiple-way network partitioning with different cost functions”, *IEEE Transactions on Computers*, Dec. 1993, vol.42, (no.12):1500-4.
43. P. R. Suaris and G. Kedem, “Quadrisection: A New Approach to Standard Cell Layout”, *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1987, pp. 474-477.
44. W. Sun and C. Sechen, “Efficient and Effective Placements for Very Large Circuits”, *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1993, pp. 170-177.
45. Y. C. Wei and C. K. Cheng, “Towards Efficient Design by Ratio-cut Partitioning”, *Proc. IEEE International Conference on Computer-Aided Design*, 1989, pp. 298-301.

Algorithms for Restoration Planning in a Telecommunications Network

S. Cwilich* M. Deng* D.F. Lynch* S.J. Phillips† J.R. Westbrook†

{seba,dflynch,mdeng}@att.com,{phillips,jeffw}@research.att.com

Abstract

One of the major tasks of telecommunications network planners is deciding where and how much spare capacity to build so that disrupted traffic may be rerouted in the event of a network failure.

This paper presents an experimental study comparing two techniques for restoration capacity planning. The first, linear programming using column generation, produces optimal fractional solutions that can be integerized in practice for little extra cost. This approach is time-consuming, however, and for some networks of interest the linear programs are too large to be practically solvable. The second algorithm is a fast heuristic called LOCAL, which can be practically applied to much larger problem sizes than column generation. A fast linear-programming lower bound is used to measure the efficiency of the solutions.

The purpose of the study was threefold: to determine how much benefit is obtained by using column generation, when the problem size is small enough that column generation is practical; to determine the quality of solutions produced by LOCAL on both small and large problems; and to investigate the utility of the lower-bound LP. We find that column generation produces networks whose restoration capacity cost is 10% to 16% less than those produced by LOCAL. Sometimes the total cost of the network is of primary interest, including both the cost of service and restoration capacity, and in this case the difference between column generation and LOCAL is 4% to 6%.

Column generation is the method of choice for making final decisions about purchasing spare capacity. When millions are to be spent, running time is not a consideration, unless the computation simply will not complete in the available time. In such cases, the differential between column generation and LOCAL is small enough and consistent enough that LOCAL can safely be used. LOCAL is the method of choice for network design "spreadsheet" applications, where different architectures can be quickly compared, modified, and then compared again, and where a consistent approximation to the optimum is sufficient. The bound produced by the lower-bound LP is consistently extremely close to the true optimum,

*AT&T Labs, 200 Laurel Avenue, Middletown, NJ 07748

†AT&T Labs-Research, 180 Park Ave, Florham Park, NJ 07932

so the lower-bound LP can be effectively used to monitor the performance of both methods.

1 Introduction

Network reliability is a major concern of telecommunications carriers. Their networks are subject to a variety of faults, including power outages, equipment failures, natural disasters and cable cuts. Customers demand uninterrupted service, so telecommunications networks must be designed for reliability in the face of failures. This means that capacity must be set aside to allow interrupted traffic to be restored by rerouting it around a fault. This spare capacity is called *restoration capacity*. The non-spare capacity on which traffic is normally carried is called *service capacity*.

Restoration capacity accounts for a large part of the infrastructure cost of telecommunications networks, so allocating restoration capacity is a major task for network planners. The problem of restoration capacity planning (also known as the spare capacity assignment problem) is to determine where and how much spare capacity to install in a network, to be able to restore disrupted services in the event of any failure, while minimizing facility cost.

This paper presents an experimental study of restoration capacity planning. Two approaches are compared, linear programming using column generation, and a fast heuristic called LOCAL [9]. A linear-programming lower bound is used to measure the efficiency of the solutions.

The column generation method produces optimal fractional solutions, and in practice integerization often produces only a small increase in the cost of the solution. The method has been used on some fairly large networks, including parts of AT&T's FASTAR-protected DS3 network. However, the method is time-consuming, and some telecommunications networks are too big to allow restoration capacity planning via column generation, so other methods must also be used.

On the other hand, LOCAL has been effectively used on very large networks [4]. It lacks the performance guarantee of column generation, and performs badly on pathological examples, but its performance can be monitored using the lower bound LP.

The purpose of this study is to study the performance of LOCAL and column generation on problems in the target domain of telecommunication networks. There were three goals underlying the study. First, determine how much benefit is obtained through the use of column generation instead of simpler methods, on networks for which it is feasible, and investigate the range of network sizes on which it is feasible. Second, determine how close to optimal LOCAL is, for networks on which the optimum can be obtained through column generation, in order to give confidence in LOCAL's performance on larger networks where the optimal solution is unavailable. Third, determine how tight the lower bound is, on practical examples, in order to estimate its effectiveness in evaluating heuristic solutions for large networks. Our experimental analysis uses data from real telecommunications applications, and random test data that was generated so as to match the characteristics of the real applications.

This paper is organized as follows. The next section describes and motivates the restoration capacity planning problem in more detail. Section 3 outlines the column generation method, while Sections 4 and 5 describe the LOCAL algorithm and the lower bound linear program respectively. The data used in the study is presented in Section 6, the results are in Section 7, and the results are analyzed in Section 8.

2 The Restoration Capacity Planning Problem

The input to the restoration capacity planning problem is a network N and a set of routed demands. The network consists of a set of n nodes and m links. For each edge there is a cost, typically representing the capital cost of equipment required on the link to support a unit of capacity. Each demand is described by a source-destination pair, the size or required capacity, and a service route in the network.

The output of a planning algorithm is a set of capacities and restoration routes for each demand. Each failure that affects the demand's service route must be associated with a single restoration route that bypasses the failure.

The objective is to minimize the total cost of the solution, which is determined from the edge capacities. Once protection routes have been determined for each demand, it is straightforward to determine the restoration capacity required on each edge. For each failure f, determine the collection of restoration paths that will be used. For each edge e in the network, determine how much capacity, $cap(e, f)$, is needed to carry the protection paths that are in use. The required restoration capacity of edge e is the maximum over failures f of $cap(e, f)$. Thus protection capacity is shared between different failures.

For some applications, only the edge capacities are required, and not the restoration routes themselves. An example is AT&T's FASTAR system, which doesn't use pre-planned restoration paths.

This problem is rather different from purely graph-theoretic network-reliability problems such as connectivity augmentation (for example adding a min-cost set of edges to a graph to increase its connectivity [5, 6]), and finding disjoint paths (see for example Kleinberg's thesis [8]). The restoration capacity planning problem differs from these more extensively studied problems in having a fixed underlying network and an explicit representation of the set of failures and the affected demands.

One of the most serious types of transport network failures is a fiber cut or fiber damage. In the worst case, an entire fiber bundle is cut through, interrupting all traffic passing along a fiber route. This is known as link failure. Because of their prevalence and severity, link failures typically subsume other network failures for planning purposes. This paper considers only link failures, though the restoration planning methods described here are applicable to other failure modes.

In general, restoration paths are unrestricted: they may overlap part of the service path, as long as they avoid the failed link. We consider also a restricted restoration scheme, called path restoration, in which there is a single restoration path for each demand, and the restoration path must be link-diverse from the service path. This restriction arises when fault-detection occurs only in the terminating equipment at the source and destination of the demand, since in that case a fault cannot be isolated to any portion of the path, and the same restoration path must be used to restore all link failures on the service path.

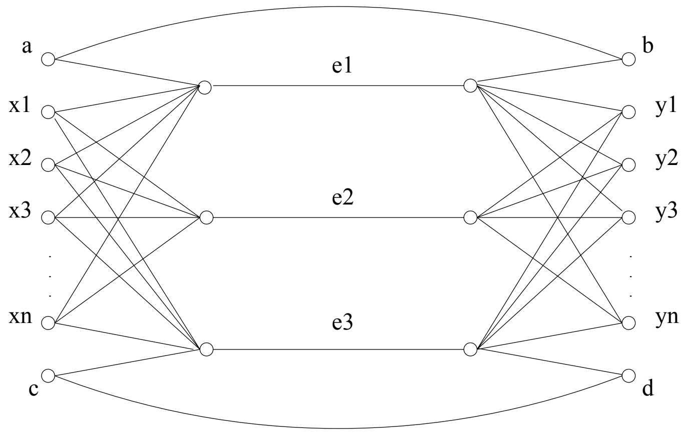


Figure 1: The reduction of Partition to the Restoration Capacity Planning Problem

The NP-hardness of restoration planning, both general and path-based, follows by reduction from the Partition problem. Given an instance of Partition consisting of n numbers $p_1 \dots p_n$, construct the network shown in Figure 1. There is a demand between each x_i and y_i with size p_i routed over e_2, and demands between a and b and between c and d each with size $\frac{1}{2} \sum p_i$ routed over their respective curved edges. The edge cost of all edges incident to exactly one endpoint of $e_1 \dots e_3$ is made high to discourage their use; other edges have low cost. Considering the failure of edges $e_1 \dots e_3$, we see that the minimum cost solution has restoration capacity of $\frac{1}{2} \sum p_i$ on each of e_1 and e_3 if and only if there is an exact partition of the numbers $p_1 \dots p_n$.

3 Column Generation

The column generation method repeatedly solves a linear program relaxation to choose between a set of candidate restoration paths for each demand and each failure. The dual variables from each solution are used to generate better candidate restoration paths for the next iteration. (See Chvátal's book [1] for a general introduction to column generation methods.) After a final set of paths has been assigned possibly fractional weights, an integerization step must be applied. Column-generation formulations for path-selection optimization problems have proven successful in practice and preferable to LP formulations based on conservation of flow.

Here we present the linear programming formulation and show how the dual can be used to generate improved candidate paths. The demands are first aggregated, with all demands between a particular pair of nodes and having the same service route being aggregated into a single demand type. The constants used to describe the linear program are shown in Table 1. The variables, which must all be non-negative, are:

y_r	:	restoration capacity for link r
x_{pft}	:	amount of demand of type $t \in T_f$ routed on path $p \in P_{ft}$ when link $f \in L$ fails

Notation	Meaning
L	The set of links
T_f	The set of demands needing restoration if link f fails
d_{ft}	Number of units of demand of type $t \in T_f$
P_{ft}	The set of candidate restoration paths for $t \in T_f$
Q_{rft}	Set of paths in P_{ft} that use link r, $\{\}$ if r is on the service path of t
R_{pft}	The set of links on path $p \in P_{ft}$ that are not on the service path of t
c_r	Cost per unit capacity for link r

Table 1: Constants to describe the column generation linear program.

The objective is to minimize

$$\sum_{r \in L} c_r y_r$$

subject to the constraints that all demand must be restored

$$\sum_{p \in P_{ft}} x_{pft} = d_{ft} \qquad \forall f \in L, t \in T_f$$

and that the restoration capacity is sufficient on each link for each failure

$$\sum_{t \in T_f} \sum_{p \in Q_{rft}} x_{pft} \leq y_r \qquad \forall r \in L, f \in L.$$

The dual formulation is also straightforward to describe. The dual variables and their corresponding primal constraints are as follows:

u_{ft} : unrestricted, all of demand t must be restored when link f fails

v_{rf} : non-negative, link r needs capacity for traffic rerouted over it when link f fails.

The objective is to maximize

$$\sum_{f \in L} \sum_{t \in T_f} d_{ft} u_{ft}$$

subject to the constraints

$$-u_{ft} + \sum_{r \in R_{pft}} v_{rf} \geq 0 \qquad \forall p \in P_{ft}, t \in T_f, f \in L \tag{1}$$

$$\sum_{f \in L} v_{rf} \leq c_r \qquad \forall r \in L. \tag{2}$$

(3)

The key to the path-generation step is to interpret the variable v_{rf} as the length of link r, for a particular failed link f, and notice that constraint (1) is upper-bounding u_{ft} by the length of the shortest candidate restoration path for demand t. Consider the shortest path (still using v_{rf} as the length of each edge r) between the endpoints of demand t: if this path is not a candidate restoration path, then adding it tightens the constraint on u_{ft}.

The path-generation step is therefore as follows. For each failure f, compute the shortest path between the endpoints of each demand t in T_f, using v_{rf} as the length of each edge r. If the shortest path is shorter than the current value of u_{ft} (i.e., it is shorter than the shortest candidate restoration path in P_{ft}) it is added to P_{ft}. If no new path is added to to P_{ft} for any t and f, an optimum solution has been found.

The number of variables in the linear program is m plus the number of candidate restoration paths. No bound on the number of candidate restoration paths is known, other than the basic exponential bound on the number of paths in a network. In practice, however, a small constant number of candidate paths is generated for each demand and link failure, on average, in which case the number of variables for each linear program is $m + O(\text{sum of lengths of all demands})$.

After a fractional solution is found it must be integerized, *i.e.* a single restoration path must be derived for each demand and failure. While integerization is hard in general, in practice many of the x_{ift} and y_r variables are integral. For some problems, most y_r variables are integral, while for others (including the test problems in this paper) half of them are fractional.

The path-generation software used for this paper is described in [2, 3]. It uses a dynamic path control scheme to control the number of candidate restoration paths, pruning unused candidate paths when the number exceeds a threshold. A natural integerization procedure is used: fractional edge variables with the largest fractions are rounded up and the final LP is rerun with those values fixed. This procedure is repeated until the solution is entirely integral.

4 Algorithm LOCAL

Algorithm LOCAL considers the demands sequentially in a single pass. It builds restoration paths for each demand in turn, taking advantage of spare capacity that was created for earlier demands. For each demand, it approximates the minimum cost augmentation of the existing restoration capacity that allows the demand to be restored for any failure along its path.

The restoration paths that LOCAL creates for a demand consist of a set of local bypasses. Consider the demand shown schematically in Figure 2, whose service route is the straight path from A to F. The curved lines represent local bypasses, each of which is a path that is node-disjoint from the service path except at the endpoints. (In Figure 2 the intermediate nodes of bypasses have been omitted for clarity.) There are three restoration paths. The first, from A to C over the first curve followed by the straight path from C to F, can restore a failure of link $A - B$ or link $B - C$. The other two restoration paths can restore faults further down the service route. If two bypasses cover a failure (*e.g.*, edge BC in Figure 2) then either can be used. (In real applications,

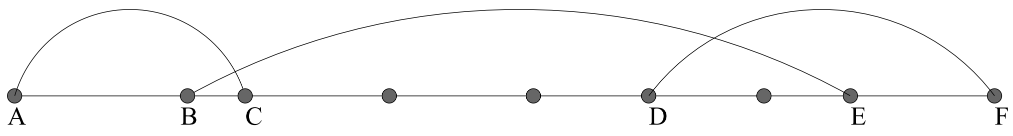


Figure 2: A virtual path and its local bypasses

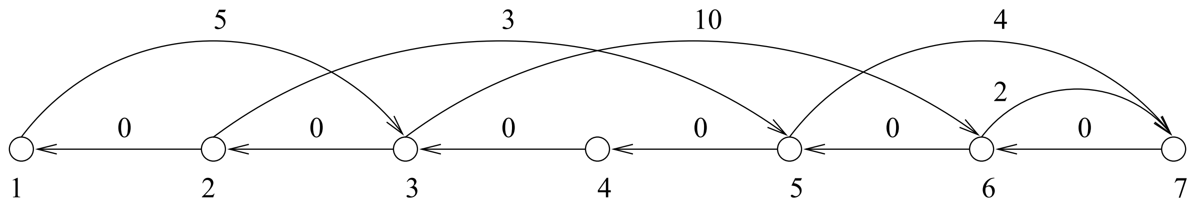


Figure 3: The directed graph H

one will be selected based on issues such as the type of fault-detection equipment.)

LOCAL uses the following process to construct bypasses. Consider a demand D, routed along the path P with nodes $a_1 \dots a_k$ (and edges $e_1 \dots e_{k-1}$). We first construct a directed graph H on k nodes (labeled $1 \dots k$), with edges corresponding to potential bypasses. For $i < j$, an edge (i, j) (called a *forward* edge) is included in H if there is a path in the network N from a_i to a_j that is node-disjoint from P, except at the endpoints. The length of (i, j) is set to the least weight such path, according to an assignment of edge weights in N described below. H is also given zero-length back edges, from i to $i - 1$ for each i.

Let SP be the shortest path from node 1 to node k in H, using the edge lengths as specified. The bypasses for P are exactly the paths in N corresponding to the forward edges of SP.

An example graph H is shown in Figure 3. The shortest path from node 1 to node 7 follows the node sequence $\{1, 3, 2, 5, 7\}$, and the bypasses are the paths in N corresponding to the edges $(1, 3)$, $(2, 5)$ and $(5, 7)$ in H.

To complete the presentation of LOCAL we need to describe how edges are weighted in N, to reflect capacity that has already been provisioned, when computing the lengths of forward edges in H. We keep two arrays, `failload` and `cap`. For each pair of edges e and f in N, `failload`$[e][f]$ is the amount of traffic, from the demands considered so far, that is rerouted onto edge e when edge f fails. For an edge e, `cap`$[e]$ is the capacity required on e so far, namely the maximum of `failload`$[e][f]$. After the local bypasses are constructed for the demand being processed, the arrays `failload` and `cap` are updated, to be used when processing the next demand.

To determine the length of the forward edge (i, j) in H, each edge $e \in N$ is weighted as follows. Let S be the spare capacity available on e under the worst case failure covered by the forward edge (i, j): $S = \texttt{cap}[e] - \max_{i \le k < j} \texttt{failload}[e][e_k]$. If S is at least the size of the demand D, the weight of e is set to a positive constant that is much less than the cost of an edge. Otherwise it is set to the cost of e times $(\text{size}(D) - S)/\text{size}(D)$, to reflect extra restoration capacity that must be built for the worst-case failure that is covered by the forward edge (i, j).

Note that if edge e has enough spare capacity to carry the demand for all failures on the service path between i and j, the small weight on e encourages its inclusion in the bypass from i to j. Such an inclusion means that capacity previously assigned to e for other failures can also be used at zero extra cost to handle failures on the service path between i and j. The length is made a small positive value rather than zero so that the algorithm will reuse capacity on shorter rather than longer paths.

The running time of LOCAL is determined by the number and length of input service routes. Let ℓ_i denote the length of service route i. Then $O(\sum_i \ell_i^2)$ shortest-path computations must be performed. These can each be accomplished in $O(m + n \log n)$ time with an implementation of Dijkstra's algorithm using Fibonnaci heaps [7]. In practice, however, we use a version of Dijkstra's algorithm for dynamic maintenance of shortest paths, recomputing only as much as necessary as edge weights change. The worst-case performance remains the same, but in practice the running time is substantially decreased.

5 A Linear Program Lower Bound

This section describes a linear program that gives a lower bound on the cost of the optimal restoration strategy. The linear program is much smaller than those used by the path-generation algorithm of Section 3, and so can be used to analyze networks that are too large to be planned using path-generation.

To describe the lower bound computation, we first describe a restoration strategy in general terms. A restoration strategy specifies what paths are to be used when various edges fail. For each edge f, the strategy contains a collection, R_f, of paths. No path in R_f can contain f, and there must be a path in R_f for each demand. A basic routing can be viewed as the collection of paths, $R_\emptyset$, to be used in the case of the "null failure."

Given a fixed basic routing, $R_\emptyset$, we construct a linear program that computes a lower bound on the cost of *any* restoration strategy that uses $R_\emptyset$.

The linear program formulation is similar to that for multicommodity flow. There is a commodity for each edge f in the network, which represents the amount of traffic that must be rerouted when edge f fails. The source of the commodity is one endpoint of f and the sink the other endpoint. The amount of commodity that must be routed is set to the total traffic crossing f in the basic routing. Standard network flow constraints are added to generate a flow between source and sink (except of course that commodity f is not allowed to use edge f). We define a non-negative variable cap(e) to represent the restoration capacity of edge e, and for each failure f add the constraint:

$$\text{cap}(e) \geq \text{flow}_f(e) - \text{shared}(e, f),$$

where flow$_f(e)$ is the flow of commodity f over e and shared(e, f) is the amount of traffic that flows over both e and f in the basic routing. The cost of a solution is the sum over edges of cap(e) times the cost of e.

The objective of the linear program is to minimize the total cost. Effectively, it does this by reusing capacity between failures. To see why it gives a lower bound on the cost of any restoration strategy, consider a collection of sets R_f of paths as described above.

We show how to construct a feasible solution to the linear program whose cost is no greater than the cost of the sets R_f. Consider a single demand D with service path P crossing edge f, and let P_f be the path for D in R_f. For the commodity for edge f in the linear program, we route the contribution from D as follows. From the endpoints of f the commodity follows P in each direction until hitting a node on P_f. Between those two nodes the commodity flows along P_f. Only the latter part of the route contributes to the cap$()$ variables, because along the rest of the route, D is contributing equally to both flow$_f()$ and shared$(., f)$.

The solution to the linear program is only a lower bound on required restoration capacity, because the aggregation of demands for each failure allows feasible solutions that do not correspond to any set of restoration routes, and because the solution may be fractional.

The linear program has $O(m)$ commodities, each of which has $O(m)$ flow variables. At each node there is a flow-conservation constraint for each commodity. Hence the LP has $O(m^2)$ variables and $O(mn)$ constraints. Note that we could alternatively use a path-generation formulation of the the lower bound. Such a formulation may be faster in practice than the LP presented here. (It would be smaller than the column-generation formulation of Section 3, since the number of commodities here is m, whereas there it is the number of distinct demand routes, which is much larger.)

6 Data Sets

The test data sets have three components; networks, demand matrices, and cost functions. Real-life telecommunications networks are typically sparse and near-planar, and our test networks reflect these characteristics. The test demand matrices are either derived from demand sets of a major U.S. telecommunications carrier, or are generated randomly using a probability density function intended to mimic the characteristics of the real data. Finally, the cost function per unit capacity on an edge is of the form $a + bx$, where x is the edge length. For some experiments the constants a and b are chosen to match current real equipment costs, and for others $a = 1$ and $b = 0$.

6.1 Networks

The *fiber* networks are derived from telecommunications fiber optic routes in the United States. There are four networks, all two-connected, sized as follows:

data set	number of nodes	number of links
A	58	92
B	78	121
C	102	165
D	452	577

The cost per unit capacity on an edge is 300 plus the edge length in miles. The constant 300 was chosen to represent the relative cost of equipment placed only at the end of a link, such as switch ports, compared to equipment which is placed at regular intervals along a link, such as optical amplifiers. Edge lengths ranged from 2 to 992

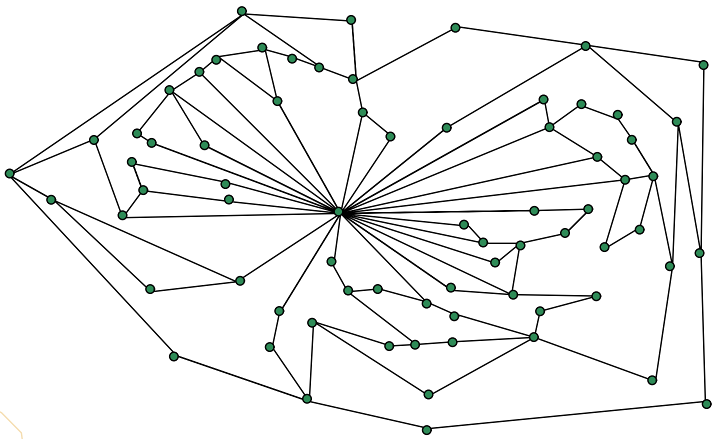


Figure 4: The network LA100

(measured in miles) for the three smaller networks, and 0.1 to 370 miles for network D, giving edge costs in the range 302 to 1292 for the smaller networks and 300 to 670 for network D.

The *Los Angeles* (LA) networks are derived from a detailed street map of the greater Los Angeles area. The input map contained 149727 nodes and 214493 edges. The test networks were generated from the input by a series of planarity-preserving edge contractions. Edges were selected to be contracted in increasing order by length.

data set	number of nodes	number of links
LA100	72	121
LA2000	487	982
LA4000	1050	1974

The LA networks have a small number of very high degree nodes; see for example the network LA100, pictured in Figure 4. The average edge length is small, about 1.5 miles. A unit cost was assigned to each edge for the test runs.

6.2 Demand sets

For each of the fiber networks, an actual matrix of traffic forecasts between approximately 600 US cities and towns was first mapped to the fiber nodes, then concentrated into large units of bandwidth, resulting in problems of moderate size, with unit demand sizes. Two different forecasts were used for each of A, B and C, resulting in two demand sets per network. These demand sets form the basis of the comparison between LOCAL and column generation. Their details are as follows:

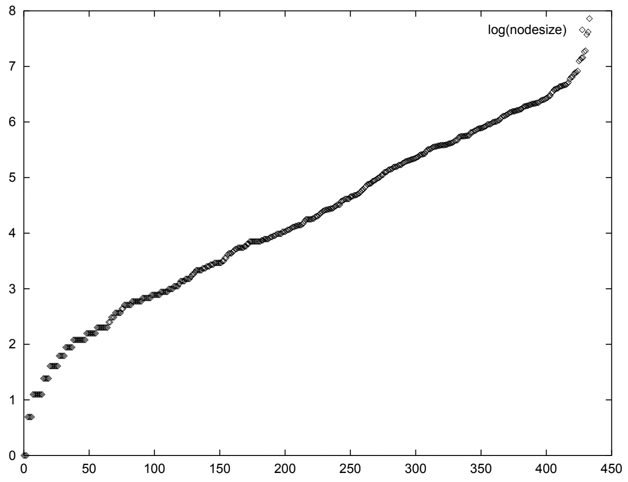


Figure 5: Log of the number of demands originating at each of the 452 nodes of network D , in increasing order.

demand set	number of demands
A 1	178
A 2	258
B 1	260
B 2	465
C 1	355
C 2	679

Larger demand sets were generated to examine the scaling ability of LOCAL to further compare LOCAL to the linear program lower bound, and to test the effectiveness of the lower bound on larger instances. For the D network, a traffic forecast was first mapped to the nodes of D, then aggregated at 3 different levels of granularity, giving the demand sets `D.T1`, `D.T3` and `D.OC48`.

For the `LA` runs, traffic was randomly generated as follows. Examination of the real demand sets above showed that the number of demands originating at nodes followed a nearly exponential curve; an example distribution for network D network is shown in Figure 5.

To make random demand sets, the nodes were permuted randomly, and then the ith node in the permuted order was assigned weight $e^{i/c}$ for a fixed constant c. Unit demands were then generated between pairs of nodes chosen randomly according to their weights.

To summarize, the larger demand sets were as follows:

Source	Protection cost			Protection		Total cost	
	Local	Column Gen.	Lower bound	CG to LB	Local to LB	CG to LB	Local to LB
A 1	73.99	65.43	64.77	1.0	14.2	0.4	5.6
A 2	69.66	61.42	60.79	1.0	14.6	0.4	5.5
B 1	67.90	61.48	61.07	0.7	11.2	0.3	4.2
B 2	63.37	55.63	55.44	0.3	14.3	0.1	5.1
C 1	66.72	57.37	57.14	0.4	16.8	0.1	6.1
C 2	61.55	53.34	53.08	0.5	15.9	0.2	5.5

Table 2: Results of the fiber runs.

Demand set	Demands	non-0 demand pairs	Demand set	Demands	non-0 demand pairs
D.T1	522321	19979	LA2000.small1	32565	16361
D.T3	44143	7032	LA2000.small2	32604	16218
D.OC48	2121	1256	LA2000.large1	527890	30885
LA100.small1	683	1565	LA2000.large2	527793	30800
LA100.small2	660	1567	LA4000.small1	70387	46711
LA100.large1	876	4736	LA4000.small2	70395	46607
LA100.large2	853	4767	LA4000.large1	1139367	112567

7 Computational Results

The comparison between network designs is done both in terms of restoration cost (relevant when working with an existing network for which restoration cost must be minimized [3]) and total cost (restoration plus service cost). The latter measure is useful in situations when restoration efficiency is only of interest to the extent that it affects the entire network cost, for example when doing studies to choose between different options for a network architecture (as in reference [4]).

The results on the fiber networks are shown in Table 2. For each of LOCAL, column generation and the lower bound, the table shows the cost of the extra capacity required for restoration, expressed as a percentage of the total cost of the service capacity. The next four columns show percentage differences between LOCAL and column generation, both for restoration cost only and for total cost. The lower bound linear program and column generation are extremely close, while LOCAL has 10% to 16% higher restoration cost than column generation. When considering total network cost, the difference between LOCAL and column generation is between 4% and 6%.

The column generation runs were made using a single processor of a SUN Ultrasparc Enterprise 3000 machine. There is some opportunity in the system for runtime reductions. The column generation runs used CPLEX 6.0, while the lower bounds used CPLEX 5.0. The LOCAL and lower bound runs were done on an SGI Challenge machine using a single MIPS R10000 processor and R10010 FPU, with 1 Mbyte secondary cache. All machines had approximately 2 gigabytes of memory. The run times

Source	Col gen runtime	Lower bound runtime	Local runtime
A 1	38 minutes	30 seconds	5 seconds
A 2	53 minutes	29 seconds	5 seconds
B 1	2 hours, 6 minutes	62 seconds	8 seconds
B 2	4 hours, 8 minutes	64 seconds	12 seconds
C 1	23 hours, 43 minutes	28 minutes, 15 seconds	16 seconds
C 2	59 hours, 33 minutes	27 minutes, 33 seconds	24 seconds

Table 3: Running times of the fiber runs.

Source	Protection cost			Protection		Total cost	
	Local	Column Gen.	Lower bound	CG to LB	Local to LB	CG to LB	Local to LB
A 1	78.11	66.17	64.77	2.2	20.6	0.9	8.1
A 2	73.72	63.81	60.79	5.0	21.3	1.9	8.0
B 1	71.51	61.91	61.07	1.4	17.1	0.5	6.5
B 2	65.51	55.94	55.44	0.9	18.2	0.3	6.5

Table 4: Results of the fiber runs with path-based restoration.

are given in Table 3. Running times on the SGI Challenge have been normalized into running time on the SUN Ultra, using comparative timing data obtained by running a benchmark on both machines.

Table 4 shows the results of the path-based restoration runs. The protection cost of LOCAL is 15% to 18% more than column generation, and when considering total network cost, the difference is 6% to 7%.

When the lower bound LP is run, it generates capacity values for each edge in the network. These values can then be used as input to LOCAL, by initializing the spare capacity on each edge to the values from the lower bound LP. Then LOCAL is run as normal, and it selects restoration paths using the spare capacity values to adjust costs.

Source	Local prot'n unprimed	Local prot'n primed	Local protn to LB protn unprimed	Local protn to LB protn primed
A 1	73.99	72.94	14.2	12.6
A 2	69.66	67.50	14.6	11.0
B 1	67.90	66.81	11.2	9.4
B 2	63.37	64.23	14.3	15.9
C 1	66.72	64.32	16.8	12.6
C 2	61.55	60.25	15.9	13.5

Table 5: Results of the fiber runs when LOCAL is primed with lower bound capacities.

Source	Protection cost		Local to LB		Time (s)	
	Local	Lower Bnd	protection	total cost	Local	LB
D.T1	70.65	61.63	14.6	5.6	27434	5000
D.T1.agg	71.63	61.63	16.2	6.2	2641	5000
D.T3	69.51	60.70	14.5	5.5	1670	4546
D.T3.agg	69.19	60.70	14.0	5.3	598	4546
D.OC48	75.23	65.94	14.1	5.6	65	4412
LA100.small1	69.07	65.48	5.5	2.2	4	54
LA100.small1.agg	68.86	65.48	5.2	2.0	2	54
LA100.small2	93.60	84.81	10.4	4.8	5	66
LA100.small2.agg	94.41	84.81	11.3	5.2	2	66
LA100.large1	62.79	59.10	6.2	2.3	15	61
LA100.large1.agg	64.02	59.10	8.3	3.1	3	61
LA100.large2	79.08	73.93	7.0	3.0	16	97
LA100.large2.agg	78.70	73.93	6.4	2.7	2	97
LA2000.small1.agg	74.38				2969	
LA2000.small2.agg	70.70				2685	
LA2000.large1.agg	77.82				3905	
LA2000.large2.agg	72.04				2645	
LA4000.small1.agg	69.11				27820	
LA4000.small2.agg	69.30				22456	
LA4000.large1.agg	70.72				53631	

Table 6: Results of the large runs.

This has a beneficial effect, as is shown by table 5. "Priming" the spare capacity in this manner generally lowers the differential between LOCAL and the lower bound by 2 to 4 percent, although occasionally things get worse, as in case B2.

Table 6 shows the results of the large runs. Blanks correspond to instances where the problem size is too large for the lower bound linear program to complete. Times are measured in seconds, and are from runs on an SGI Challenge machine as described above. For all the runs ending in the suffix ".agg", traffic was aggregated for the LOCAL runs. All the demands with the same service route between a pair of nodes were aggregated into a single demand, which LOCAL then restored as a single entity. This reduces LOCAL's flexibility in choosing restoration routes, so causes an increase in restoration cost, but the benefit is much faster running time. Matching the results on the smaller networks, the overhead for restoration is 5% to 16% higher for LOCAL than the lower bound, and total network cost is 2% to 8% higher. Aggregating demands increases the restoration cost by at most 2%.

8 Analysis

The results presented above show a significant cost differential between column generation and LOCAL. This suggests that when final decisions are to be made about building spare capacity in a moderate-size network, it is worthwhile to spend the effort to do so using column generation. However, the difference is small enough that it is safe to use LOCAL in network design studies and to build capacity for networks that are too large for column generation. The results are consistent across many sizes of networks and demand sets, which suggests that it is safe to extrapolate the results to problems too large to run the lower bound.

When path-based restoration is required, the restoration cost for both LOCAL and column generation is close to the cost for the non-path-based runs, from which we can derive the rather surprising conclusion that for these networks and demand sets, there is only a very small penalty for using path-based rather than general restoration.

The lower bound LP consistently produces bounds that are very close to the upper bounds produced by column generation. This suggests that the lower bound can be used as a very accurate tool for evaluating the quality of restoration designs for large networks. It also suggests a novel algorithm for restoration capacity design: run the lower bound LP to derive a set of link capacities, then test each link failure to determine whether all the traffic can be restored, and if not, augment the capacities accordingly. The findings of this paper suggest that very few augments will be necessary. We leave the investigation of this approach as a topic for future study.

9 Acknowledgements

We would like to thank Bruce Cortez for stimulating this study, Dave Houck for extensive work in developing the column generation approach and models, Bob Doverspike for advice on telecommunication network planning, and Carsten Lund and Nick Reingold for assistance in acquiring data sets used in this study.

References

[1] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1983.

[2] S. Cwilich, M. Deng, D. Houck, and D. Lynch. Applications and extensions in restoration network design. Technical report, AT&T Labs, 1999.

[3] S. Cwilich, M. Deng, D. Houck, and D. Lynch. An lp-based approach to restoration network design. In *Submitted to ITC16, International Teletraffic Congress*, 1999.

[4] B. Doverspike, S. Phillips, and J. Westbrook. Comparison of transport network architectures. In *Submitted to ITC16, International Teletraffic Congress*, 1999.

[5] K. Eswaran and R. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–665, 1976.

[6] A. Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM J. DISC. MATH.*, 5(1):25–53, 1992.

[7] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[8] J. Kleinberg. *Approximation algorithms for disjoint paths problems*. PhD Thesis, Department of EECS, MIT, 1996.

[9] S. Phillips and J. Westbrook. A network capacity and restoration planning toolkit. Technical Report HA6171000-980910-04TM, AT&T Labs, 1998.

Computing the $n \times m$ Shortest Paths Efficiently

Tetsuo Shibuya

IBM Research Division, Tokyo Research Laboratory,
1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan.
tshibuya@trl.ibm.co.jp

Abstract. Computation of all the shortest paths between multiple sources and multiple destinations on various networks is required in many problems, such as the traveling salesperson problem (TSP) and the vehicle routing problem (VRP). This paper proposes new algorithms that compute the set of shortest paths efficiently by using the A* algorithm. The efficiency and properties of these algorithms are examined by using the results of experiments on an actual road network.

1 Introduction

Computation of all the shortest paths between multiple sources and multiple destinations on various networks is required in many problems, such as the traveling salesperson problem (TSP), the vehicle routing problem (VRP), the warehouse location problem (WLP), and the quadratic assignment problem (QAP). Accordingly, a function for performing such computation is required in geographical information systems (GISs), logistics tools, and so on. There are many fast heuristic algorithms for solving such problems as TSP, and the computation time needed to find all the shortest paths sometimes occupies a large percentage of the total computation time. A more efficient way of computing the set of shortest paths is therefore desired.

The Dijkstra method [6] is the most traditional and widely-used algorithm for this kind of problem. It can compute the shortest paths from one source to n destinations in $O(|E| + |V| \log(|V|)$ time on a directed graph $G = (V, E)$ with no negative edges [7]. Note that, assuming integer edge lengths stored in words or in one word, this bound can be improved [4,8,17,18,20]. For a long time, this algorithm has been believed to be the best for computing all the shortest paths between two sets of vertices, especially on sparse graphs like actual road networks. We call this problem the $n \times m$-pairs shortest paths problem or simply the $n \times m$ shortest paths problem. Using the Dijkstra method, it takes $O(\min(n, m) \cdot (|E| + |V| \log(|V|))$ time to obtain all the shortest paths between n sources and m destinations.

On the other hand, much work has been done on improving its efficiency in solving the 2-terminal shortest path problem. The most famous example is the A* algorithm [1,5,9,10,14,15,19], which improves the efficiency of the Dijkstra algorithm using a heuristic estimator. Another famous technique is the bidirectional search algorithm [3,11,12,13,16], which searches from both the source and the destination. But these techniques have been believed to be inapplicable to the $n \times m$ shortest paths problem.

This paper proposes two new algorithms based on the A* algorithm for solving the $n \times m$ shortest paths problem, and examines the efficiency and properties of these algorithms by using the result of experiments on an actual digital road network in the Kanto area of Japan (around Tokyo). We also discuss how to apply these algorithms to dynamic networks.

One of the algorithms uses an estimator based on those used for the 2-terminal A* algorithm. In the case of digital road networks, we can use an estimator based on Euclidean distance, and computing the estimates takes $O(|V| \log \max(n, m))$ time. This algorithm does not improve the computational bound of the Dijkstra method. But according to the experiments, this algorithm reduces the time for loading data, and in some cases, the total computing time.

The other algorithm uses the concept of the network Voronoi diagram (which is a division of V similar to the Voronoi diagram). Although it is based on the same principle as the other one, it is very closely related to the bidirectional search method, so we call it the bidirectional-method-based A* algorithm, or simply the bidirectional method. This algorithm can be used on networks which does not have appropriate estimator for 2-terminal A* algorithm (Euclidian distance, etc), which is also the feature of the bidirectional method for 2-terminal problems. It takes $O(|E| + |V| \log(|V|))$ time to compute the estimates; this is also the time taken to construct the network Voronoi diagram. This algorithm also does not improve the computational bound of the Dijkstra method. But according to the experiments, this algorithm reduces the computing time by 30%-70% in most cases, compared with the Dijkstra method.

2 Preliminaries

2.1 The Dijkstra Method

The Dijkstra method [6] is the most basic algorithm for the shortest path problem. The original algorithm computes the shortest paths from one source to all the other vertices in the graph, but it can be easily modified for the problem of computing the shortest paths from one source to several specified other vertices.

Let $G = (V, E)$ be a directed graph with no negative edges, $s \in V$ be the source, $T = \{t_1, t_2, \ldots, t_m\}$ be the set of destinations, and $l(v, w)$ be the length of an edge $(v, w) \in E$. Then the outline of the algorithm is as follows:

Algorithm 1

1. *Let U be an empty set, and the potential $p(v)$ be $+\infty$ for each vertex $v \in V$ except for $p(s) = 0$.*
2. *Add to U the vertex v_0 that has the smallest potential in $V - U$. If $T \subseteq U$, halt.*
3. *For each vertex $v \in V$ such that $(v_0, v) \in E$, if $p(v_0) + l(v_0, v) < p(v)$, update $p(v)$ with $p(v_0) + l(v_0, v)$ and let previous(v) be v_0.*
4. *Goto step 2.*

The s-t_i shortest path is obtained by tracing $previous(v)$ from t_i to s, and its length is stored in $p(t_i)$. Note that the overall required time for this algorithm is $O(|E| + |V| \log(|V|)$ [7].

The most traditional and widely used method for solving the $n \times m$ shortest paths problem is doing this procedure n times. Note that it is better to do m times the same kind of procedure in which we search from the destinations if $m \leq n$. Thus the required time for this problem is $O(\min(n, m) \cdot (|E| + |V| \log(|V|))$. This paper focuses on how to improve this method.

2.2 The A* Algorithm

The A algorithm, an extension of the Dijkstra method, is a heuristic algorithm for the 2-terminal shortest path problem. It uses a heuristic estimator for the shortest path length from every vertex in the graph to the destination. Let $h(u, v)$ be the estimate for the u-v shortest path length, and $h^*(u, v)$ be the actual u-v shortest path length. Then the algorithm is as follows, letting t be the destination.

Algorithm 2

```
1. Let U be an empty set, and let the potential $p(v)$ be $+\infty$ for each vertex $v \in V$ except for $p(s) = 0$.
2. Add to U the vertex $v_0$ that has the smallest value of $p(v) + h(v_0, t)$ in $V - U$. If $v_0 = t$, halt.
3. For each vertex $v \in V$ such that $(v_0, v) \in E$, if $p(v_0) + l(v_0, v) < p(v)$, update $p(v)$ with $p(v_0) + l(v_0, v)$, let $previous(v)$ be $v_0$, and remove $v$ from U if $v \in U$.
4. Goto step 2.
```

If $h(v, t)$ satisfies the following constraint, which means $h(v, t)$ is a lower bound of $h^*(v, t)$, the obtained path is guaranteed to be the optimal shortest path and the algorithm is called the A* algorithm [1,5,9,10,14,15,19].

$$\forall v \in V \quad h(v, t) \leq h^*(v, t) \tag{1}$$

Note that if $h(v, t)$ equals $h^*(v, t)$ for all $v \in V$, the A* algorithm is known to search only the edges on the s-t shortest path. Moreover, the removal of vertices from U in step 3 can be omitted if the estimator satisfies the following constraint, which is called monotone restriction:

$$\forall (u, v) \in E \quad l(u, v) + h(v, t) \geq h(u, t) \tag{2}$$

An estimator under this constraint is called a dual feasible estimator. For example, the Euclidean distance on a road network is a dual feasible estimator. Obviously, $h^*(v, t)$ also satisfies the above constraint. Note that the number of vertices searched in this case is always not larger than the number searched by the Dijkstra method.

2.3 The Bidirectional Method

The bidirectional method [3,11,12,13,16] is also considered for the 2-terminal shortest path problem. It does not require any heuristic estimator, but can reduce the number of searched vertices in many cases. In this algorithm, the searches are done not only from the source but also from the destination. The algorithm is as follows:

Algorithm 3

1. *Let U and W be empty sets, and let the potentials $p_s(v)$ and $p_t(v)$ be $+\infty$ for each vertex $v \in V$ except for $p_s(s) = 0$ and $p_t(t) = 0$.*
2. *Add to U the vertex v_0 that has the smallest potential $p_s(v)$ in $V - U$. If $v_0 \in W$, goto step 7.*
3. *For each vertex $v \in V$ such that $(v_0, v) \in E$, if $p_s(v_0) + l(v_0, v) < p_s(v)$, update $p_s(v)$ with $p_s(v_0) + l(v_0, v)$ and let $previous_s(v)$ be v_0.*
4. *Add to W the vertex v_0 that has the smallest potential $p_t(v)$ in $V - W$. If $v_0 \in U$, goto step 7.*
5. *For each vertex $v \in V$ such that $(v, v_0) \in E$, if $p_t(v_0) + l(v, v_0) < p_t(v)$, update $p_t(v)$ with $p_t(v_0) + l(v, v_0)$ and let $previous_t(v)$ be v_0.*
6. *Goto step 2.*
7. *Find the edge $(u_0, w_0) \in E$ that has the smallest value of $p_s(u) + l(u, w) + p_t(w)$. The s-t shortest path consists of the s-u_0 shortest path, the edge (u_0, w_0), and the w_0-t shortest path.*

3 New Approaches for Computing the $n \times m$ Shortest Paths

3.1 The Basic Principle

We discuss in this section how to compute all the shortest paths between two sets of vertices efficiently. Let $S = \{s_1, s_2, \ldots, s_n\}$ be the set of sources, and $T = \{t_1, t_2, \ldots, t_m\}$ be the set of destinations. It does not matter if some of the vertices are in both S and T.

The basic idea of our approach is to find an estimator that can be used in every search between two vertices s_i and t_j. Let $h(v, t_i)$ be an estimator for t_i. Then consider the following estimator:

$$h(v) = \min_i h(v, t_i) \tag{3}$$

Can this estimator be used in the search from any source to any destination? To answer this question, we obtain the following theorems:

Theorem 1. *The estimator h as in expression (3) can be used as an A^* estimator for any t_i, if $h(v, t_j)$ is a lower bound of $h^*(v, t_j)$ for each j.*

Proof. Consider the case of searching the shortest path to t_k.

$$h(v) = \min_i h(v, t_i) \le h(v, t_k) \le h^*(v, t_k) \tag{4}$$

This inequality means that $h(v)$ is also a lower bound. Thus we can use it for an A* estimator for any t_i.

Theorem 2. *The estimator h as in expression (3) is a dual feasible estimator for any t_i if, for any j, $h(v, t_j)$ is a dual feasible estimator for t_j.*

Proof. Consider the dual feasibility around some arbitrary edge (u, v). There must be some k such that $h(v) = h(v, t_k)$, and the following inequality is derived from the dual feasibility of $h(v, t_k)$:

$$l(u, v) + h(v, t_k) \ge h(u, t_k) \tag{5}$$

Thus we can obtain the following inequality:

$$l(u, v) + h(v) = l(u, v) + h(v, t_k) \ge h(u, t_k) \ge \min_i h(u, t_i) = h(u) \tag{6}$$

This means that $h(v)$ is a dual feasible estimator.

Using this dual feasible estimator, we can solve the $n \times m$ shortest paths problem as follows:

Algorithm 4 *For each i, do the following:*

```
1. Let $U$ be an empty set, and let the potential $p(v)$ be $+\infty$ for each vertex
   $v \in V$ except for $p(s_i) = 0$.
2. Add to $U$ the vertex $v_0$ that has the smallest value of $p(v) + h(v)$ in $V - U$.
   If $T \subseteq U$, halt.
3. For each vertex $v \in V$ such that $(v_0, v) \in E$, if $p(v_0) + l(v_0, v) < p(v)$, update
   $p(v)$ with $p(v_0) + l(v_0, v)$ and let previous$(v)$ be $v_0$.
4. Goto step 2.
```

3.2 Techniques for a Road Network

For the 2-terminal problem in a road network, we often use the Euclidean distance $d(v, w)$ as a dual feasible estimator of the v-w shortest path length. Thus we can consider the following estimator for the $n \times m$ shortest paths problem:

$$h(v) = \min_i d(v, t_i) \tag{7}$$

This estimate is the Euclidean distance to the nearest vertex in the destination set T.

k-d tree [2] is a very efficient data structure for coping with this nearest neighbor problem especially in the two dimensional space. In k-dimensional Euclidean space, the time for building a *k-d* tree for m points is $O(m \log m)$, and the time for querying the nearest neighbor of some other point is $O(\log m)$.

We have to compute the nearest neighbor of a particular point only once, because we use the same estimator in each search from each source. Thus, the extra time needed to compute all the required estimates is $O(|V| \log m)$, because we can ignore the time $O(m \log m)$ for building the tree. We call this algorithm the Euclidean-distance-based A* algorithm.

3.3 The Bidirectional Method

In this subsection, we discuss how we can solve the $n \times m$ shortest paths problem efficiently even if we do not have any appropriate estimator, as when we use the bidirectional method in the 2-terminal case.

Consider the following estimator in the 2-terminal shortest path problem:

$$h(v,t) = \min(c, h^*(v,t)), \quad c: \textit{constant} \tag{8}$$

The following corollary of Theorem 2 shows that this estimator is dual feasible.

Corollary 1. *The estimator $h'(v,t) = \min(c, h(v,t))$ is dual feasible if $h(v,t)$ is a dual feasible estimator and c is a constant.*

How does the algorithm behave if we use this estimator? First, we must search from the destination t to obtain the estimates until we find some vertex from which the shortest path length to the destination is larger than c. Let T' be the set of vertices covered by this backward search. The search will be done from the source s until it encounters some vertex in T'. Let S' be the set of vertices searched by this forward search at the time, and let E' be the set of edges $(u,v) \in E$ such that $u \in S'$ and $v \in T'$. Let v_0 be the vertex such that $e = (u_0, v_0) \in E'$ for some $u_0 \in S'$ and the s-t shortest path includes the vertex v_0. After the encounter, the algorithm searches only edges in E and on the v_0-t shortest path. Thus, its behavior is very similar to that of the bidirectional algorithm (Algorithm 3). If we let c be the largest value of $p_t(v)$ in the bidirectional method except for $+\infty$, the region searched by this algorithm is almost the same as that searched by the bidirectional method. Thus, the bidirectional method can be said to be a variation of the A* algorithm.

On this assumption, the bidirectional method can be extended for the $n \times m$ shortest paths problem to the A* algorithm, which uses the following estimator:

$$h(v) = \min_i h^*(v, t_i) \tag{9}$$

This estimator gives the shortest path length to the set of destinations. According to the theorems in the last subsection, it is a dual feasible estimator.

We can obtain this estimator by a variation of the backward Dijkstra method as follows. Let U be the set of vertices for which we want to know the value $h(v)$.

Algorithm 5

```
1. Let $W$ be an empty set. Let the potential $p(v)$ be $+\infty$ for each vertex $v \in V - T$, and $p(v)$ be $0$ for each vertex $v \in T$.
2. Add to $W$ the vertex $v_0$ that has the smallest potential in $V - W$, and set $h(v_0)$ with $p(v_0)$. If $U \subseteq W$, halt.
3. For each vertex $v \in V$ such that $(v, v_0) \in E$, update $p(v)$ with $p(v_0) + l(v, v_0)$ if $p(v_0) + l(v, v_0) < p(v)$.
4. Goto step 2.
```

Thus, the extra time taken to compute estimates as in (9) for all the nodes is $O(|E| + |V|\log(|V|)$, which is the same as the time taken by the ordinary Dijkstra method. Note that we here construct the network Voronoi diagram of the destination set T. The network Voronoi diagram of T is a subdivision of V to $\{V_i\}$ where the shortest path length from $v \in V_i$ to t_i is not larger than those to any other vertices in T. Note also that we do not have to compute these estimates twice or more, because we use the same estimator in each search from each source. We call this algorithm the bidirectional-method-based A* algorithm, or simply the bidirectional method.

This backward search should be performed as required at the same time of the forward search. In this way, we can, in most cases, reduce the number of vertices covered by the backward search. But if there are vertices from which there is no path to any of the destinations, the backward search may continue throughout the graph without stopping. Thus, if the graph has such vertices and is very large compared with the regions searched by the forward searches, we should modify the estimator as follows, using some appropriate constant c:

$$h(v) = \min(c, \min_i h^*(v, t_i)) \tag{10}$$

According to corollary 1, this is also a dual feasible estimator. To compute this kind of estimate for all the vertices, we only have to let $p(v)$ be c in step 1 of the algorithm 5. Note that this estimator is more similar to the estimator in expression (8) than that in expression (9). If we use this estimator, we do not have to search the whole graph to obtain this estimate for any vertex, but it may be difficult to decide an appropriate c. A good way to set c is to set some appropriate value larger than $\max_i \min_j h^*(s_i, t_j)$, which means that we do not set c until the estimates for all the sources are computed.

3.4 Computation on Dynamic Networks

Computation of the $n \times m$ shortest paths on dynamic networks is also important. For example, an actual road network varies continuously because of traffic jams, road repairing, and so on.

Related to this, the heuristic estimators for the A* algorithm has the following properties. Let $G' = (V, E')$ be a modified graph of $G = (V, E)$ by increasing some of the edge lengths. Note that this modification includes deletion of edges: we only let these edge lengths $+\infty$.

Theorem 3. *If a heuristic estimator h satisfies inequality (1) on graph G, it also does on graph G'.*

Proof. Let $h^*_G(v, w)$ be the shortest path length on graph G. It is obvious that the shortest path on G' is longer than that on G. Thus if the estimator satisfies inequality (1) on G, then the following inequality is satisfied:

$$\forall v \in V \quad h(v, t) \leq h^*_G(v, t) \leq h^*_{G'}(v, t) \tag{11}$$

It means h satisfies the inequality on G' too.

Theorem 4. *If a heuristic estimator h satisfies inequality (2) on graph G, it also does on graph G'.*

Proof. Let $l'(v,w)$ be the length of edge (v,w) on graph G'. According to the definition of G', $l'(v,w)$ is not smaller than $l(v,w)$. Thus the following inequality is satisfied if h satisfies inequality (2) on G:

$$\forall (u,v) \in E \;\; l'(u,v) + h(v,t) \geq l(u,v) + h(v,t) \geq h(u,t) \tag{12}$$

It means h satisfies the inequality on G' too.

According to these theorems, we can use the same estimator on the dynamic graph and do not have to recompute the estimates, if the change is only the increase of the edge lengths or deletion of edges. All the estimators we proposed in this paper satisfy inequalities (1) and (2), and we can efficiently use them on such dynamic graphs.

4 Computational Experiments on a Road Network

In this section, we investigate the efficiency of our algorithms by using actual digital road network data. The network covers a square region of 200km×200km in the Kanto area in Japan, which contains several large cities such as Tokyo and Yokohama. There are 387,199 vertices and 782,073 edges in the network. We did all the experiments on an IBM RS/6000 Model 7015-990 with 512M bytes of memory. We use the time taken to traverse an edge as the length of that edge. Thus we compute the Euclidean-distance-based estimator using the value of the Euclidean distance divided by the maximum speed. In this section, we call the Euclidean-distance-based A* algorithm simply "the A* algorithm," and the bidirectional-method-based A* algorithm "the bidirectional method."

In our algorithms, we compute estimates only once at most for one node regardless of the number of sources and destinations. Therefore analyzing searching time without the time for computing estimates is very important. Figures 1 and 2 show the ratios of number of nodes searched by our algorithms to that by the Dijkstra method. In the experiments, we first chose randomly 1000 points within a circle whose diameter is 100km, for starting points of searches. For the destination set, we also chose randomly (1) 5 / (2) 10 / (3) 20 / (4) 50 / (5) 100 / (6) 200 points within a circle whose diameter is 20km and whose center is same as that for the starting points. Note that the center is located around Shibuya, Tokyo. Then we searched from these 1000 points to the destination sets by various algorithms, and we plotted the ratio of the number of nodes searched by our algorithms to that by the Dijkstra method, and the distance from the center to the starting point.

As for the A* algorithm, the ratio is about 50-80% in the cases the number of destinations is small, and 60-90% in the cases the number of destinations is large, but difference between them is not so remarkable. The ratio becomes better if the starting point goes away from the center up to about 20km, twice as the radius of distribution of the destination set. As for the points whose distance is

Table 1. Time (sec) for computing estimates for all nodes in the network.

#destinations	5	10	20	50	100	200
the A* algorithm	6.33	8.15	9.32	13.88	16.78	24.75
the bidirectional method	3.78	3.78	3.78	3.78	3.80	3.82

larger than 20km, the average ratio does not change remarkably as the distance grows up, but some have much better or much worse ratio than those nearer to the center.

As for the bidirectional method, the ratio is about 20-50% in the cases the number of destinations is small, and 40-70% in the cases the number of destinations is large. In this case, the number of destinations influences the ratio very much. On the other hand, the influence of the distance from the center is very similar to the case of the A* algorithm. But, in this case, the ratio decreases as the distance grows up to to about 30km, which is larger than that in the previous case.

The above analyses are done not on the actual computing time, but on the number of searched nodes, because the actual computing time is influenced by the method of implementation, the machine type, and so on. Figure 3 shows that the same experiment using the destination set (5), but it shows the ratio of time (which does not include the time for computing estimates). We can easily see that the actual computing time clearly reflects the number of searched nodes.

Table 1 shows the time for computing estimates for all 387, 199 nodes in the network for the same destination sets as in the previous experiments. Hence the actual time for computing those estimates must not be larger than this. In the case of the A* algorithm, this time increases as the number of destinations becomes larger. On the other hand, the number of destinations does not influences the time at all in the case of the bidirectional method. It is because the computation is always done by only one search of the Dijkstra method in the case of the bidirectional method.

According to the number of searched nodes, and the time for computing estimates, the bidirectional method seems far more better than the A* algorithm. Table 2 shows the results of the experiments in several actual cases. In the table, #Searched means the total number of vertices searched by all the n searches, #Loaded means the number of vertices loaded to memory, T_{total} means the total computing time (in seconds), and $T_{estimate}$ means the time taken to compute estimates. Note that T_{total} includes $T_{estimate}$. In case 1, we compute all the shortest paths among 50 points around Tokyo. The problem in case 2 is to compute the shortest paths between 20 sources and 150 destinations, both of which are distributed around Tokyo, while in case 3, the problem is to compute the shortest paths between 30 sources around Tokyo and 40 destinations around Yokohama. Note that situations of all of these cases are very common in real problems. Note also that situation like case 3 is very advantageous to our algorithms.

Table 2. Computation Results in Actual Cases

case	Method	#Searched	#Loaded	T_{total}	$T_{estimate}$
1 (50×50)	Dijkstra	5671181	232117	65.58	-
	A*	4793515	180913	58.50	5.38
	Bidirectional	3860605	247245	42.98	1.98
2 (20×150)	Dijkstra	2636279	215130	30.10	-
	A*	2219739	170147	31.20	6.12
	Bidirectional	1868263	228316	21.83	2.43
3 (30×40)	Dijkstra	4818566	193723	56.00	-
	A*	2901990	125212	35.13	4.06
	Bidirectional	1559049	135635	17.37	0.84

According to the table, the bidirectional method shows the best performance in all cases. It reduces the computing time by about 30% compared with the simple Dijkstra method in normal cases (1 and 2). If the sources and destinations are located in two distant clusters, as in case 3, the ratio becomes almost 70%, because the number of searched vertices is dramatically reduced. Note that both the original A* algorithm and the original bidirectional method reduce the computing time by 40% to 60% compared with the Dijkstra method in the 2-terminal case on such a road network [12]. The A* algorithm also reduces the number of searched vertices, but takes a long time to compute the estimates, even though we use a 2-d tree as in section 3.2. Thus it is not efficient, especially when the number of the destinations (sources if the searches are done from the destinations) is large, as in the case 2. However, it performs well compared with the Dijkstra method in situations like case 3, because the searching is done mainly in the direction of the destinations by using the Euclidean-distance-based estimator. Figure 4 shows the regions searched from the same source as in case 2. We can easily see that the bidirectional method searches the fewest vertices.

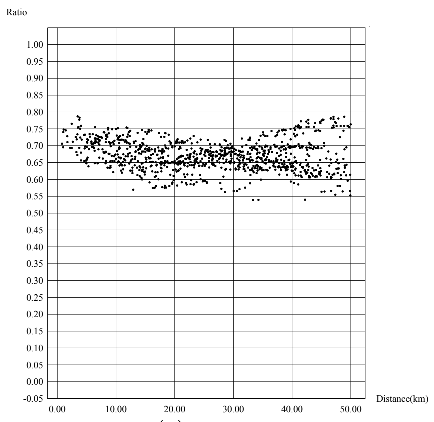
The A* algorithm is not fast as the bidirectional method on our system, but the experiments reveal that it may be useful on some other systems. Table 2 shows that the number of vertices loaded to memory is small if we use the A* algorithm. Figure 5 shows the region loaded to memory in case 2. We can easily see that the A* algorithm loads to memory the fewest vertices among the three algorithms. To compute the estimates, the bidirectional method must load more vertices to memory than the A* algorithm. This means that the A* algorithm is one of the choices if the data storage device on the system is very slow.

5 Concluding Remarks

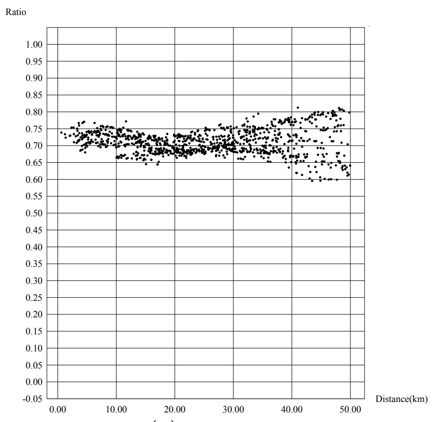
We have proposed new algorithms for the $n \times m$ shortest paths problem. We showed what kind of estimators for the A* algorithm could deal with this $n \times m$ shortest paths problem. As examples, we proposed two kinds of estimators, one based on Euclidean distance, and the other on the bidirectional method. We examined the efficiency of the algorithms using these estimators through experiments on an actual digital road network. The experiments revealed that the bidirectional-method-based A* algorithm is the best, and that it reduces the computing time by 30%-70% compared with the simple Dijkstra method. They also implied that the Euclidean-distance-based A* algorithm is useful on systems with very slow storage devices.

References

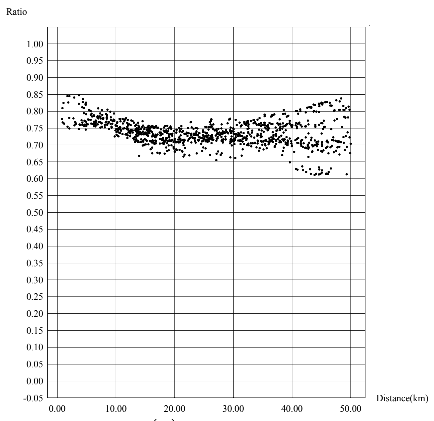
1. A. Barr and E. A. Feigenbaum, *Handbook of Artificial Intelligence*, William Kaufman, Inc., Los Altos, Calif., 1981.
2. J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Commun. ACM, vol. 18, no. 9*, 1975, pp. 509-517.
3. D. Champeaus, "Bidirectional Heuristic Search Again," *J. ACM, vol. 30*, 1983, pp.22-32.
4. B. V. Cherkassky, A. V. Goldberg, and C. Silverstein, "Buckets, heaps, lists and monotone priority queues," *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms,* 1997, pp. 83-92.
5. R. Dechter and J. Pearl, "Generalized Best-First Search Strategies and the Optimality of A*," *J. ACM, vol. 32, no. 3*, 1985, pp. 505-536.
6. E. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numerical Mathematics, vol. 1*, 1959, pp. 395-412.
7. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *J. ACM, vol. 34, no. 3*, 1987, pp. 596-615.
8. M. L. Fredman and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *J. Comp. Syst. Sc.* vol. 48, 1994, pp. 533-551.
9. D. Gelperin, "On the Optimality of A*," *Artif. Intell. vol. 8, no. 1*, 1977, pp. 69-76.
10. P. E. Hart, N. J. Nillson, and B. Rafael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Sys. Sci. and Cyb. SSC-4,* 1968, pp. 100-107.
11. T. Hiraga, Y. Koseki, Y. Kajitani, and A. Takahashi, "An Improved Bidirectional Search Algorithm for the 2 Terminal Shortest Path," *The 6th Karuizawa Workshop on Circuits and Systems,* 1993, pp. 249-254 (in Japanese).
12. T. Ikeda, M. Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, K. Tenmoku, and K. Mitoh, "A Fast Algorithm For Finding Better Routes By AI Search Techniques," *IEEE VNIS'94*, 1994, pp. 90-99.
13. M. Luby and P. Ragde, "A Bidirectional Shortest-Path Algorithm With Good Average-Case Behavior," *Proc. 12th International Colloquium on Automata, Languages and Programming, LNCS 194,* 1985, pp. 394-403.
14. N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.
15. N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, Calif., 1980.
16. I. Pohl, "Bi-Directional Search," *Machine Intelligence, vol. 6,* pp. 127-140, 1971.
17. R. Raman, "Priority queues: small monotone, and trans-dichotomous," *Proc. ESA'96, LNCS 1136,* 1996, pp. 121-137.
18. R. Raman, "A summary of shortest path results," Technical Report TR 96-13, Kings College, London, 1996.
19. Y. Shirai and J. Tsuji, "Artificial Intelligence," *Iwanami Course: Information Science, vol. 22,* Iwanami, Japan, 1982 (in Japanese).
20. M. Thorup, "On RAM priority queues," *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms,* 1996, pp. 59-67.



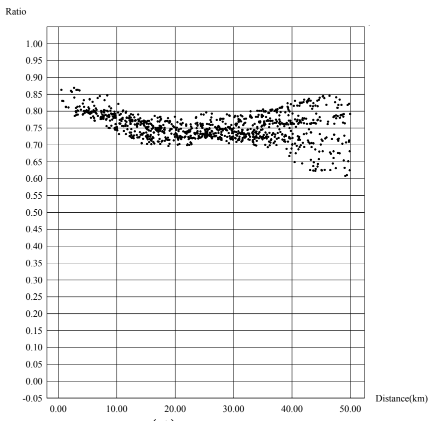
(1) 5 points



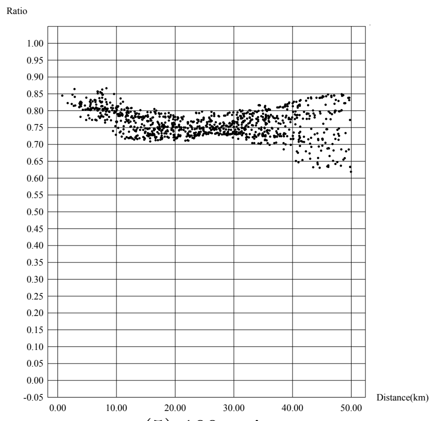
(2) 10 points



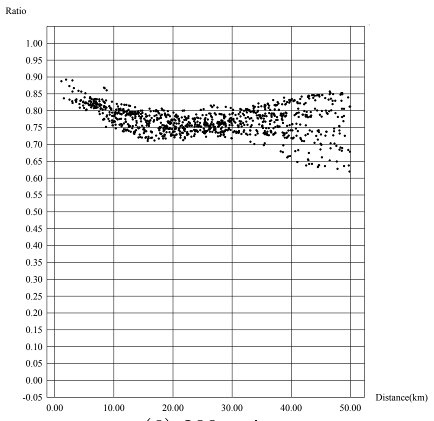
(3) 20 points



(4) 50 points

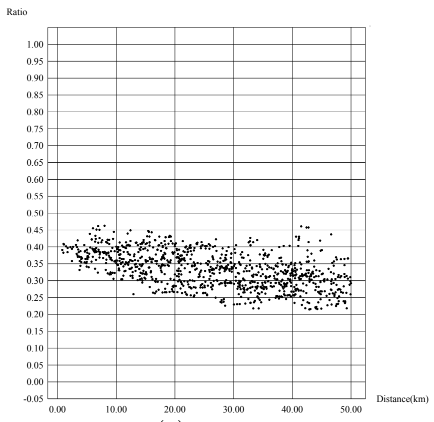


(5) 100 points

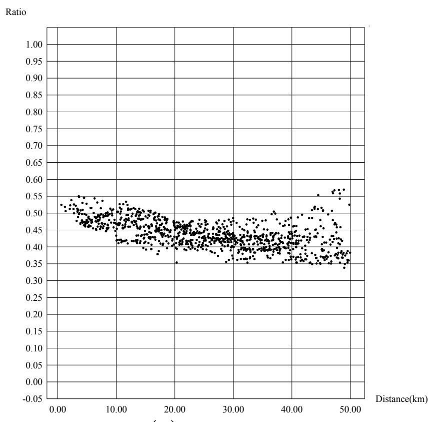


(6) 200 points

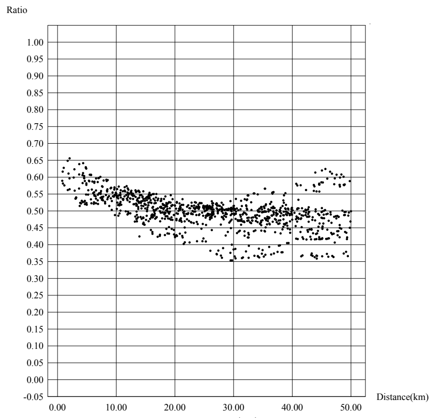
Fig. 1. Ratios of number of nodes searched by the A* Algorithm to that by the Dijkstra method.



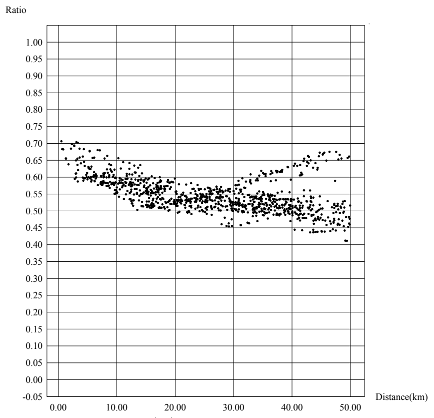
(1) 5 points



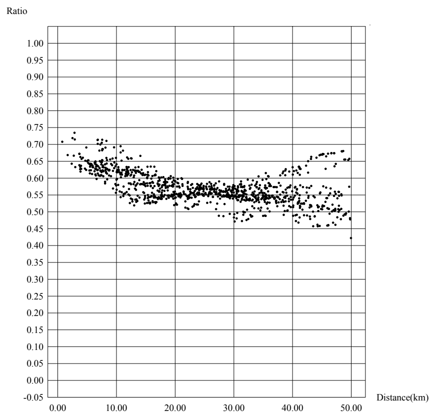
(2) 10 points



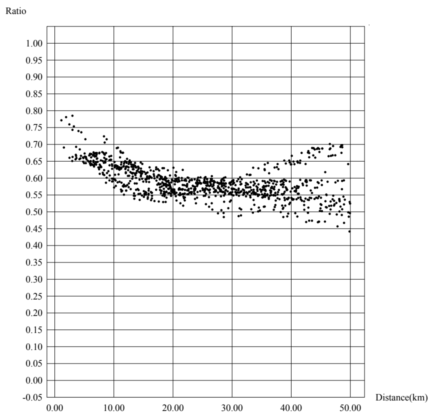
(3) 20 points



(4) 50 points

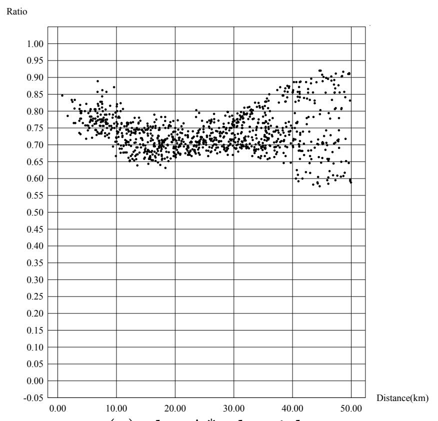


(5) 100 points

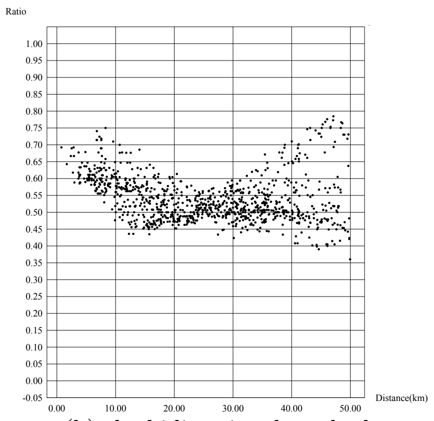


(6) 200 points

Fig. 2. Ratios of number of nodes searched by the bidirectional method to that by the Dijkstra method.

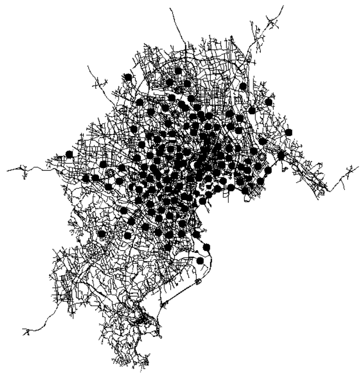


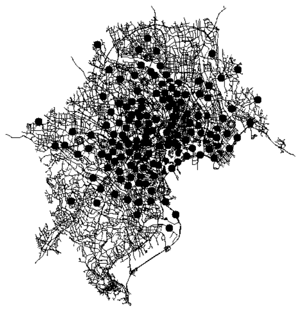
(a) the A* algorithm



(b) the bidirectional method

Fig. 3. Ratios of time taken by our algorithms to that by the Dijkstra method.


(a) Dijkstra method


(b) A* algorithm

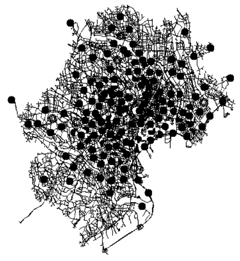
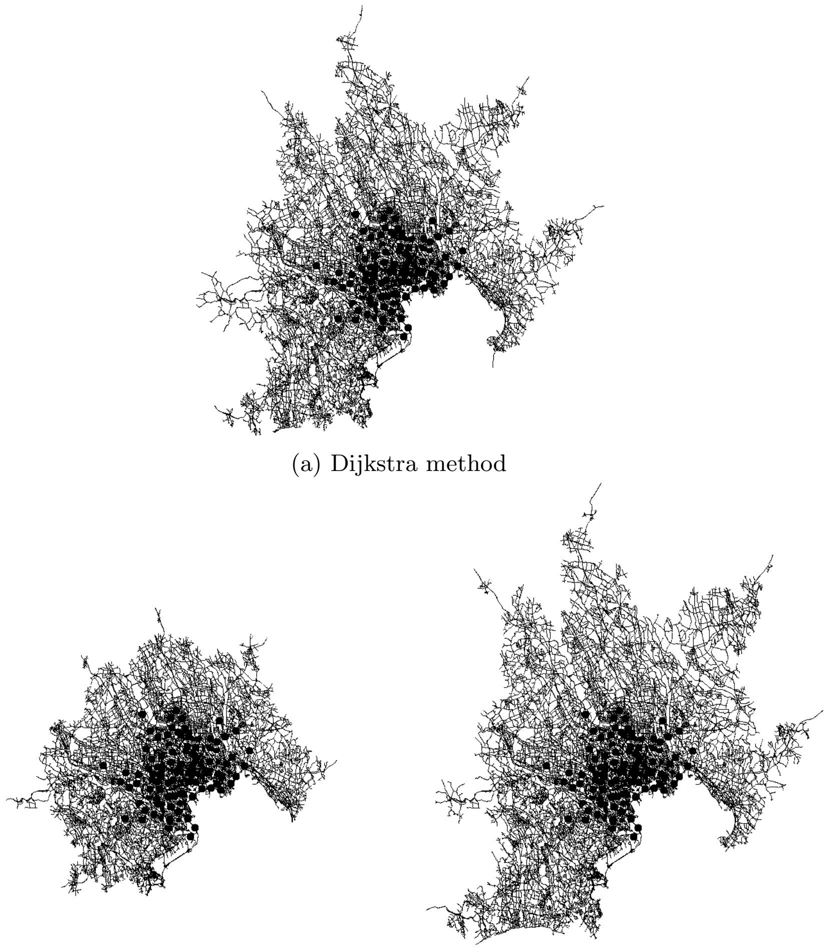

(c) Bidirectional method

Fig. 4. Searched regions from one of the sources by various algorithms


(a) Dijkstra method

(b) A* algorithm

(c) Bidirectional method

Fig. 5. Regions of vertices loaded by various algorithms

Image Watermarking for Copyright Protection

Gregory L. Heileman[*], Carlos E. Pizano[**], and Chaouki T. Abdallah[**]

Department of Electrical and Computer Engineering
University of New Mexico, Albuquerque, NM, USA 87131
{heileman,cpizano,chaouki}@eece.unm.edu

Abstract. In this paper we provide a description of the application of digital watermarking for use in copyright protection of digital images. The watermarking process we consider involves embedding visually imperceptible data in a digital image in such a way that it is difficult to detect or remove, unless one possesses specific secret information. The major processes involved in the watermarking processes are considered, including insertion, attack, and detection/extraction. Generic features of these processes are discussed, along with open issues, and areas where experimentation is likely to prove useful.

1 Introduction

The term *digital watermarking* is currently being used to describe the process of adding information, in particular digital watermarks, to digital multimedia content. Digital watermarks are being used for the purposes of documenting or ensuring (i.e., verifying, guaranteeing, or proving) the integrity of the multimedia content. Specifically, there are two general ways in which digital watermarks are being used with regards to multimedia content: (1) To add information to the content in such a way that it is clearly, and purposefully, available to those accessing the content. This is related to the traditional idea of watermarking (e.g., translucent marks that are routinely placed in paper currency or bond paper). A typical application is the automatic placement of a logo in an image taken by a digital camera. (2) To add information to the content in such a way that it is hidden from those accessing the content, unless they know to look for it, and possess any secret information needed to decode it. This is related to the traditional idea of *steganography*—a word derived from the Greek word meaning *covered writing*. The field of steganography is also referred to as *information hiding*; however, both of the uses described above are commonly referred to simply as *digital watermarking* in the current literature. The notion of embedding hidden or barely perceptible information within a message or picture is actually an old idea, dating back to antiquity. Interesting accounts of this history can be

[*] Supported in part by the Spanish Ministry of Education while on sabbatical at the Departamento de Tecnologías de las Comunicaciones, Universidad Carlos III de Madrid, Leganés-Madrid.

[**] Supported in part by Prometheus, Inc., 103 Mansfield St., Sharon, MA

found in [13, 14]. In this paper we will focus on the use of digital watermarking techniques for the purpose of copyright protection and ownership verification of digital images. This requires information to be hidden in digital images (i.e., this is an application of usage (2) described above).

For ease of exposition, throughout this paper the terms *watermarking* and *watermark* will be taken to mean *digital watermarking* and *digital watermark*, respectively.[1] The need for developing watermarking techniques that protect electronic information has become increasingly important due to the widespread availability of methods for disseminating this information (e.g. via the Internet), and the ease with which this information can be reproduced. In fact, the inability to develop provably strong watermarking methods for copyright protection is often cited as a major stumbling block to the further commercial development of the Internet. For this reason, watermarking research has received considerable attention over the past few years, and an ever-increasing number of watermarking methods for copyright protection are being proposed both in the open and patent literatures. These methods are invariably accompanied by the claim that they are "robust against malicious attacks," and these claims are often backed up via experimental tests devised by the developers themselves. Rarely do the authors of different papers consider the same suite of attacks, and worse yet, it is often the case that the same attack (e.g., subsampling) is implemented differently (e.g., some authors assume the image can be resized during a manual preprocessing step and others do not). The need for the establishment of a standard set of attacks by which watermarking methods can be benchmarked is clearly evident. Similarly, since the performance of all watermarking methods is image dependent, a standard image database must be established. In order to establish these standards, however, it is first necessary construct appropriate models and make reasonable assumptions regarding all of the processes associated with the use of watermarking algorithms. In this paper we attempt to establish an appropriate framework for these purposes, and also make the case that experimentation is likely to be an important tool in the development, testing, and possible standardization of watermarking algorithms.

In Sect. 2 we describe how watermarks are used for copyright protection, and present a general model for watermarking systems. We also describe desirable properties for watermarks, as well as some important constraints on what the processes associated with a watermarking system can do to an image. In separate subsections, we then treat in detail each of the major processes in our general model. These include the insertion, detection/extraction, and attack processes. In each case, our goal is to describe generic features of these processes, constraints that should be observed, along with any unresolved issues. In Sect. 3 we present a simple example the illustrates many of the issues discussed in the previous section. In Sect. 4 we summarize some of the weaknesses of current watermarking systems, describe developments that need to take place in order to correct them,

[1] There has been an effort to establish a common terminology for the digital watermarking field [24], but this terminology only addresses usages related to information hiding, and has not been widely adopted.

and give suggestions as to how experimentation is likely to be used to assist these developments.

2 Watermarking For Copyright Protection

In the literature, copyright protection is the application area in which watermarking is most often considered. In the remainder of this paper, unless explicitly stated otherwise, we will assume this application area when the term watermarking is used. The goal in this case is for the owner of a piece of watermarked information to be able to assert their ownership over the information after the information has been disseminated. A general model for this process in the case of digital images is depicted in Fig. 1.[2] In this figure, the (unwatermarked) input image is denoted I, the watermark is denoted w, and the watermarked image obtained from the insertion process is denoted I_w. The watermark is simply a signal that is embedded in data in such a way that it can later be detected or extracted in order to make some assertion about the data. In the figure, we distinguish between the detection and extraction processes. In detection, we are trying to determine whether or not a (possibly tampered with) test image $\hat{I}_w$ contains the watermark w, while in the extraction process we attempt to recover the actual bits associated with the watermark. These extracted bits are denoted $\hat{w}$. Most watermarking methods make use of a secret key (or keys) k in both the insertion and detection/extraction processes. The role of this key will be discussed in more detail shortly. Finally, note that we assume that I is not available during the detection/extraction processes. The reason for this will be discussed in Sect. 2.3.

Before discussing the watermark insertion process in detail, it is useful to consider a typical scenario: A content owner sells watermarked multimedia content to a customer who then makes the purchased content publicly available via a web page (e.g., for advertising purposes). Alternatively, a content owner may wish to make watermarked multimedia content publicly available via a web page through which prospective customers may view the content prior to making a purchase. In either case, content owners would like to protect their digital property from illicit copying by embedding a watermark in the content that is difficult to remove. In these cases, it is appropriate to use an invisible watermark for two reasons. First, an invisible watermark will not alter the aesthetic content of the image. Second, the work of malicious users, intent on removing the watermark, should be made more difficult if they do not know its location. Thus, for purposes of copyright protection, watermarks that are robust and invisible are desired. However, there exists a tradeoff between these two properties, as well as a third property, the information capacity of the watermark. These tradeoffs

[2] Although we restrict our attention to digital images in this paper, the reader should be aware that most of the techniques proposed for watermarking digital images for the purpose of copyright protection are also applicable (in varying degrees) to audio and video. See [30] for a survey.

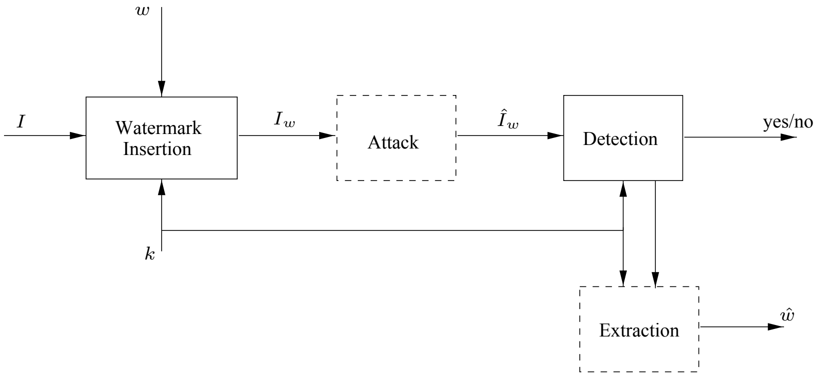


Fig. 1. A general model for watermarking in the copyright protection application. Dashed boxes indicate optional processes.

will be discussed in more detail in Sect. 4 after we have had the opportunity to discuss the entire watermarking process depicted in Fig. 1.

The required invisibility of the watermark in the image leads to the following constraints: The watermark insertion cannot change the perceptual content of the image I, and an attack will only be considered successful if it defeats the detection/extraction processes without changing the perceptual content of the image I_w. We will use the notation $I_a \sim I_b$ to denote that image I_a is perceptually equivalent to I_b. The constraint $I \sim I_w$ models the typical situation in which a graphic artist, photographer, etc. has created an image in which they would like to place a watermark, but they require that this be done without altering the esthetics of the image. The constraint $I_w \sim \hat{I}_w$ captures the notion that a pirate will only find it useful to steal an image if they can do so without altering its esthetics. Thus, if it is possible to construct a a watermarking method in which perfect detection is obtained whenever $I_w \sim \hat{I}_w$, then a pirate will be forced to noticeably alter an image in order to steal it. We believe that this is the best that one can hope for in this application.

The main problem with the previous discussion is that the concept of a perceptual invariant is an ill-defined, and in fact involves open problems in brain science, psychology, as well as subjective opinions. Nevertheless, enough is known about the human visual system to exploit the idea of perceptual invariants in watermarking methods. We will present one useful model for the human visual system in the following section when we discuss the watermark insertion process.

2.1 Watermark Insertion

In Fig. 2 we present a more detailed model of the watermark insertion process. It is not uncommon for a watermarking algorithm to include some type of domain transformation and perceptual analysis prior to inserting the watermark w into the image I. The transform domain coefficients are denoted β, and the results of perceptual analysis (often an image known as a *perceptual mask*) are denoted I^p. These elements, along with a secret key k and the image pixel values themselves, may all be used to insert the watermark w.

Watermarking methods can be divided into three groups based upon the manner in which the watermark is inserted in the image: (1) spatial domain methods embed the watermark by altering intensity values in the image I (e.g., [4, 10, 17, 21]), (2) frequency domain methods embed the watermark by altering coefficients in some transform (e.g., DCT, FFT, wavelet) domain of the image I (e.g., [5, 16, 26]), and (3) colorspace methods that insert the watermark in the colormap of an indexed image (e.g., [25]).

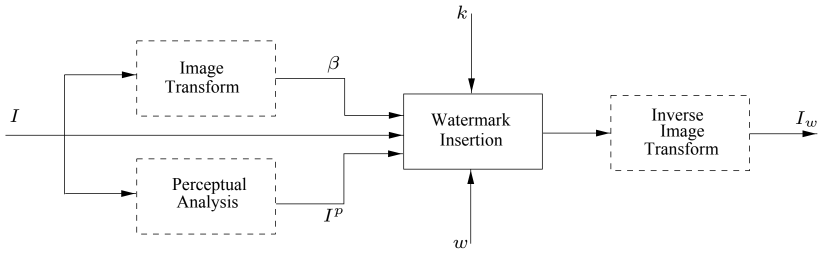


Fig. 2. A more detailed view of the watermark insertion process.

The key k shown in Fig. 2, typically chosen as a pseudorandom number, may determine not only the exact placement of the watermark signal in the image, but also the watermark signal itself (in some methods two pseudorandom keys are used for these purposes [10]). We will see in Sect. 2.2 that some of the most successful attacks on a watermarking system work by "breaking the synchronization" between the watermark w and the watermarked image I_w. For example, if the value of the key k is used directly to select pixels for watermarking, then even a slight amount of cropping or rotation is often enough to "misalign" the detection algorithm when it is searching for a watermark w in a test image $\hat{I}_w$. To combat this, more sophisticated watermarking algorithms insert synchronization signals into the image I during the insertion process. Many of these algorithms use techniques originally developed for spread-spectrum communications [11, 31, 32, 33, 34].

In order to complete our discussion of the insertion process, and understand how a watermark can be inserted robustly, while at the same time maintaining invisibility, it is necessary to consider in more detail the properties of digital images along with the properties of the human visual system.

An Image Model. An $N \times M$ digital image I can be thought of as a two-dimensional random process that represents the sampling (i.e., digitization) of a real or synthetic scene. These samples are called pixels and are denoted by

$I(x, y)$, $x \in \{0, \dots, N-1\}$, $y \in \{0, \dots, M-1\}$. Although it is possible to develop models for mathematically describing the properties of an image I, such descriptions are highly dependent on the image content [3]. Thus, the watermarking insertion process also tends to be highly image dependent. For this reason, many watermarking methods compute a perceptual mask during watermark insertion and use this to determine watermark placement within the image. This is discussed in more detail below.

The format of an image can also impact the watermarking process. The image format determines the type of information that is stored for a given pixel. Three different formats are commonly used. The first, used for color images, represents each pixel directly as a triplet of nonnegative real numbers. The model used for the color space determines what type of information is stored in the triple. Often these numbers are used to represent the red, green, and blue intensity values, respectively, of the image. These values, typically 8-bit numbers, are combined to produce the color of the pixel. The image itself is referred to as an *RGB image*. The JPEG format is an example of a specific methodology for representing images in this fashion. An alternative, and less common, color space is the HSV model. In this case, the triple represents the hue, saturation, and value, respectively, of the corresponding pixel. The second format, also used for color images, stores an image using two arrays: an image array and a colormap array. An image of this type is referred to as an *indexed image*. The colormap is an $n \times 3$ array containing the possible colors that may appear in the image. Each of the n rows stores three values—typically these are RGB components for a specific color. Each pixel in the image array stores an integer value in the range $[0, \dots, n-1]$ that corresponds to an index into the colormap. That is, the colormap is simply a lookup table. The GIF and TIFF formats are specific examples of this format. As we have previously mentioned, the colormap itself can be used for watermarking. The final format we consider is used to store grayscale images. Like the RGB image format, the pixel values are stored directly without the use of a lookup table. Typically each pixel is represented by an 8-bit integer value in the range $[0, 255]$, with 0 representing the black end of the spectrum, and 255 representing the white (or full intensity) end of the spectrum. RGB and indexed images are often converted to intensity images. Although there is the possibility of lossless conversion between certain image formats, in general this is not the case, and a certain amount of degradation will occur. In fact, this is one of the more probable forms of attacks since image conversions are often carried out automatically by software programs. This issue is discussed in more detail in Sect. 2.2 we consider accidental attacks.

The Human Visual System. The human visual system (HVS) has known characteristics that allow certain changes to the pixel values in an image to go undetected. The amount of change that can occur is directly related to the image composition, and to several HVS phenomena known as masking, spatial resolution, intensity resolution, and blue channel intensity. We present a simple model of the HVS next, and then consider each of these HVS phenomena in turn.

A simple model of the HVS is shown in Fig. 3 [18]. The *peripheral level* is the portion of the HVS where light is converted into neural signals. This level is fairly well understood, particularly for monochrome images. The *central level* processes neural signals in order to extract information. Simple models for the peripheral level exist, but not for the central level.

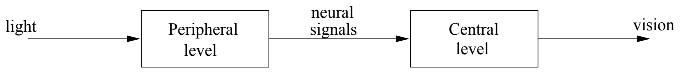


Fig. 3. A model for the human visual system.

A simple model of the peripheral level for monochrome images, that agrees with a number of well-known HVS phenomena, is shown in Fig. 4 [29]. The nonlinearity shown in this figure compresses high-level intensities, and expands low-level intensities (similar to a logarithm operation), while the bandpass filter can be modeled as a linear shift invariant system with spatial frequency response that is maximum for mid-range frequencies (5–10 cycles/degree). It is believed that finite pupil size, and the finiteness of the receptor cells in the eye may cause the low-pass response, while the lateral inhibition between receptor cells may cause the high-pass response.

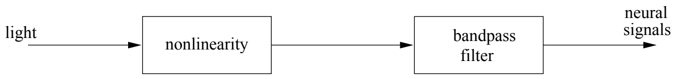


Fig. 4. A model of the peripheral level of the human visual system for monochrome images.

One or more of the following perceptual "holes" in the HVS are typically exploited by modern invisible watermarking method. All are thought to result from peripheral level processing. We are not aware of any watermarking methods that exploit central level processing. Conceivably such methods would be quite strong, but they would rely on understanding the content of an image. However, the field of image understanding contains many problems that need to be solved before this approach would be feasible. Thus, we restrict our attention to what is believed to be peripheral level processing in the following discussion. For each of the visual phenomena discussed next, we also discuss the impact it has on watermarking, and in particular on the constraint $I \sim I_w$.

Mach-band Effect. In this well-known phenomenon, depicted in Fig. 5, the HVS perceives a change in intensity in regions that actually have a constant intensity. For example, each of the regions shown in Fig. 5 has a uniform intensity, yet each region appears darker towards the left, and lighter towards the right. Thus, the HVS has an uneven brightness perception within regions of uniform intensity. This response is consistent with spatial filtering, where overshoot and undershoot occur where the signal has sharp discontinuities. For the purposes of watermarking this phenomenon demonstrates that precise preservation of edge shapes are not necessary in order to satisfy the constraint $I \sim I_w$.

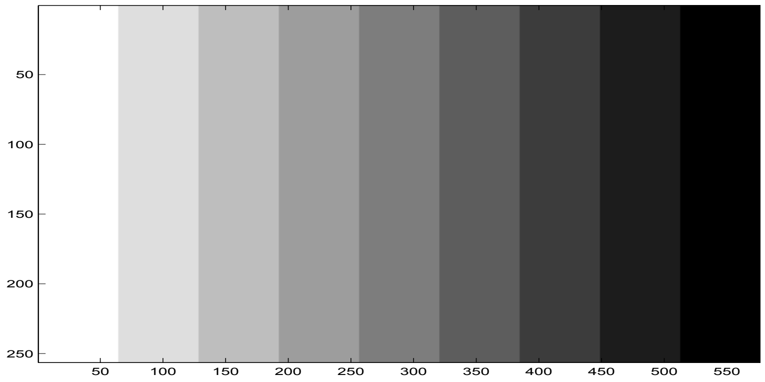


Fig. 5. An example of the Mach-band effect in the human visual system.

Intensity Resolution. This feature of the HVS refers to the fact that small changes in the intensity of pixels in I will be unnoticed. For the purposes of watermarking, this means that the watermark bits must be placed in the lower-order bits of an image if $I \sim I_w$ is to be maintained.

Intensity Sensitivity. This phenomenon refers to the fact that in the HVS, the amount of unnoticed change in intensity that can occur in an image varies according to the intensity of the surrounding pixels. Specifically, the value of ΔI that causes a just-noticeable difference to occur between intensities I and $I + \Delta I$ increases proportionally with the intensity of I. In fact, in the log domain, the just-noticeable difference is constant for a wide range of I, which is consistent with the nonlinear processing previously described for the peripheral level. For the purposes of watermarking this means that darker regions can support larger changes to their pixel values than can lighter regions.

Spatial Resolution. This phenomenon refers to the fact that isolated pixels in an image can often be changed without being perceived by the HVS. The main problem with exploiting this feature is that isolated pixel modifications will introduce higher frequency components in the spectrum of the modified image. These are easily removed via low-pass filtering, with minimal perceptual effect on the image. That is, it will in general be easy to remove watermarks that exploit this phenomenon, while at the same time maintaining $I_w \sim \hat{I}_w$.

Spatial Masking. This phenomenon accounts for the fact that when random noise is added to an image, it is more noticeable in regions that have uniform intensity values, than in regions that have high contrast (i.e., edge regions which correspond to high spatial frequency). Furthermore, noise in dark regions is less noticeable than noise in light regions. Thus, for the purposes of watermarking, the intensities of pixels that are close to a edge can be changed a great deal without being perceived, particularly dark edges (due to intensity sensitivity). In addition, large changes to pixels within a homogeneous color or intensity region should be avoided, particularly in lighter regions.

Blue Channel Insensitivity. This refers to the fact that the HVS is not symmetric with respect to the manner in which changes to the three basic colors are perceived. In particular, the HVS is roughly three to five times less sensitive to changes in the blue component of a image, than it is to changes in the red and green components. This is often exploited in the watermarking of color images by placing the watermark primarily, or entirely, in the blue channel. The blue channel can be simply treated as an intensity image in the latter case.

One or more of the aforementioned perceptual holes in the HVS are typically exploited by invisible watermarking methods. Indeed, any watermarking method must exploit some type of perceptual hole if it is to be considered invisible. This discussion also makes it clear that not every pixel is equally suited for encoding watermark bits. Thus, the purpose of the perceptual analysis shown in Fig. 2 is to identify a perceptual mask I^p that can be used to insert a “stronger” watermark. Specifically, the goal is to use I^p to amplify watermark strength in regions where the change will be less noticeable (for robustness purposes), and decrease it in regions that are sensitive to changes (for perceptibility purposes). In addition, we would like to place more of the watermark in perceptually significant regions of the image. If we are successful at this latter task, then in order to remove the watermark, the perceptual characteristics of the image would have to be changed, leading to a violation of the constraint $I_w \sim \hat{I}_w$.

2.2 Attack Process

We now consider the attack process depicted in the general watermarking model of Fig. 1. This is certainly one of the more difficult processes to characterize, and its treatment in the current literature is the source of much confusion. We believe much of the confusion is alleviated by characterizing the robustness of watermarking methods with respect to specific classes of attacks. A number of categories for attacks have been described in the literature, e.g., [8, 23]. For the purposes of the copyright protection application as we have described it, we believe that at least the following classes of attack make sense for reasons discussed below: accidental, robustness, protocol, and legal attacks. It seems unlikely that watermarking methods can be developed that are robust against all of these categories of attack (at least in the near term), but success on specific attack categories is more probable. Let us consider each of these in turn.

Accidental Attacks. These attacks are caused by routine manipulation of an image, such as compression or image conversion during downloading or email transmission, cropping and/or rotation during image acquisition, etc. For example, we envision methods that are robust to this form of attack being used in monitoring applications where "accidental pirates" are identified and subsequently notified that they are in possession of copyrighted material. We feel that such an approach may prove effective since a warning is often all that is needed to deter the accidental pirate.

What distinguishes this class from the robustness attacks, discussed next, is that methods designed with accidental attacks in mind do *not* have to be concerned with a malicious attacker. Rather, they only need to be made resistant to common forms of image manipulation in order to be considered useful. For methods to be robust against these forms of attack, it is important to understand the technology associated with the handling of images. This in concept appears to be an easier situation to deal with, rather than being forced to stay "one step ahead" of a malicious adversary who is actively seeking to thwart watermarking efforts.

Specific attacks that should be considered here include JPEG compression across a common range of compression ratios, conversion between file formats (e.g., bitmap to JPEG, TIFF to JPEG, and vice versa), and minor geometric transformations (e.g., cropping, scaling, rotation) that occur while manipulating an image.

Robustness Attacks. With robustness attacks, we are confronted with an adversary that is actively trying to make the watermark unreadable, without affecting the quality of the image. The attacker may in fact attempt to manipulate the watermark signal directly. Examples of robustness attacks might include low-pass filtering, histogram equalization, frequency domain smoothing, median filtering, the addition of noise, as well as geometric transformations such as cropping, rotation, or mirroring. The results of our experimentation indicate that current watermarking algorithms are not at all immune to fairly simple forms of robustness attacks [12]. For example, simple geometric attacks, even though they leave the actual watermark information in the image, have a devastating effect on watermark detection. Ironically, these are also the most likely forms of attack, since they are easily performed using freely available software. This suggests that a useful line of research might involve watermarking methods that can account for these common distortions during the insertion process, or invert them during the detection/extraction process.

Robustness attacks appear to be closely related to the field of cryptography, where malicious adversaries are commonly assumed. In fact, it is useful to briefly consider cryptography when considering this class of attacks. The Fundamental Axiom of Cryptography, which is assumed by all reasonable cryptanalysts, is that the adversary has full knowledge of the details of the cipher, and lacks only the specific key used in encryption [27]. This principle was adopted as far back as 1883 by Kerckhoffs [15], and history gives us ample reason to heed this axiom—a reading of the American code breaking effort during World War II is convincing enough [13]. For these same reasons, it is prudent to adopt a similar Fundamental Axiom of Watermarking when we consider robustness attacks. Namely, we should assume the adversary knows not only that watermarks are hidden in the image, but the algorithm being used to hide them (only secret keys associated with the algorithm are unknown). This axiom makes sense for another reason, if an adversary is to be challenged in a court of law, then certainly the details of the watermarking algorithm used to identify the suspected cheating will have to be divulged.

Although cryptography and watermarking appear to have much in common, it is also instructive to note some differences. For example, with cryptography once a message is decoded, it is completely "in the clear," but with watermarking the watermark is part of the message. Thus, data can continue to be protected *after* it has been decrypted. In addition, with cryptography we are usually trying to prevent illicit acts, but with watermarking we are trying to prove that an illicit act has occurred after the fact. Indeed, watermarking appears to be the only solution available for protecting intellectual property from active pirates and performing use-tracking *after* content has been securely transmitted. Thus, the two are likely to be used together in certain applications—traditional cryptography to ensure a message is not compromised during transmission, and watermarking to ensure the receiver, or others, do not misuse the information after it is in the clear. Other differences are revealed if we consider the goals of the user and pirate. Specifically, with cryptography the goal of the user is to conceal the content of a message. The goal of the user in watermarking is to conceal the existence of the message. The goal of the pirate in cryptography is to uncover the secret information contained in a transmission. It is usually easy to verify success/no success in this effort. With watermarking, the goal of the pirate is to destroy the watermark. If any information from the watermark remains in the content, it may be possible to recover the watermark. It is difficult for the pirate to verify success/no success in this effort. We believe it is important to keep these differences in mind when attempting to apply cryptographic theory to the development of watermarking systems.

We may also look to cryptography to help with notions of security for watermarking systems in the face of robustness attacks. For example, cryptographers have developed well-defined notions of unconditional, computational, provable, and practical security that are tied to the computational resources required by the malicious attacker in order to compromise a cryptosystem [28]. However, such a classification for watermarking systems has not been forthcoming. This may be due to the fact that watermarking technology, as compared to that of cryptography, is relatively immature and rapidly changing. For instance, compromises in cryptosystems are now mostly due to protocol failures, direct assaults on the cryptographic algorithms themselves are not very common. The same is not true for current watermarking systems; a number of effective algorithms are available for directly attacking suspected watermarks in images [22, 23, 35]. Indeed, comprehensive protocols for watermarking systems have yet to be developed. There may be other difficulties associated with developing theories of security in watermarking systems. Consider for example that with cryptosystems it is completely reasonable (and in fact desirable) to send a message that appears as random noise; however, this is not possible with watermarking systems since we are constrained by $I \sim I_w$. In addition, note that the value associated with "breaking" a given message in a cryptosystem tends to decrease rapidly with time. Copyrights, however, can exist for as long as 75 years after the death of the author. Thus, when judged against the rate at which advances in computation are being made, watermarks must provide protection for an extremely long period of time. Thus, notions of security may have to take on a slightly different meaning when applied to watermarking.

In summary, since we assume the attacker knows the details of the watermarking method, we believe the only chance for success here appears to require that the insertion process take the watermarked image close to the limit of $I \sim I_w$, so that any additional change will push I_w "over the edge" so that $I \not\sim I_w$. That is, we would assume the attacker can also compute a perceptual mask, and attack the image as strongly as possible according to the perceptual mask. However, if attackers are not left with enough "room" to effectively attack the image, they will be forced to make $I \not\sim I_w$.

Protocol Attacks. In a protocol attack, the attacker does not try to remove the watermark directly. Rather, an attacker attempts to cast doubt on the ownership of the watermark, or avoid its detection, by exploiting weaknesses in the watermarking protocol or other loopholes associated with the usage of the system. An example is the *multiple watermark attack* that involves placing additional watermarks in the image in an effort to create "ownership deadlock". This attack is successful when a watermarking method does not provide a mechanism for determining with of two watermarks was added first [6, 7]. It appears that some timestamping mechanism, via a central authority, is necessary in order to deal with this attack.

Other examples of protocol attacks include a failure to keep the secret key k secret, or a *conspiracy attack* where multiple owners of a watermarked image (each with a different watermark) get together and use these copies to determine the location of the watermark in each image.

Another form of protocol attack deserves special mention (and possibly its own attack category); these are the so-called *presentation attacks* which are intended to defeat "webcrawler" applications (e.g., Digimarc's MarcSpider [9]) that search the Web looking for watermarked images. In a presentation attack, the manner in which the image is presented is changed, not the underlying pixel values. For example, an image might be broken into pieces (with each piece written to a different file), but the pieces are put together into a mosaic when displayed [22, 23]. This attack has proven very successful because most watermarking methods have difficulties embedding watermarks in small images (e.g., below 100×100 pixels).

Legal Attacks. This refers to the ability of an attacker to cast doubt on the validity of a watermarking method in a court of law. Attacks of this form can only be judged successful or unsuccessful after they have been tried in a court of law. Yet it seems that detection methods having a sound statistical basis are more likely to survive such attacks. The usage of watermarking methods will likely force the establishment of legal precedents at some point in the near future.

In summary, we feel it is useful to categorize the different types of attack that a watermarking system may be subjected to in order to develop appropriate notions for their security. Furthermore, the features that might be incorporated into a watermarking system in order to make it resistant to accidental attacks are likely to be different from those that are incorporated to make a system resistant to protocol attacks. The establishment of appropriate benchmark attacks for at least the accidental and robustness attack categories would be very worthwhile.

2.3 Watermark Detection/Extraction

We now consider the problem of detecting and extracting a watermark from an image that may have been attacked. Figure 6 presents a more detailed view of the detection process. An important observation to make at this point is that we assume the original image is not available during the detection/extraction processes. This constraint is generally adopted for the following reason: If the watermarking system is to be used in a mode that allows arbitrary third parties to check the copyright status of an image, then obviously I must be kept secret or there is no purpose in watermarking it. This constraint is not necessary if the watermarking system is to be used for monitoring purposes only—in this case the insertion and detection/extraction processes will generally be accomplished by the same entity. Nevertheless, the case for this constraint is still made by many researchers using the argument that excessive data handling will result if the original image is required during watermark detection/extraction. It seems that the former usage would require some type of public-key methods (similar to those used in public-key cryptography); however, we are not aware of any such methods.[3] Rather, the key used during insertion must be secretly communicated to the party performing the detection process in order to maintain security for the overall system. Unless such public-key methods can be developed, it will not be possible to deploy secure watermarking algorithms that are capable of informing arbitrary third parties about the copyright status of a possibly attacked image. We note, however, that for many users, communicating copyright information to third parties has lower priority than utilizing watermarking technology to monitor the use of their content and to identify cases of copyright abuse. The former can be accomplished with public-key technology, while the later requires only private-key technology.

[3] A restricted notion of public-key watermarking is described in [2].

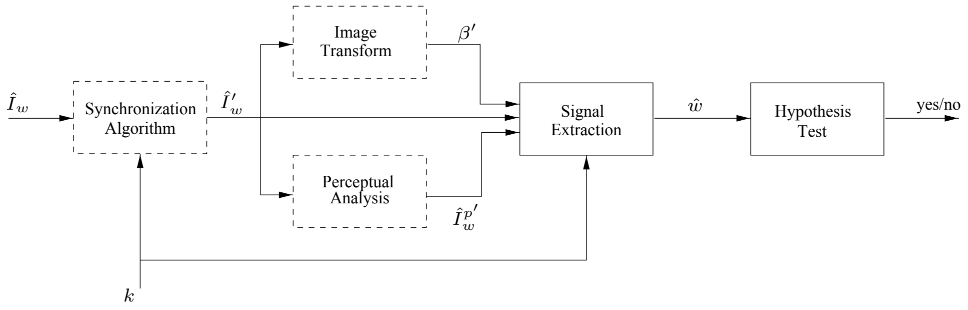


Fig. 6. A more detailed view of the processes involved in the watermark detection process.

Returning to Fig. 6, notice that an optional synchronization step preprocesses the test image $\hat{I}_w$ in an attempt to create a "realigned" test image $\hat{I}'_w$. This step is only necessary if a synchronization signal was embedded in the watermarked image during the insertion process. Next, an optional image transformation or perceptual analysis may be performed. Which processes are performed at this step will depend upon those processes performed during watermark insertion. We note, however, that since we assume $I_w \sim \hat{I}_w$, we can use "perceptual invariants" in the test image, and assume they existed in the presumed original image I.

After the suspected watermark signal is extracted, a decision must be made as to whether or not the extracted signal corresponds to the watermarked signal. The part of decision theory known as hypothesis testing is often used to make the yes/no decision regarding the presence of a watermark in the test image I_w.

The extraction process, not shown in Fig. 6, is similar to the detection process, except that we are interested in explicitly recovering the watermark bits. In this case, these bits will normally encode additional information, such as the name of the author of the watermarked information, the date of copyright, etc. The use of error-correcting codes during extraction is a logical idea, and has in fact been investigated [10, 11].

3 An Example

It is instructive at this point to present an example the illustrates many of the important concepts we have considered. In this example we will use the least-significant-bit (LSB) watermarking method, which embeds the watermark in the spatial domain of the image. This was the first approach suggested for invisible watermarking in the modern era [32], is quite easy to implement, and illustrates many of the important concepts have considered. Furthermore, this method (or some variation thereof) is used in a number of currently available software packages for watermarking. The LSB method exploits the fact that changes to the least significant bits of the pixels in an image will yield changes that are below the intensity resolution of the HVS (i.e., it exploits the intensity resolution phenomenon). To illustrate this fact, consider the original 512×512 image shown in Fig. 7 (a). In Fig. 7 (b), the same image is shown, but the least significant bits of a 124×96 block (in all three channels) of the image have been modified so that they store the entire text of Lincoln's Gettysburg Address shown in Fig. 8 (a). That is, the Gettysburg address is stored in the image three times, once in each of the red, green, and blue channels. Even on a high-resolution monitor, with the images side-by-side, there are no perceptible differences between the original and watermarked images. This demonstrates that it is possible to store a large amount of information in an image without making any perceptible changes.

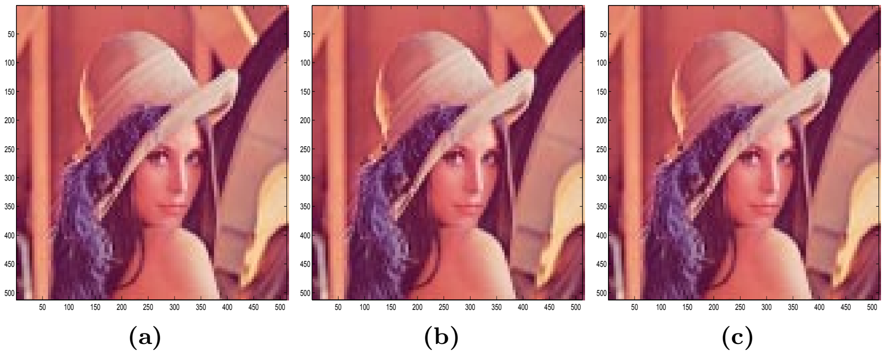


Fig. 7. (a) Original Image. **(b)** Original image watermarked with Lincoln's Gettysburg Address. **(c)** Watermarked image after it has been attacked with additive Gaussian noise ($\mu = 0$, $\sigma^2 = 0.000001$) on all three channels.

Next, a very simple robustness attack is applied to the watermarked image shown in Fig. 7 (b). Specifically, random Gaussian noise with zero mean and a variance of 0.000001 is added to every pixel of the image on all three color channels. Because this noise level is so small (below the intensity resolution of the HVS), once again the perceptual effects are not detectable (Fig. 7 (c)). However, when the watermark is extracted (Fig. 8 (b)), we see that it is severely degraded. Specifically, the watermark extraction algorithm took a majority vote of the bits in all three channels of the corrupted image in the location of the original watermark. If a slightly larger variance is used, the watermark is completely destroyed. This demonstrates the extreme fragility of this method to even simple attacks, and illustrates the trade-off discussed previously between the amount of information contained in a watermark and its robustness. As we have mentioned, at least two techniques can be used to make the watermark more robust: use a perceptual mask to more aggressively mark selected pixels, and repeating watermark bits throughout the image.

```
Four score and seven years ago, our fathers brought forth upon this continent a new nation:
conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war. . .testing whether that nation, or any nation so
conceived and so dedicated. . .can long endure. We are met on a great battlefield of that
war.

We have come to dedicate a portion of that field as a final resting place for those who
here gave their lives that this nation might live. It is altogether fitting and proper
that we should do this. But, in a larger sense, we cannot dedicate. . .we cannot
consecrate. . .we cannot hallow this ground. The brave men, living and dead who
struggled here have consecrated it, far above our poor power to add or detract. The
world will little note, nor long remember, what we say here, but it can never forget what
they did here. It is for us the living, rather, to be dedicated here to the unfinished
work which they who fought here have thus far so nobly advanced. It is rather for us to
be here dedicated to the great task remaining before us. . .that from these honored dead
we take increased devotion to that cause for which they gave the last full measure of
devotion. . .that we here highly resolve that these dead shall not have died in
vain. . .that this nation, under God, shall have a new birth of freedom. . . and that
government of the people. . .by the people. . .for the people. . .shall not perish
from this earth.
```

(a)

```
@oUr s@ose i.@ sere@ Yeqps ago, mur(vatherc broegIt &krth up@n thyq!cGftioeNv@a nmw .adion:
$conceivem En liber@y, a@d de@icaped @o@the prGtosytion dhAt ‘ll@men"are cveatg@(eqqa@>

Now@w‘ aR@0enfagED in a gsEat civil Ser. . .teqding‘whud@e2 txav$~ataon- or a@Y naTynn so
concekvdd and(So$dedicat@d.@. .ca. l/nc eldurg."Ge aru mep onaq"gpeqt batt,efield of(that
7ar.

Ug xara!come"|o egdIkate i pord)on‘od that$fiel$@a{ m f9nal vesty.f place(tgr djose Whm@
her} gavg(thai3 @ives$d@at$t@is nadion might |ive6 I@ is qlt@geth-p dipTioG anv"proXmr
that @e shoEld‘do t@iSJwp@ in a largec0s@n3m< we!saj|o@(ded-kaden & .wE cannot‘
kg~seazatu.", @we canlot hallo? T@hq@gpGune. Th% bra@‘ men, li^i@g cnd $ua$@w(o
stzu@gled he2e h@Ve(coNcgcrAtad it, fas ab/ve"ous rokz @/sgr t/ ad@@/r!detVA@t@‘ \xe
w@zld will dittle@node, @or‘lkng@relumber, whav we sqi here,@bwt yt aan nev@r@@opogT@what
they d@d |ere. It is!fo6 us the livkng, za4her( to je d@dkca4Ed ler@ t@ the$uhfiLished
Wk@k wJich@@he{0@ho fough| hewe@ha~A 4hus f‘r s@ nobly advanced.1It k3 zatle{"dkr us to
be hesd de@icatee(to tje0great task Ramein@fg(bedo@d ua@ , .that frnm phase$ho~g@ed @mqd
we‘take incv@csed ddvotioo @o thCp‘gaus@ dob Which txe9 ga~e‘the last &Ult m%asure@of
dE@0tIon. . nthet @e here hichni!@e@olve thau"theRE)dea$ vHqllhnou@lcve di%d @n
tai@@"> /dhat thi@ natykn,!under g@d, SH‘ll have a kEu birti of .reetol. .@. and that
govdRnment of@tle xd0p@m. @ nj@ the xeoplE, .@.fos t@e peo@lA,@. n3hall ngt ‘avish
frNm thqs$earth&
```

(b)

Fig. 8. (a) The original Gettysburg Address text watermark inserted in the test image shown in Fig. 7 (a). **(b)** The Gettysburg Address text watermark extracted from the test image after an attack involving the addition of zero mean, 0.000001 variance Gaussian noise to all three channels of the watermarked image. Unprintable ASCII characters have been replaced by the @ symbol

From this example, it is clear that there is enough "room" in a typical image to add a certain amount of information in such a way that it will not be perceived by the HVS. The only problem is whether or not this can be done in a robust fashion, i.e., so that it is difficult for the accidental or malicious attacker to remove the watermark.

4 Conclusions

Throughout this paper we have discussed a number of open issues related to the development of robust watermarking algorithms for copyright protection of digital images. We summarize the difficulties of this problem by considering the tradeoffs associated with watermark *robustness*, *imperceptibility*, and *information capacity*. A watermarking technique cannot simultaneously maximize these three quantities; rather, they tend to work against one another, as is depicted in Fig. 9. Specifically, in this figure we denote the space of all watermarking methods accorinding to the three axes of robustness, imperceptibility, and information capacity. The methods that are realizable are shown as the shaded region in the figure, we envision them as forming a connected region about the origin, but we make claims about the actual shape of this region. This figure supports the notion that it is possible to make a watermark more robust to various forms of attack, but this must come at the expense of conveying less information in the watermark, and more image degradation. For example, watermarks can generally be made more robust to attacks if the watermark bits are replicated throughout the image (this is particularly helpful against the geometric attacks discussed in Sect. 2.2), but this obviously leads to a reduction in the information capacity of the watermark. Furthermore, a watermark can be made more robust to attack if it is placed in perceptually important regions of the image (see the discussion of the HVS given in Sect. 2.1). In this case, attacks that attempt to remove the watermark are likely to result in noticeable changes to the image. However, only a limited amount of data can be hidden in these regions; if too much is hidden, the watermark itself leads to unacceptable perceptual changes. Finally, since an increase in the information capacity of an efficiently coded watermark will require an increase in the number of bits used by the watermark, we see that increasing the information capacity will eventually lead to perceptually significant image modifications. For this reason, a desirable property of watermarking methods used for copyright protection is the ability of the user to specify the level of robustness. The user can then examine the watermarked data to determine if it is acceptable, and repeat this process with a different level of robustness if necessary.

We have only briefly considered the issue of computational constraints when the attack process was discussed in Sect. 2.2. However, computational requirements may also play a limiting role in the achievability of the specific watermarking methods depicted in Fig. 9, and will factor prominently in the development of any watermarking protocols. These issues require further investigation.

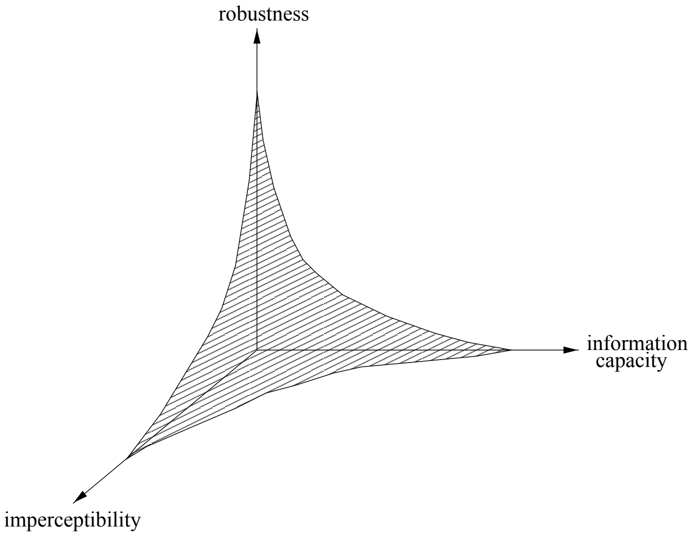


Fig. 9. The space of all watermarking methods for copyright protection.

Increasingly, calls are being made for the establishment of watermarking standards (c.f., [19]). Prior to the establishment of any standards for attacks or image databases, it will be important to establish reasonable models for watermarking. We believe this paper is a useful step in that direction. It will also be important to have unbiased means of judging watermarking methods during any standardization process. Based on the discussions contained in this paper, we believe experimental investigations can help lend direction to theoretical research on watermark robustness in this case. This is not to claim that theoretical work on watermark robustness is not being attempted. However, the viability and usefulness of this combined approach to research in the study of watermarking algorithms (i.e., an interchange between rigorous experimentation and theory [1, 20]) is justified given the immature state of development of the watermarking field, along with the image dependence of all watermarking methods.

References

[1] R. J. Anderson. The role of experiment in the theory of algorithms. In *5th DIMACS Challenge Workshop: Experimental Methodology Day*, Rutgers University, Oct. 1996. `http://www.cs.amherst.edu/~dsj/methday.html`.

[2] R. J. Anderson. Stretching the limits of steganography. In *Information Hiding, Springer Lecture Notes in Computer Science*, volume 1174, pages 39–48. Springer-Verlag, 1996.

[3] M. F. Barnsley and L. P. Hurd. *Fractal Image Compression.* AK Peters, Ltd., Wellesley, MA, 1993.

[4] W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM Systems Journal*, 35(3&4), 1996. `http://www.almaden.ibm.com/journal/sj/mit/sectiona/bender.pdf`.

[5] I. J. Cox, J. Kilian, T. Leighton, and T. Shamoon. Secure spread spectrum watermarking for multimedia. Technical Report 95-10, NEC Research Institute, May 1995. `http://www.neci.nj.nec.com/tr/index.html`.

[6] S. Craver, N. Memon, B.-L. Yeo, and M. Yeung. Can invisible watermarks resolve rightful ownership. Technical Report RC20509, IBM Research Report, July 1996.

[7] S. Craver, N. Memon, B.-L. Yeo, and M. Yeung. Can invisible watermarks resolve rightful ownership. In *Proceedings SPIE Storage and Retrieval for Image and Video Databases V*, July 1997.

[8] S. Craver, N. Memon, B.-L. Yeo, and M. Yeung. Technical trials and legal tribulations. *CACM*, 41(7):45–54, July 1998.

[9] Digimarc Homepage. `http://www.digimarc.com`.

[10] J. R. Hernández, F. Pérez-González, J. M. Rodríguez, and G. Nieto. Performance analysis of a 2d-multipulse amplitude modulation scheme for data hiding and watermarking of still images. to appear in *IEEE Journal on Selected Areas in Communication*.

[11] J. R. Hernández, F. Pérez-González, J. M. Rodríguez, and G. Nieto. Coding and synchronization: A boost and a bottleneck for the development of image watermarking. In *Cost #254 Workshop on Intelligent Communications*, L'Aquila, Italy, June 1998.

[12] L. Irwin, G. L. Heileman, C. E. Pizano, C. T. Abdallah, and R. Jordán. The robustness of digital image watermarks. In *Proceedings of the International Conference on Imaging Science, Systems, and Technology*, pages 82–85, Las Vegas, NV, July 1998.

[13] D. Kahn. *The Codebreakers.* MacMillan, New York, 1967.

[14] D. Kahn. The history of steganography. In R. Anderson, editor, *Information Hiding, Springer Lecture Notes in Computer Science*, volume 1174, pages 183–206. Springer-Verlag, 1996.

[15] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX:5–38 (Jan), 161–191 (Feb), 1883.

[16] E. Koch and J. Zhao. Towards robust and hidden image copyright labelling. In *Proceedings of the IEEE Workshop on Nonlinear Signal and Image Processing*, Neos Marmaras, Greece, June 1995.

[17] M. Kutter, F. Jordan, and F. Bossen. Digital signature of color images using amplitude modulation. *Journal of Electronic Imaging*, 7(2):326–332, 1998.

[18] J. S. Lim. *Two-Dimensional Signal and Image Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[19] F. Mintzer, G. W. Braudaway, and A. E. Bell. Opportunities for watermarking standards. *CACM*, 41(7):57–64, 1998.

[20] B. M. Moret. Towards a discipline of experimental algorithmics. In *5th DIMACS Challenge Workshop: Experimental Methodology Day*, Rutgers University, Oct. 1996. `http://www.cs.amherst.edu/~dsj/abstract.moret`.

[21] N. Nikolaidis and I. Pitas. Copyright protection of images using robust digital signatures. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 2168–2171, Atlanta, GA, May 1996. `http://poseidon.csd.auth.gr/signatures/`.

[22] F. A. Petitcolas. Image watermarking: Weaknesses of existing schemes. `http://www.cl.cam.ac.uk/~fapp2/watermarking/image_watermarking/`.

[23] F. A. Petitcolas, R. J. Anderson, and M. G. Kuhn. Attacks on copyright marking systems. In *Proceedings Second International Workshop on Information Hiding, Lecture Notes in Computer Science*, Portland, Oregon, April 1998. Springer-Verlag. `http://www.cl.cam.ac.uk/~fapp2/papers/ih98-attacks/`.

[24] B. Pfitzmann. Information hiding terminology. In *Information Hiding, Springer Lecture Notes in Computer Science*, volume 1174, pages 347–350. Springer-Verlag, 1996.

[25] C. E. Pizano and G. L. Heileman. A GIF watermarking technique based on inveriant measures. In *Proceedings of the International Conference on Imaging Science, Systems, and Technology*, pages 60–67, Las Vegas, NV, July 1998.

[26] M. Schneider and S.-F. Chang. A content-based approach to image signature generation and authentication. In *International Conference on Image Processing*, volume III, pages 227–230, Sept 1996.

[27] B. Schneier. *Applied Cryptography : Protocols, Algorithms and Source Code in C.* John Wiley & Sons, New York, 2nd edition, 1996.

[28] G. J. Simmons. *Contemporary Cryptography : The Science of Information Integrity.* IEEE Press, New York, 1992.

[29] T. G. Stockham, Jr. Image processing in the context of a visual model. *Proceedings of the IEEE*, 60:828–842, 1972.

[30] M. D. Swanson, M. Kobayashi, and A. H. Tewfik. Multimedia data-embedding and watermarking technologies. *Proceedings of the IEEE*, 86(6):1064–1087, 1998.

[31] A. Z. Tirkel, G. A. Rankin, R. M. van Schyndel, W. J. Ho, N. Mee, and C. F. Osborne. Electronic watermark. In *Digital Image Computing, Technology and Applications*, pages 666–673, 1993.

[32] R. G. van Schyndel, A. Z. Tirkel, and C. F. Osborne. A digital watermark. In *International Conference on Image Processing*, volume 2, pages 86–90, Austin, TX, Nov 1994.

[33] R. B. Wolfgang and E. J. Delp. A watermark for digital images. In *International Conference on Image Processing*, pages 219–222, Sept 1996.

[34] R. B. Wolfgang and E. J. Delp. A watermarking technique for digital imagery: Further studies. In *International Conference on Imaging, Systems, and Technology*, pages 279–287, June 1997.

[35] J. Zhao. The unZign homepage. `http://www.altern.org/watermark/`.

A Self Organizing Bin Packing Heuristic

Janos Csirik[1], David S. Johnson[2], Claire Kenyon[3], Peter W. Shor[4], and Richard R. Weber[5]

[1] Department of Computer Sciences
University of Szeged, Szeged, Hungary
csirik@inf.u-szeged.hu
Supported in part by a Fullbright Fellowship, by DIMACS, and by AT&T Labs
[2] AT&T Labs – Research, Room C239,
Florham Park, NJ 07932-0971, USA
dsj@research.att.com
[3] Laboratoire de Recherche en Informatique
Bâtiment 490, Université Paris-Sud, 91405 Orsay Cedex, France
Claire.Kenyon@lri.fr
Supported in part by AT&T Labs
[4] AT&T Labs – Research, Room C237,
Florham Park, NJ 07932-0971, USA
shor@research.att.com
[5] Statistical Laboratory, Cambridge University
16 Mill Lane, Cambridge CB2 1SB, England
rrw1@cam.ac.uk
Supported in part by AT&T Labs

Abstract. This paper reports on experiments with a new on-line heuristic for one-dimensional bin packing whose average-case behavior is surprisingly robust. We restrict attention to the class of "discrete" distributions, i.e., ones in which the set of possible item sizes is finite (as is commonly the case in practical applications), and in which all sizes and probabilities are rational. It is known from [7] that for any such distribution the optimal expected waste grows either as $\Theta(n)$, $\Theta(\sqrt{n})$, or $O(1)$, Our new *Sum of Squares* algorithm (SS) appears to have roughly the same expected behavior in all three cases. This claim is experimentally evaluated using a newly-discovered, linear-programming-based algorithm that determines the optimal expected waste rate for any given discrete distribution in pseudopolynomial time (the best one can hope for given that the basic problem is NP-hard). Although SS appears to be essentially optimal when the expected optimal waste rate is sublinear, it is less impressive when the expected optimal waste rate is linear. The expected ratio of the number of bins used by SS to the optimal number appears to go to 1 asymptotically in the first case, whereas there are distributions for which it can be as high as 1.5 in the second. However, by modifying the algorithm slightly, using a single parameter that is tunable to the distribution in question (either by advanced knowledge or by on-line learning), we appear to be able to make the ratio go to 1 in all cases.

1 Introduction

In the classical one-dimensional bin packing problem, one is given a list $L = \{a_1, ..., a_n\}$ of items, with a size $s(a_i) \in [0, 1]$ for each item in the list. One desires to pack the items into a minimum number of unit-capacity bins, i.e, a partition of the items into a minimum number of subsets such that the sum of the sizes of the items in each subset is one or less. This problem is NP-hard, so much research has concentrated on designing and analyzing polynomial-time approximation algorithms for it, i.e., algorithms that construct packings that use relatively few bins, although not necessarily the smallest possible number. Of special interest have been *on-line* algorithms, i.e., ones that must permanently assign each item in turn to a bin without knowing anything about the sizes or numbers of additional items, a requirement in many applications.

In this paper we concentrate on the average-case behavior of such algorithms. The key metrics with which we are concerned can be defined using the following notation. For a given algorithm A and list L, let $A(L)$ be the number of bins used when A packs L, let $s(L) = \sum_{a \in L} s(a)$, and let $OPT(L) \geq s(L)$ be the optimal number of bins. For a given probability distribution F on item sizes, let $L_n(F)$ be a random n-item list with item sizes chosen independently according to distribution F. Then the *asymptotic expected performance ratio* for A on F is

$$ER_A^\infty(F) \equiv \limsup_{n \to \infty} \left(E \left[\frac{A(L_n(F))}{OPT(L_n(F))} \right] \right)$$

and the *expected waste rate* for A on D is

$$EW_A^n(F) \equiv E\left[A(L_n(F)) - s(L_n(F))\right]$$

Note that because of the low variance of $s(L_n(F))$ for any fixed F, $EW_A^n(F) = o(n)$ implies $ER_A^\infty(F) = 1$ (although not necessarily vice versa). When the context is clear, we will often omit the "(F)" in the above notation.

To date, the most broadly effective practical on-line bin packing algorithm has been *Best Fit* (BF), in which each item is placed in the fullest bin that currently has room for it. Best Fit has been studied under a significant range of distributions. The classical results concern the continuous uniform distributions $U[0, b]$, where item sizes are uniformly distributed over the real interval $[0, b]$. For $b = 1$ we have $EW_A^n = \Theta(n^{1/2}(\log n)^{3/4})$ [13,10], and for $b < 1$ experiments reported in [1,3] suggest that $ER_A^n > 1$, with a maximum value of approximately 1.014, attained for $b \sim 0.79$.

More recently, the behavior of BF has been studied in [3,5,8] for the discrete uniform distributions $U\{j, k\}$, $1 \leq j < k$, in which the allowed item sizes are $1/k, 2/k, ..., j/k$, all equally likely. For $k \geq 3$ and $j = k - 1$, BF's behavior for $U\{j, k\}$ approximately mimics that for $U[0, 1]$, and we have $EW_A^n = \Theta(n^{1/2}(\log k)^{3/4})$ [4]. Moreover, for $j = k - 2$ or $j < \sqrt{2k + 2.25} - 1.5$, much better performance occurs and we have $EW_A^n = O(1)$ [3,8]. However, there appears to exist a constant c such that $ER_A^n > 1$ for $c\sqrt{k} < j \leq k - 3$ and k sufficiently large, with the behavior for $U\{j, k\}$ roughly mimicking that for $U[0, j/k]$.

If running time is no object, algorithms with significantly better expected behavior are possible. Rhee and Talagrand [11] have shown that for any fixed distribution F, there is an algorithm X_F such that $ER^{\infty}_{X_F}(F) = 1$ and such that if $EW^n_{OPT}(F) = o(n)$, then $EW^n_{X_F}(F) = O(n^{1/2}(\log n)^{3/4})$. Moreover, if one is willing to repeatedly solve instances of an NP-hard partitioning problem as part of the algorithm, this level of asymptotic performance can be attained *without* knowing the distribution F in advance, simply by obtaining better and better estimates of it as one goes along, i.e., by learning F on-line [12].

If one restricts attention to discrete distributions, i.e., ones in which the item sizes are all members of a fixed finite set of rational numbers and the corresponding probabilities are all rational numbers as well, even stronger results are possible. For discrete distributions F, the only possible values of $EW^n_{OPT}(F)$ are $\Theta(n)$, $\sqrt{n}$, and $O(1)$, as shown in [7], and for any fixed discrete distribution F there is a linear time on-line algorithm Y_F that has $EW^n_{Y_F}(F) = O(EW^n_{OPT}(F))$. As was the case with the algorithms X_F, the performance of the algorithms Y_F can also be obtained by a single distribution-free algorithm that learns the distribution as it goes along and repeatedly solves NP-hard problems.

Neither of these generic approaches seems practical, and even the distribution-specific algorithms X_F and Y_F are far too complicated to use. They require the (possibly repeated) construction of detailed and distribution-dependent models of multi-bin packings, into the slots of which the incoming items must be separately matched. In this paper we shall present a new and quite simple algorithm *Sum of Squares* (SS) that we conjecture approximately attains the same level of performance as the Y_F for any discrete distribution F, without knowing or attempting to learn F. (We say "approximately" because in some cases where $EW^n_{OPT} = O(1)$, the new algorithm can be shown to yield $EW^n_{SS} = \Omega(\log n)$.) Moreover, although SS like the Y_F's can have $ER^{\infty}_{SS}(F) > 1$ when $EW^n_{OPT} = \Theta(n)$, for each such distribution F, there is a simple-to-construct and practical variant SS_F that we conjecture does yield $ER^{\infty}_{SS_F}(F) = 1$.

For simplicity in what follows, we shall assume that all discrete distributions have been scaled up by an appropriate multiplier B to obtain an equivalent distribution where all item sizes are integers (and for which the bin capacity is B). For example, the scaled $U\{j,k\}$ distributions have item sizes $1, 2, ..., j$ and bin capacity k. This scaling leaves the values of ER^{∞}_{A} unchanged and only affects the constant of proportionality for EW^n_A. In Section 2, we describe SS and its original motivation, and present experimental results comparing it with BF for the distributions $U\{j,k\}$, $1 \leq j < k = 100$. It was these results that first suggested to us SS's surprising effectiveness.

For the $U\{j,k\}$ distributions, the needed comparison values of ER^{∞}_{OPT} and EW^n_{OPT} are already known from theoretical results in [2,3]. For more general classes of discrete distributions, determining these values can be NP-hard. However, as we show in Section 3, the determination can be made by solving a small number of linear programs (LP's) with $O(B^2)$ variables and $O(B)$ constraints, a process that is feasible for B as large as 1000. We use this LP-based approach in Section 4, where we study a generalization of the $U\{j,k\}$ to what we call the *interval distributions* $U\{h..j,k\}$, $1 \le h \le j < k$, in which the bin capacity is k and the item sizes, all equally likely, are the integers s, $h \le s \le j$. Using linear programming, we first determine the values of ER^{∞}_{OPT} and EW^{n}_{OPT} for all such distributions with $k = 19$ or $k = 100$. Then, based on simulations with 10^5, 10^6 and 10^7 items, we estimate the corresponding values for SS. For $k = 19$ we do this for all relevant values of h and j; for $k = 100$ we do this for a challenging subset of the relevant values. In all cases tested our data is consistent with the hypothesis that $EW^{n}_{SS} = O(\max\{\log n, EW^{n}_{OPT}\})$, as claimed. The need for the $\log n$ option is illustrated by tests of the interval distribution $U\{2..3,9\}$, and we describe the conditions under which EW^{n}_{SS} can be proved to grow at least at this rate even though $EW^{n}_{OPT} = O(1)$.

The apparent success of SS far outstrips our original motivation for proposing it. In Section 5 we suggest an intuitive explanation for its behavior that views the operation of SS as a self-organizing process. This explanation is illustrated using detailed measurements of the algorithm's internal parameters under various distributions.

As observed above, SS can have $ER^{\infty}_{SS}(F) > 1$ when $EW^{n}_{SS}(F) = \Theta(n)$. In Section 6, after first presenting a sequence of distributions for which the limiting value of ER^{∞}_{SS} is 1.5, we report on various modification of SS aimed at reducing the value of ER^{∞}_{SS} when $EW^{n}_{OPT} = \Theta(n)$. Most importantly, we show how we can use the results of the LP computation that we performed to determine the value of $EW^{n}_{OPT}(F)$ to devise simple variants SS_F that appear to have $ER^{\infty}_{SS_F} = 1$. This approach can in turn be incorporated into a single polynomial-time "learning" algorithm that we conjecture has an asymptotic expected performance ratio of 1 for all discrete distributions F. Experimental results for the distributions considered in Section 4 are presented that appear to support this conjecture.

We conclude in Section 7 with a preview of the journal version of this paper, which will contain additional experimental and theoretical results, most importantly a proof by Jim Orlin of one our main conjectures.

2 The Sum of Squares Algorithm and $U\{j,k\}$

The sum of squares algorithm works as follows. Assume that our instance has been scaled so that it consists of integer-size items with an integral bin capacity B. Define $N(g)$ to be the number of bins in the current packing with gap g, $1 \le g < B$, where a bin has *gap g* if the items contained in it have total size $B - g$. Initially $N(g) = 0$, $1 \le g < B$. To pack the next item a_i, we place it in a bin (either a currently empty one or a partially full bin with gap at least $s(a_i)$) that will yield the minimum updated value of $\sum_{1 \le g < B} N(g)^2$. If there is a tie, we break it in favor of a candidate bin with the largest current total contents.

A naive motivation for this algorithm (and indeed, the one that led us to propose it in the first place) starts with a fact about Best Fit. This algorithm performs surprisingly well on symmetric discrete distributions, i.e., distributions in which for all sizes s, items of size s and $B - s$ occur with equal probability (e.g., see [3,4]). The reason for this is that typically when the next item to be packed will fit in *some* partially-filled bin, there already exists such a bin whose gap precisely equals the size of the new item. Hence most bins end up being perfectly packed, i.e., having gap 0, and there is very little total waste. How might one extend this behavior to non-symmetric distributions? One idea would be to use items that don't fit perfectly in some current gap to help build and maintain an inventory of gaps so that future items *will* be likely to find a perfect fit. In the absence of other information about the distribution, an initial goal for such an inventory would be to aim for equal numbers of bins for each possible gap. The sum of squares criterion would seem to do this, since given a set of variables whose sum is fixed, their sum of squares is minimized when the values are as close to equal as possible.

This argument does not apply precisely to bin packing, since it is the total number of items in the packing that is fixed, not the total number of bins, but the simplicity of the sum of squares criterion argues in its favor. An item of size s must either (1) start a new bin, in which case the sum of squares increases by $2N(B-s)+1$, (2) perfectly fill an old bin, in which case $N(s)>0$ and the sum of squares decreases by $2N(s)-1$, or (3) go into a bin with gap g, $s<g<B$, in which case the best choice is a g which maximizes $N(g)-N(g-s)$, and the sum of squares decreases by $2(N(g)-N(g-s))-2$. Thus no squares need actually be computed. However, unless we find a way to improve on exhaustive search for (3), we will have a worst-case time of $\Theta(\min(n,B))$ per item as opposed to $\Theta(\log(\min(n,B)))$ for Best Fit. The question is whether this extra running time might provide us better average performance, as hoped.

Let us first consider $U\{j,k\}$ distributions. The behavior of EW_{OPT}^n for these distributions has been characterized in [2,3]: $EW_{OPT}^n = O(1)$ for $1 \le j \le k-2$ and $EW_{OPT}^n = \Theta(\sqrt{n})$ for $j=k-1$. For each of the $U\{j,100\}$ distributions, $1 \le j \le 99$, and each $n \in \{10^5, 10^6, 10^7, 10^8\}$ we computed the average of $SS(L)-s(L)$ and $BF(L)-s(L)$ over a set of random n-item instances (100, 32, 10, and 3 instances respectively) to obtain estimates of EW_{SS}^n and EW_{BF}^n. Instances were generated using the "shift register" random number generator described in [9, pages 171–172]. Previous experiments have shown that for bin packing simulations, this choice is unlikely to introduce significant biases.

These were the first instances we tested after proposing SS, and the results are even better than we had hoped. Whereas Best Fit's performance is as suggested in Section 1, with EW_{BF}^n apparently growing as $\Theta(n)$ for each j, $25 \le j \le 97$, EW_{SS}^n appeared to be $O(1)$ for all j, $1 \le j \le 98$. This is the same range for which $EW_{OPT}^n = O(1)$, and the results for $j=99$ were consistent with $EW_{SS}^n = O(\sqrt{n})$, again the same value as for EW_{OPT}^n. Figure 1 depicts the average waste for SS as a function of j on a log scale, with the averages for each value of n connected in a curve. The log scale is necessary since although $EW_{SS}^n(U\{j,100\})$ does not appear to grow with n for $j \le 98$, it does grow substantially with j. Note that the curves for each of the four values of n all more or less coincide, except possibly for j very close to 100. The solid curve is for

$n = 10^8$, and its higher variability is due to the fact that we tested only three instances of this size.

Table 1 shows the specific averages obtained for $j \in \{24, 25, 60, 97, 98, 99\}$. The first two values of j were chosen as these represent the critical region for Best Fit, where EW^n_{BF} makes a transition from $O(1)$ to $\Theta(n)$. The results for $j = 60$ are typical (except in precise values) of the broad range of j between 25 and 96. The results for $97, 98, 99$ display a critical region for both algorithms, as EW^n_{SS} goes from $O(1)$ to $\Theta(\sqrt{n})$ and EW^n_{BF} goes from $\Theta(n)$ to $O(1)$ to $\Theta(\sqrt{n})$. Our experiments for these last three values of j were extended to include instances with $n = 10^9$, as the rate of convergence is much slower when j is close to k. Although the variance is still sufficiently large for $j = 98$ that we would need substantially more samples if we wanted to get good estimates of the constant to which the expected waste rates are converging, the fact that the ratios are bounded is strongly suggested by the data.

Alg	n	Samples	$j = 24$	25	60	97	98	99
SS	10^5	100	223	223	884	23,350	28,510	34,286
	10^6	32	233	249	894	48,896	70,453	105,277
	10^7	10	212	217	797	64,997	150,291	343,958
	10^8	3	267	213	779	82,378	321,068	1,232,118
	10^9	3				68,719	184,328	3,512,397
BF	10^5	100	78	167	16,088	22,669	24,736	25,532
	10^6	32	76	831	154,460	59,015	77,831	88,258
	10^7	10	102	7,737	1,536,747	213,447	185,870	277,278
	10^8	3	67	75,546	15,340,879	1,800,011	254,235	1,081,251
	10^9	3				17,607,786	187,061	2,757,530

Table 1. Measured waste rates for SS and BF under distributions $U\{j, 100\}$.

As suggested by Figure 1 and Table 1, for fixed n the average waste for SS increases monotonically and fairly smoothly with j, but follows a more adventuresome path for BF. More details on the behavior of BF are reported in [3]. For now it is interesting to note on behalf of Best Fit that although the average waste for BF is enormously larger that that for SS when $25 \le j \le 97$ and EW^n_{BF} appears to grow linearly, the situation is different when EW^n_{BF} is sublinear, as it is for $1 \le j \le 24$ and for $j \in \{98, 99\}$. In these cases its value for fixed n is typically significantly lower than that for EW^n_{SS}, even though the latter has the same growth rate to within a constant factor.

Similar positive results for SS were obtained for $U\{j, k\}$ distributions with other values of k, leading us to conjecture that $EW^n_{SS} = O(EW^n_{OPT})$ for all such distributions. Could something like this conjecture extend to even wider ranges of discrete distributions? A fundamental stumbling block to investigating this question lies in the fact that, in general, determining $EW^n_{OPT}(F)$ given F is an NP-hard problem. The $U\{j, k\}$ distributions are to date the most complicated special cases for which the answers have been obtained analytically. Fortunately, there is a way around this obstacle, at least for moderate values of B.

3 How to Determine EW^n_{OPT}

In order to test the conjecture made in the previous section, we need a way of determining $EW^n_{OPT}(F)$, given a discrete distribution F. It turns out that the slightly simpler question of whether $EW^n_{OPT}(F) = o(n)$ can be formulated as a surprisingly simple linear program related to standard network flow models. Suppose our discrete distribution consists of item sizes s_i, $1 \leq i \leq J$, with the probability that s_i occurs being p_i, and let B be the bin size. Our program will have $J(B+1)$ variables $v(i,g)$, $1 \leq i \leq J$ and $0 \leq g \leq B$, where $v(i,g)$ represents the rate at which items of size s_i go into bins with gap g. The constraints are:

$$\begin{aligned} v(i,g) &= 0, && s_i > g \\ \sum_{g=1}^{B} v(i,g) &= p_i, && 1 \leq i \leq J \\ \sum_{i=1}^{J} v(i,g) &\leq \sum_{j=1}^{J} v(j, g+s_j), && 1 \leq g \leq B-1 \end{aligned}$$

where the value of $v(j, g+s_j)$ when $g + s_j > B$ is taken to be 0 by definition for all j. The first set of constraints says that no item can go into a gap that is smaller than it. The second set says that all items must be packed. The third says that bins with a given gap are created at least as fast as they disappear. The goal is to minimize

$$\sum_{g=1}^{B-1} \left(g \cdot \left(\sum_{j=1}^{J} v(j, g+s_j) - \sum_{i=1}^{J} v(i,g) \right) \right)$$

that is, the rate at which waste space is created.

Let $c(F)$ be the optimal solution value for the above LP, and let $s(F) = \sum_{i=1}^{J} s_i p_i$ be the average item size under F. Then it can be shown based on results in [7] that $ER^\infty_{OPT} = c(F)/s(F)$ and if $c(F) = 0$, then EW^n_{OPT} is either $\Theta(\sqrt{n})$ or $O(1)$.

Moreover, in the latter case, the determination of which growth rate applies can be made by solving J additional LP's, one for each item size: In the LP for item size s_i, we add an additional variable $x \geq 0$, replace the constraint $\sum_{g=1}^{B} v(i,g) = p_i$ by $\sum_{g=1}^{B} v(i,g) = p_i + x$, add a constraint setting the original objective function to 0, and attempt to maximize x. If the optimal value for x is 0 in any of these LP's, then $EW^n_{OPT} = \Theta(\sqrt{n})$, otherwise it is $O(1)$, again by results in [7].

Using the software packages AMPL and CPLEX, we have created an easy-to-use system for generating, solving, and analyzing the solutions of these LP's, given B and a listing of the s_i's and p_i's, or given the parameters h, j, k of an interval distribution. In the next section we describe our results for such distributions.

4 Experiments with General Interval Distributions

In Section 2 we raised the question of whether our conjecture that $EW_{SS}^n = O(EW_{OPT}^n)$ holds for all distributions $U\{j,k\}$ might extend to broader classes of distributions. A natural class to consider is that of the interval distributions $U\{h..j,k\}$, as defined in Section 1.

Before summarizing our experiments with such distributions, however, we must first make admit that they caused us to make a slight modification to our conjecture. It turns out that there are interval distributions with $EW_{OPT}^n = O(1)$ for which EW_{SS}^n appears unavoidably to be $\Omega(\log(n))$. Consider for example $U\{2..3,9\}$, a simple distribution with $EW_{OPT}^n = O(1)$. Note that for $M >> \limsup_{n\to\infty} EW_{OPT}^n$, a sequence of M items of size 2 is likely to create $\Theta(M)$ bins with gap 1 under SS. But gaps of size 1 can never be filled, because there are no items of size 1. Sequences of M consecutive items of size 2 will be rare for large M, but instances with 2^M items can be expected to contain at least one such sequence. This implies that the expected waste will be $\Omega(\log n)$, although the constant of proportionality may be quite small. As an empirical verification, consider Table 2, which summarizes results for runs of SS for instances based on $U\{2..3,9\}$ with n ranging from 10^4 to 10^{10}. Note that the average waste does appear to grow roughly as $\Theta(\log n)$.

n	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}
# Samples	10000	3162	1000	316	100	32	10
Average Waste	7.6	8.6	10.1	10.8	12.1	12.6	14.5
95% Conf. Int.	±0.1	±0.1	±0.2	±0.4	±0.8	±1.0	±1.9

Table 2. Measured average waste for SS under distributions $U\{2..3;9\}$.

We shall thus revise our conjecture about SS and split it into two parts:

Conjecture 1 $EW_{OPT}^n(F) = O(\sqrt{n})$ implies $EW_{SS}^n(F) = O(\sqrt{n})$.
Conjecture 2 $EW_{OPT}^n(F) = O(1)$ implies $EW_{SS}^n(F) = O(\log(n))$.

To test these conjectures, we investigated interval distributions $U\{h..j,k\}$ for two specific values of k, namely $k = 19$ and $k = 100$. For $k = 19$, we tested all pairs $h \le j < k$ with $h \le 9$ using the techniques of the previous section to determine ER_{OPT}^∞ and EW_{OPT}^n and then testing SS and BF on collections of randomly generated instances for the given distribution with $n \in \{10^5, 10^6, 10^7\}$. Pairs h, j with $h \ge 10$ were omitted since for these distributions BF, SS, and OPT all simply place one item per bin and unavoidably have a $\Theta(n)$ expected waste growth. The results are summarized in Table 3.

The "expected" waste rates for OPT in Table 3 are theorems, as determined using the LP's of Section 3, whereas those for SS and BF are for the most part conjectures with which our data is consistent. (We do have proofs for some of the

j	Alg	$h=1$	2	3	4	5	6	7	8	9
18	OPT	$\sqrt{n}$	n	n	n	n	n	n	n	n
	SS	$\sqrt{n}$	n	n	n	n	n	n	n	n
	BF	$\sqrt{n}$	n	n	n	n	n	n	n	n
17	OPT	1	$\sqrt{n}$	n	n	n	n	n	n	n
	SS	1	$\sqrt{n}$	n	n	n	n	n	n	n
	BF	1	$\sqrt{n}$	n	n	n	n	n	n	n
16	OPT	1	n	$\sqrt{n}$	n	n	n	n	n	n
	SS	1	n	$\sqrt{n}$	n	n	n	n	n	n
	BF	n	n	$\sqrt{n}$	n	n	n	n	n	n
15	OPT	1	1	n	$\sqrt{n}$	n	n	n	n	n
	SS	1	$\log n$	n	$\sqrt{n}$	n	n	n	n	n
	BF	n	n	n	$\sqrt{n}$	n	n	n	n	n
14	OPT	1	1	$\sqrt{n}$	n	$\sqrt{n}$	n	n	n	n
	SS	1	$\log n$	$\sqrt{n}$	n	$\sqrt{n}$	n	n	n	n
	BF	n	n	n	n	$\sqrt{n}$	n	n	n	n
13	OPT	1	1	1	n	n	$\sqrt{n}$	n	n	n
	SS	1	$\log n$	$\log n$	n	n	$\sqrt{n}$	n	n	n
	BF	n	n	n	n	n	$\sqrt{n}$	n	n	n
12	OPT	1	1	1	n	n	n	$\sqrt{n}$	n	n
	SS	1	$\log n$	$\log n$	n	n	n	$\sqrt{n}$	n	n
	BF	n	n	n	n	n	n	$\sqrt{n}$	n	n
11	OPT	1	1	1	1	n	n	n	$\sqrt{n}$	n
	SS	1	$\log n$	$\log n$	$\log n$	n	n	n	$\sqrt{n}$	n
	BF	n	n	n	n	n	n	n	$\sqrt{n}$	n
10	OPT	1	1	1	1	n	n	n	n	$\sqrt{n}$
	SS	1	$\log n$	$\log n$	$\log n$	n	n	n	n	$\sqrt{n}$
	BF	1	n	n	n	n	n	n	n	$\sqrt{n}$
9	OPT	1	1	1	n	n	n	n	n	n
	SS	1	$\log n$	$\log n$	n	n	n	n	n	n
	BF	1	n	n	n	n	n	n	n	n
8	OPT	1	1	1	1	n	n	n	n	
	SS	1	$\log n$	$\log n$	$\log n$	n	n	n	n	
	BF	1	n	n	n	n	n	n	n	
7	OPT	1	1	1	1	n	n	n		
	SS	1	$\log n$	$\log n$	$\log n$	n	n	n		
	BF	1	n	n	n	n	n	n		
6	OPT	1	1	1	n	n	n			
	SS	1	$\log n$	$\log n$	n	n	n			
	BF	1	n	n	n	n	n			
5	OPT	1	1	1	n	n				
	SS	1	$\log n$	$\log n$	n	n				
	BF	1	n	n	n	n				
4	OPT	1	1	1	n					
	SS	1	$\log n$	$\log n$	n					
	BF	1	n	n	n					
3	OPT	1	1	n						
	SS	1	$\log n$	n						
	BF	1	n	n						
2	OPT	1	n							
	SS	1	n							
	BF	1	n							

Table 3. Orders of magnitude of the measured waste rates under distributions $U\{h..j, 19\}$.

$h = 1$ entries for BF, in particular those for $j \in \{2, 3, 4, 17, 18\}$ [3,8].) Overall the data is consistent with Conjectures 1 and 2. The values of n tested were not sufficiently large for our measurements to make a convincing case for the $\log n$ growth rates reported for SS in the table; in many cases one might just as well have conjectured $EW_{SS}^n = O(1)$. However, in each of these cases the same sort of argument as was used above for $U\{2..3, 9\}$ applies.

Let us now consider the distributions $U\{h..j, 100\}$. Figure 2 graphically represents the values for EW_{OPT}^n for such distributions, where an entry of "–" represents $O(1)$, an entry of "+" represents $\Theta(\sqrt{n})$, and an entry of "·" represents $\Theta(n)$. Note that this picture appears to be a refinement of the structure apparent in Table 2. Moreover, if one ignores the distinction between –'s and +'s, it is a fairly accurate discretization of the results for the continuous uniform distributions $U[a, b]$, $0 \le a \le b \le 1$, depicted in Figure 5.2 of [6], which partitions the unit square into regions depending on whether $ER_{OPT}^\infty(U[a, b])$ is equal to or greater than 1.

There are far too many $U\{h..j, 100\}$ distributions for us to test SS and BF on them all. We therefore settled for testing isolated examples plus what looks like a challenging slice through Figure 2 – the distributions with $h = 18$. This is a particularly interesting value for h because of the large number of transitions that EW_{OPT}^n makes as j goes from h up to $k - 1$. In all cases, our experimental results were consistent with Conjectures 1 and 2. There is not room here to present the results in detail, but a high-level view of the performance of SS, BF and OPT is presented in Figure 3. This figure graphs the average values $BF(L)/s(L)$ and $SS(L)/s(L)$ over three instances with $n = 10^8$ for each distribution $U\{18..j, 100\}$, $18 \le j \le 99$, and compares these to the value of $\lim_{n\to\infty} E[OPT(L_n)/s(L_n)]$, as determined by the LP's of Section 3. The figure gives a good indication of those j that produce linear waste for each algorithm, i.e., those yielding average values of $A(L)/s(L)$ greater than 1. (For this statistic, the variance between instances is insignificant, so that three instances suffice to give good estimates.) Note that SS has linear waste in precisely those cases where the optimal packing does, although typically the average value is significantly greater. BF has linear waste for all values of j except 82 (the one value that yields a symmetric distribution).

5 *SS* as a Self-Organizing Algorithm

Why does SS do so well? Clearly the idea that SS is simply making sure bins are available into which new items will fit exactly does not suffice as an explanation for performance as good as that we have observed. For example, under $U\{25, 100\}$ there are no items available that will fit exactly into gaps of size exceeding 25, even though the algorithm will tend to produce bins with those gaps if none exist. A possibly better explanation is the following. Because of the sum of squares criterion, the continuing creation of bins with a given gap will be inhibited unless there is some way for bins with that gap to continually disappear. One way for a bin to disappear is for it to have its gap exactly filled;

it then no longer contributes to any of the $N(g)$'s. Another way for a bin to disappear is for it to have its gap reduced to one that already disappears for another reason, for instance if the next *two* items it receives will together fill its gap exactly, or the next three, etc. Thus the algorithm will be driven to favor the creation of precisely those gaps that can (eventually) lead to perfectly packed bins, and the sum of squares criterion is possibly providing a subtle feedback mechanism to maintain the production of the various gaps at the appropriate rates. In other words, it can be thought of as organizing itself for a maximum rate of production of perfectly packed bins.

A likely way to see this organization in action would be to look at the average gap counts $N(g)$ for $1 \le g \le B-1$ as a function of n. Figures 4 through 7 display such *gap count profiles* for $n \in \{100, 200, 800, 50K, 100K, 200K, 400K, 800K\}$ for four different interval distributions. Here, averages are taken over 1,000 separate instances. In each figure, the solid line represents the profile for $n = 800K$. The variety in patterns confirms that SS is indeed organizing itself differently for different distributions, although only the profiles for $U\{18..27, 100\}$ (Figure 4) seem to correlate naturally with the above explanation. Indeed, for $U\{1..19, 100\} = U\{19, 100\}$, all the average counts are 1 or less, so any inhibiting effects of the counts must be fairly subtle. Further study is clearly needed.

6 Improving on the Performance of SS when $EW_{OPT}^n = \Theta(n)$

Extrapolating from the results reported in Sections 2 and 4, one might propose that Conjectures 1 and 2 of Section 4 both hold for *all* discrete distributions. However, although this would imply that $ER_{SS}^{\infty} = 1$ whenever $EW_{OPT}^n = o(n)$, it still allows the possibility that $ER_{SS}^{\infty} > 1$ when $EW_{OPT}^n = \Theta(n)$. We have already remarked that this appears to be the case for several of the $U\{18..j, 100\}$ distributions, as illustrated by Figure 3. A simple distribution for which ER_{SS}^{∞} provably exceeds 1 is $F = U\{2..2, 5\}$, i.e., the distribution in which all items have size 2 and $B = 5$. Here an optimal packing places two items in each bin for a total of $\lceil n/2 \rceil$ bins, whereas SS will create a bin with a single item in it for every two bins that contain two, yielding $ER_{SS}^{\infty} = 1.2$.

For any bin packing algorithm A, let us define

$$\max ER_A^{\infty} \equiv \sup \{ER_A^{\infty}(F) : F \text{ is a discrete distribution}\}$$

Generalizing the above example to the sequence of distributions $U\{2..2, 2m+1\}$, $m \to \infty$, we have $\max ER_{SS}^{\infty} \ge 1.5$.

It is thus natural to search for variants on SS that retain its good behavior when $EW_{OPT}^n = o(n)$, while yielding smaller values of $\max ER_A^{\infty}$. One idea is to add an additional "bin closing" rule to SS. By *closing* a bin we mean declaring it off limits for further items and removing it from the $N(g)$ counts. In SS, the closing rule is simply to close a bin with gap 0, i.e., one that is completely full, as soon as it is created. When $EW_{OPT}^n = \Theta(n)$, even the optimal packing ends up with $\Theta(n)$ incompletely-packed bins, so it might make sense for our algorithm to close some such incompletely-packed bins as well.

We have investigated several variants of SS based on *ad hoc* closing rules that seem to outperform the basic algorithm. However, by far the best results we have obtained via this approach use closing rules that are tailored to the distribution at hand. It appears that for each discrete distribution F there is a variant $SS1_F$, based on a distribution-dependent closing rule, such that $ER^{\infty}_{SS1_F}(F) = 1$.

The closing rule in question is derived from the optimal solution to the LP for F presented in Section 3, Let $v^*(i,g)$ be the value of the variable $v(i,g)$ in this solution, $1 \le i \le j$ and $0 \le g \le B$. For $0 \le g < B$, define

$$r_g \equiv \sum_{j=1}^{J} v(j, g+s_j) - \sum_{i=1}^{J} v(i,g)$$

Note that the r_g are non-negative due to the constraints of the LP, and they can be interpreted as the rate at which bins with final gap g are produced in an optimal packing. Our closing rule is the following: When a bin with gap g is created, check to see if the current number of closed bins with gap g is less than nr_g, where n is the number of items in the current packing. If so, close the bin.

Figure 8 depicts the measured average values of $SS1_F(L)/s(L)$ for the distributions $U\{18..j, 100\}$ and $n \in \{10^5, 10^6, 10^7, 10^8\}$, with the standard numbers of instances for each n. The average values of $A(L)/s(L)$ for each n are linked by a dashed line. As in Figure 3, the solid line connects the values of $\lim_{n\to\infty} E[OPT(L_n)/s(L_n)]$. Note that for each value of j, the average values of $SS1_F(L)/s(L)$ do seem to be converging to the expected ratio for OPT.

Interestingly, if we assume that Conjecture 1 holds for all discrete distributions, there is for each F a (less efficient) variant $SS2_F$ on $SS1_F$ that *provably* will have $ER^{\infty}_{SS2_F}(F) = 1$. Suppose F is as above, and let $R = \sum_{g=1}^{B-1} g \cdot r_g$, the rate at which a unit quantity of waste is produced in an optimal packing. The key idea behind $SS2_F$ is to imagine what would happen if (additional) unit-size items arrived at the same rate. It is not difficult to see that, after normalizing so that the sum of the probabilities again equals 1 (i.e., dividing by $1+R$), one would have a new probability distribution F' for which the solution value $c(F')$ for the LP of Section 3 equals 0. Hence $EW^n_{OPT}(F') = O(\sqrt{n})$. Thus, by Conjecture 1, SS will also have $O(\sqrt{n})$ expected waste for F'. However, note that we can simulate the SS packing of an instance generated according to F' by simply taking a list generated according to F and randomly introducing additional *imaginary* items of size 1 at rate R. The space taken up by these items represents wasted space in the actual packing, but we already knew such waste was unavoidable. The total expected waste is thus $EW^n_{OPT} + O(\sqrt{n})$, and so $ER^{\infty}_{SS2_F}(F) = 1$, as claimed.

Typically $SS2_F$ can be significantly slower than $SS1_F$, since it actually has to pack the imaginary items, and even though standard priority queue data structures can be used to reduce the time for packing them, there may well be a large number of them, especially when B is large. Moreover, the additional running time (and the theoretical guarantee, assuming Conjecture 1) does not seem to be justified by the results. We tested $SS2_F$ on the same test bed of $U\{18..j, 100\}$ instances as we did for $SS1_F$, and the resulting chart of average values of $A(L)/s(L)$ looked essentially identical to Figure 8.

What the *SS2* approach does have going for it is simplicity, and its potential as the basis for a learning algorithm in which one starts with no knowledge of F, but instead learns about it as one goes along. This is the type of approach proposed by [12] in a purely theoretical context. Here however it is quite practical, especially in the context of *SS2*, which only has one distribution-specific parameter to adjust, the rate $r(F)$ at which imaginary 1's are introduced. In our implementation, which we shall call SS^+, we pause to estimate the distribution at ever-increasing intervals, the ith stoppage occurring immediately after $10B * (2^i - 1)$ real items have arrived. During the pause, an estimate F_i of the distribution is derived based on the counts of items of each size received to date and the LP of Section 3 is solved for F_i. The resulting rate $r(F_i)$ is then used until the next stoppage. (Note that by starting with $r = 0$ and waiting for $10B$ items to have arrived before our first adjustment, we guarantee that the algorithm runs in polynomial time.) In tests, SS^+ worked essentially as well as the distribution-specific algorithms $SS1_F$ and $SS2_F$ on the $U\{18..j, 100\}$ distributions, again yielding essentially the same chart as Figure 8.

In practice, SS^+ is much slower than the algorithms that are given F in advance, because of the need to repeatedly construct and solve linear programs. It also requires access to an LP solver. Fortunately, the fact that *SS2* has only one distribution-specific parameter suggests an alternative approach: instead of learning F, why not simply learn $r(F)$? If Conjecture 2 is true for all discrete distributions, we should get good feedback on our estimates of r: If our current r is too small, then waste should clearly grow linearly, whereas if our current r is too big, Conjecture 2 in conjunction with results of [7] says that waste should grow only as $O(\log(n))$. If one waits for enough items to arrive and be packed, one should be able to distinguish between these two cases, and with some complicated algorithm engineering we have managed to implement this approach with some degree of success. The resulting algorithm SSA is too complicated to present in detail here, but Figure 9 summarizes the results for it on the same test bed covered for $SS1_F$ in Figure 8. Although the results are not quite as good as those for SS^+, they do again appear to be converging to the optimal expected waste.

7 Directions of Ongoing Research

Although the experiments reported in this paper have all been for selected interval distributions $U\{h..j, k\}$, we have in fact tested SS on a variety of distributions proposed to us, including various Zipf distributions and randomly-generated discrete distributions, as we shall describe more fully in the journal version of this paper. So far, we have been unable to find counterexamples to either of our two conjectures. In the case of Conjecture 1, this is not surprising, since Jim Orlin, after hearing a talk covering the results and open problems of this paper, has proved Conjecture 1. The journal version of this paper will add Jim as a co-author and include this result. It will also include a detailed proof that the LP's of Section 3 have the properties we claimed for them, as well as more detailed coverage of some of the other theoretical issues we have raised, such as bounds on $\max ER_{SS}^{\infty}$ and an investigation of the worst-case behavior of SS for arbitrary lists.

Another interesting question is how sacrosanct is the exponent of 2 in the definition of SS? What if we tried instead to minimize the sum of cubes, or the sum of 1.5 powers? Preliminary experiments suggest that these variants also satisfy Conjectures 1 and 2, although we should point out that Orlin's proof applies only when the exponent is precisely 2.

References

1. J. L. Bentley, D. S. Johnson, F. T. Leighton, and C. C. McGeoch. An experimental study of bin packing. In *Proceedings of the 21st Annual Allerton Conference on Communication, Control, and Computing*, pages 51–60, Urbana, 1983. University of Illinois.
2. E. G. Coffman, Jr., C. Courcoubetis, M. R. Garey, D. S. Johnson, P. W. Shor, R. R. Weber, and M. Yannakakis. Bin packing with discrete item sizes, part I: Perfect packing theorems and the average case behavior of optimal packings. *SIAM J. Disc. Math.* Submitted 1997.
3. E. G. Coffman, Jr., C. A. Courcoubetis, M. R. Garey, D. S. Johnson, L. A. McGeogh, P. W. Shor, R. R. Weber, and M. Yannakakis. Fundamental discrepancies between average-case analyses under discrete and continuous distributions: A bin packing case study. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 230–240. ACM Press, 1991.
4. E. G. Coffman, Jr., D. S. Johnson, P. W. Shor, and R. R. Weber. Bin packing with discrete item sizes, part IV: Average-case behavior of best fit. In preparation.
5. E. G. Coffman, Jr., D. S. Johnson, P. W. Shor, and R. R. Weber. Markov chains, computer proofs, and best fit bin packing. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 412–421, New York, 1993. ACM Press.
6. E. G. Coffman, Jr. and G. S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms*. Wiley, New York, 1991.
7. C. Courcoubetis and R. R. Weber. Stability of on-line bin packing with random arrivals and long-run average constraints. *Prob. Eng. Inf. Sci.*, 4:447–460, 1990.
8. C. Kenyon, Y. Rabani, and A. Sinclair. Biased random walks, Lyapunov functions, and stochastic analysis of best fit bin packing. *J. Algorithms*, 27:218–235, 1998.
9. D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 2nd Edition, Addison-Wesley, Reading, MA, 1981.
10. T. Leighton and P. Shor. Tight bounds for minimax grid matching with applications to the average case analysis of algorithms. *Combinatorica*, 9:161–187, 1989.
11. W. T. Rhee and M. Talagrand. On line bin packing with items of random size. *Math. Oper. Res.*, 18:438–445, 1993.
12. W. T. Rhee and M. Talagrand. On line bin packing with items of random sizes – II. *SIAM J. Comput.*, 22:1251–1256, 1993.
13. P. W. Shor. The average-case analysis of some on-line algorithms for bin packing. *Combinatorica*, 6(2):179–200, 1986.

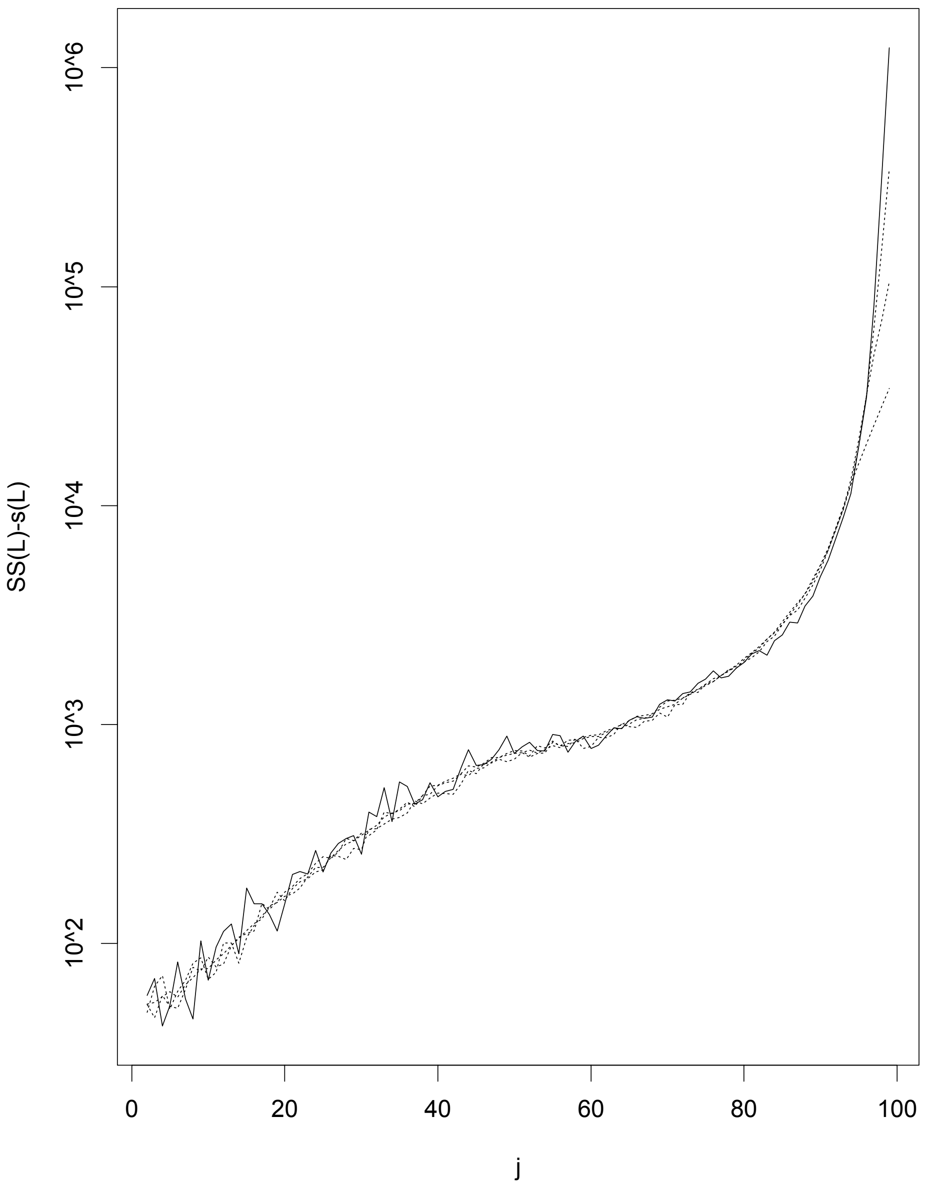


Fig. 1. Average values of $SS(L) - s(L)$ for distributions $U\{j, 100\}$, $1 \leq j \leq 98$ and $n \in \{10^5, 10^6, 10^7, 10^8\}$.

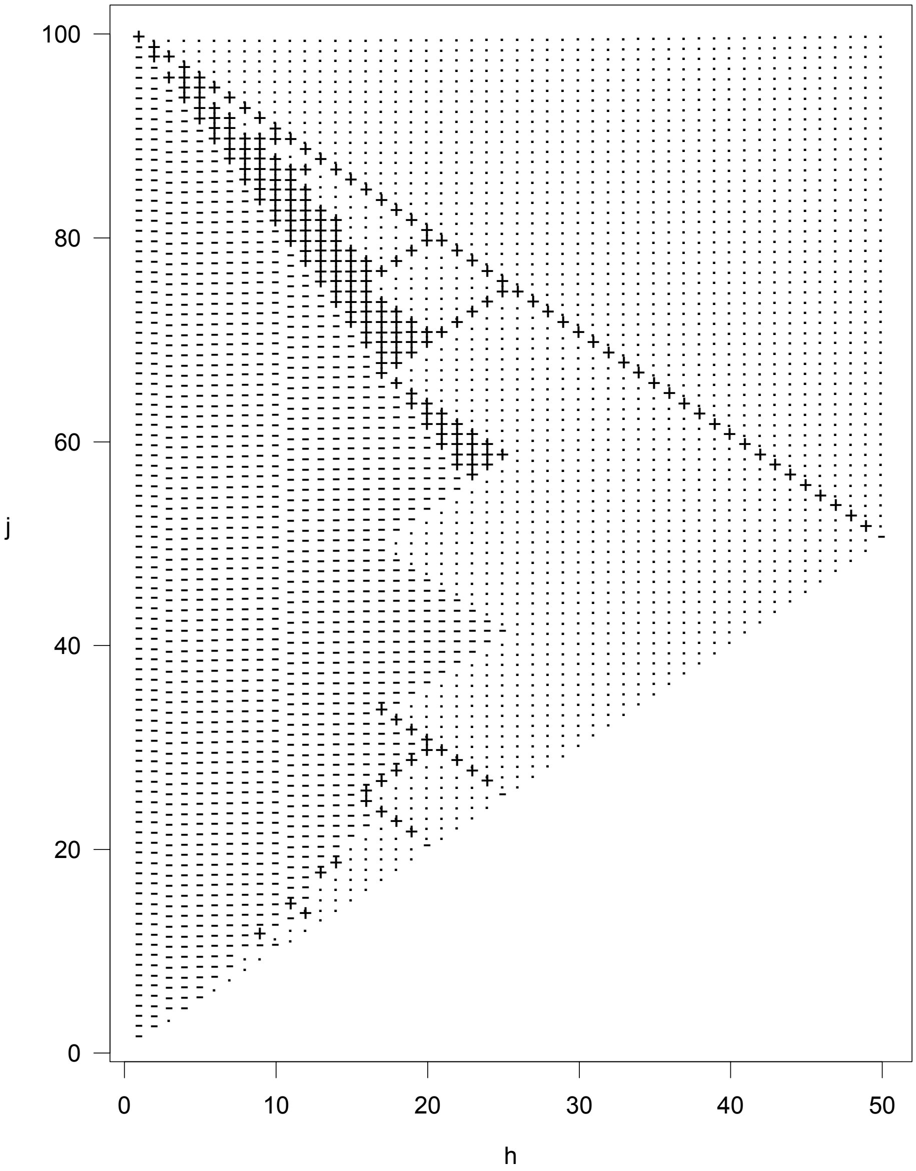


Fig. 2. $EW^n_{OPT}(U\{h..j, 100\})$: "$-$" means $O(1)$, "$+$" means $\Theta(\sqrt{n})$, and "$\cdot$" means $\Theta(n)$.

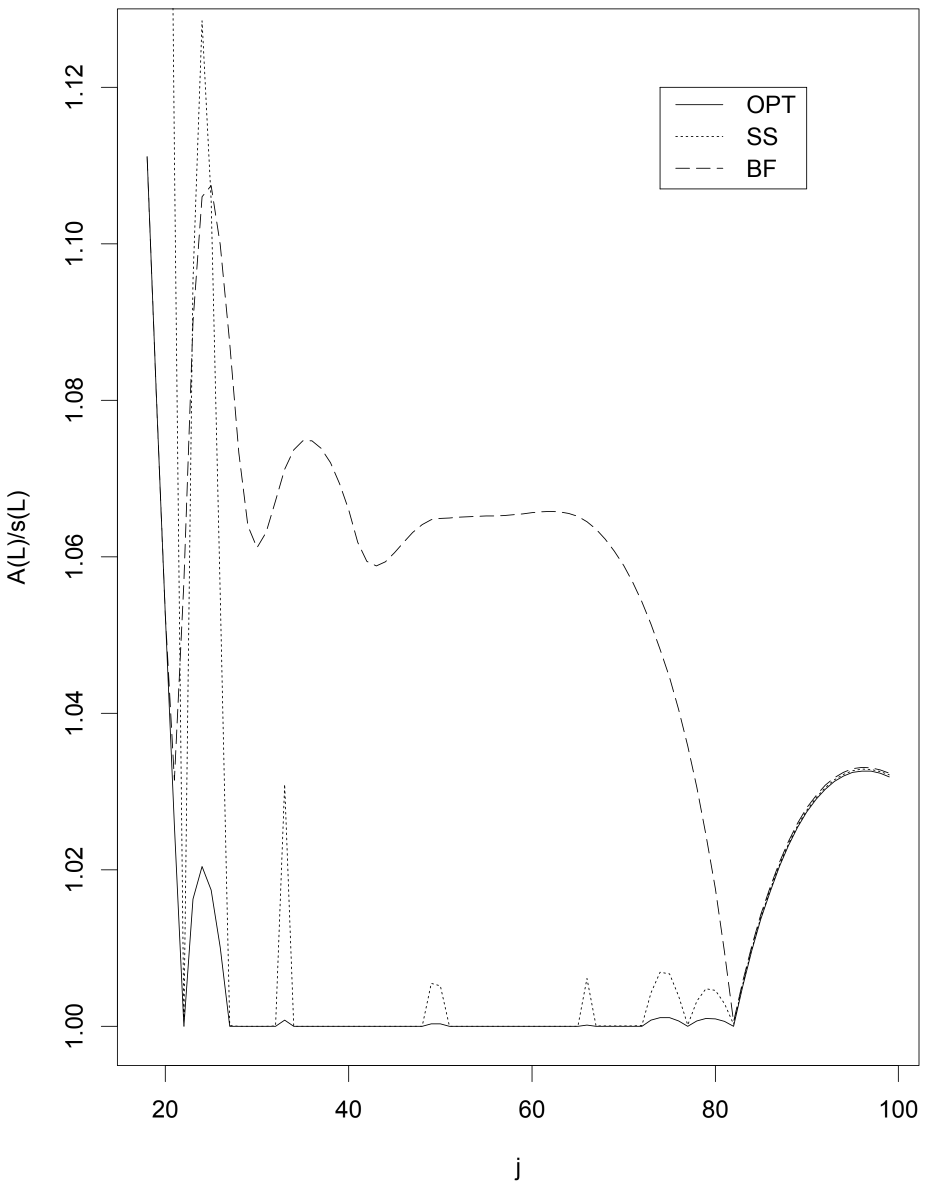


Fig. 3. Average values of $A(L)/s(L)$ for distributions $U\{18..j, 100\}$, $18 \leq j \leq 99$.

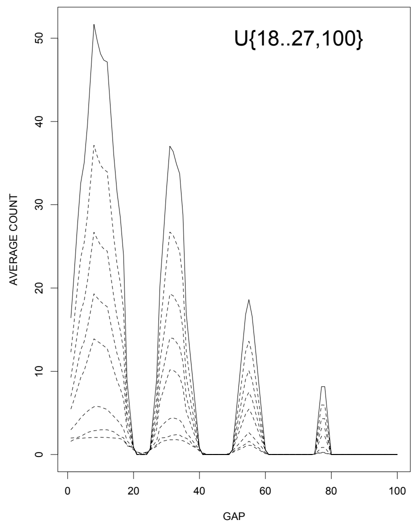


Fig. 4. Gap profiles for $U\{18..27, 100\}$.

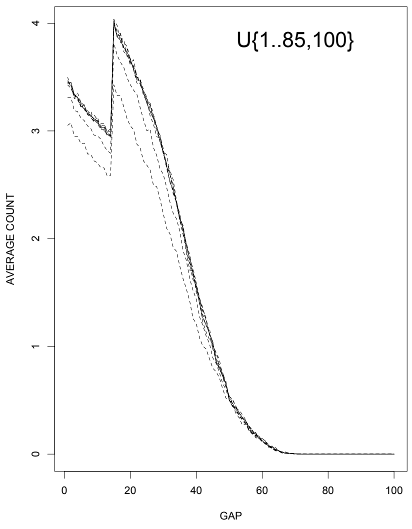


Fig. 6. Gap profiles for $U\{1..85, 100\}$.

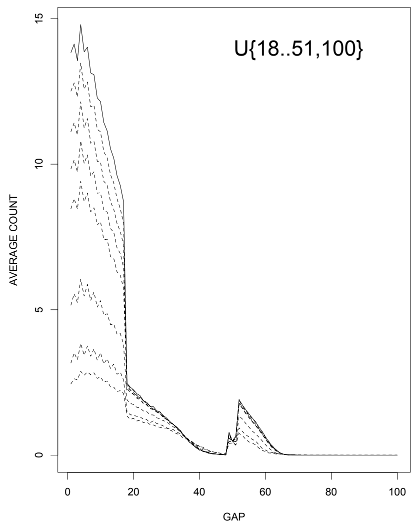


Fig. 5. Gap profiles for $U\{18..51, 100\}$.

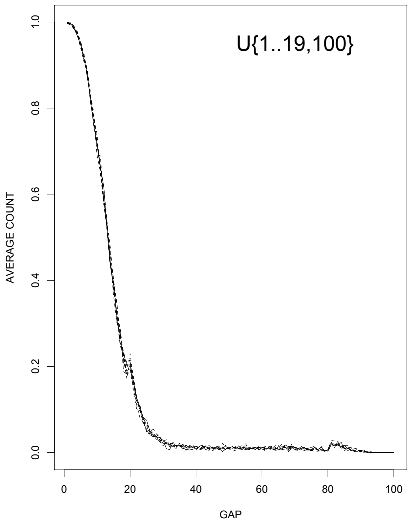


Fig. 7. Gap profiles for $U\{1..19, 100\}$.

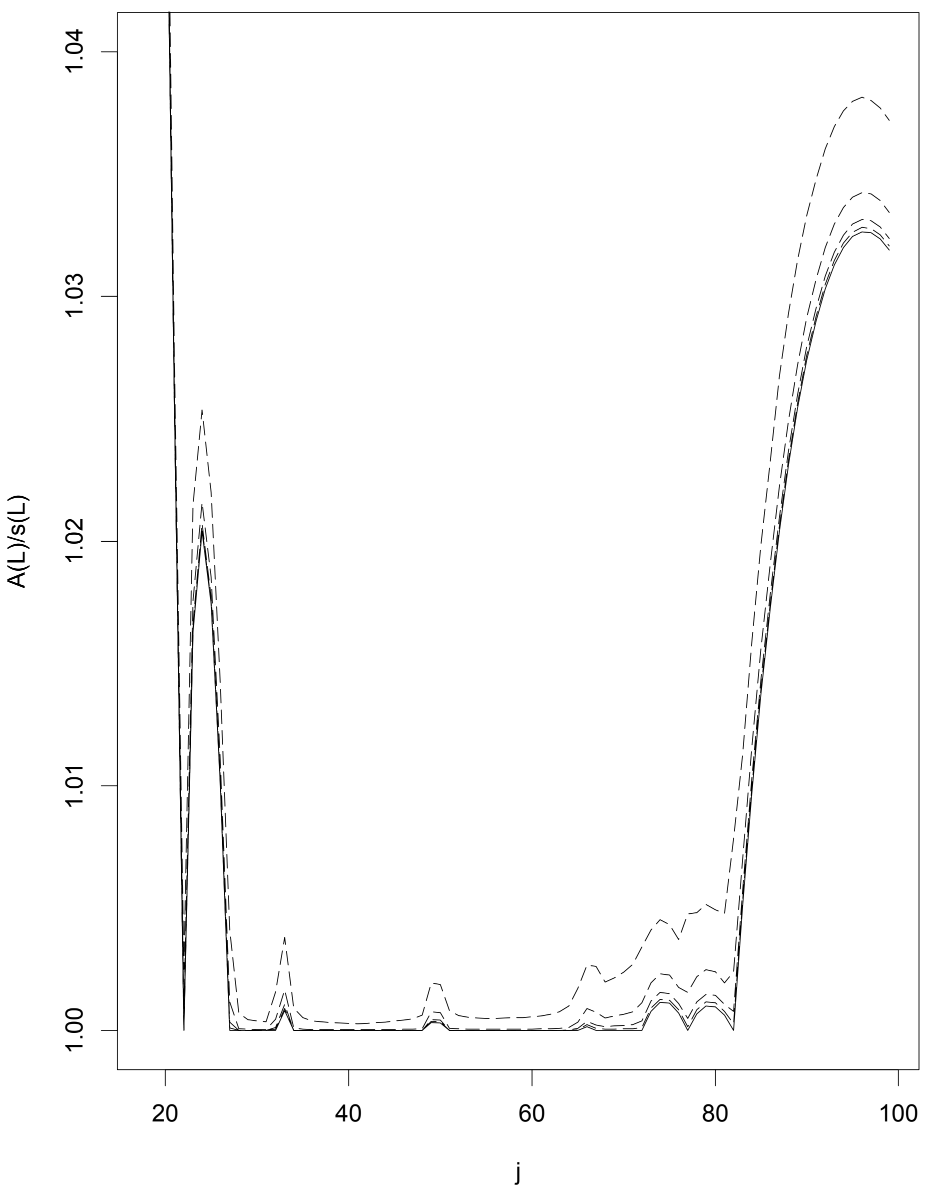


Fig. 8. Average values of $A(L)/s(L)$ for $SS1_F$ when $F = U\{18..j, 100\}$, $18 \leq j \leq 99$ and $n \in \{10^5, 10^6, 10^7, 10^8\}$

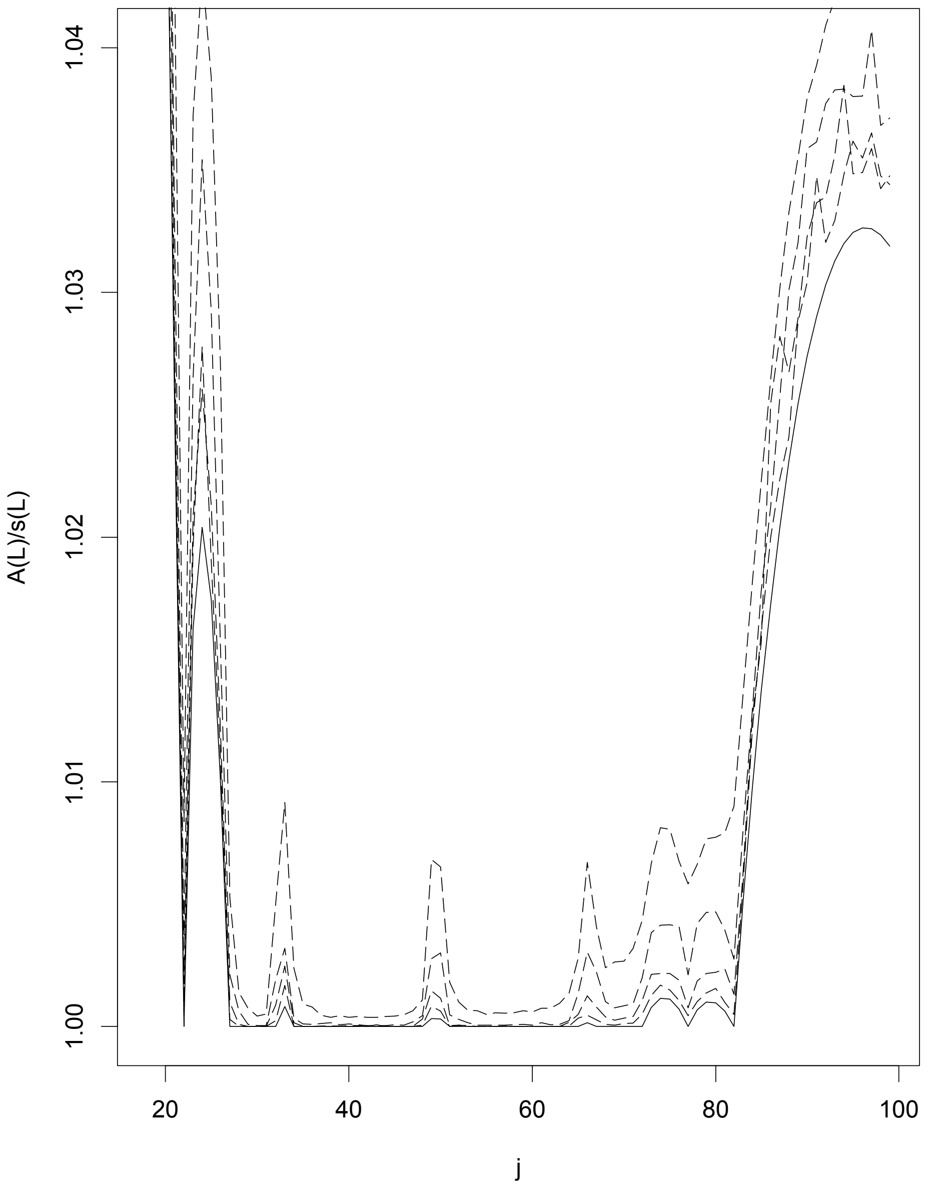


Fig. 9. Average values of $A(L)/s(L)$ for SSA when $F = U\{18..j, 100\}$, $18 \leq j \leq 99$ and $n \in \{10^5, 10^6, 10^7, 10^8\}$

Finding the Right Cutting Planes for the TSP

Matthew S. Levine[⋆]

Massachusetts Institute of Technology
Cambridge, MA 02138, USA
mslevine@theory.lcs.mit.edu
http://theory.lcs.mit.edu/~mslevine

Abstract. Given an instance of the Traveling Salesman Problem (TSP), a reasonable way to get a lower bound on the optimal answer is to solve a linear programming relaxation of an integer programming formulation of the problem. These linear programs typically have an exponential number of constraints, but in theory they can be solved efficiently with the ellipsoid method as long as we have an algorithm that can take a solution and either declare it feasible or find a violated constraint. In practice, it is often the case that many constraints are violated, which raises the question of how to choose among them so as to improve performance. For the simplest TSP formulation it is possible to efficiently find *all* the violated constraints, which gives us a good chance to try to answer this question empirically. Looking at random two dimensional Euclidean instances and the large instances from TSPLIB, we ran experiments to evaluate several strategies for picking among the violated constraints. We found some information about which constraints to prefer, which resulted in modest gains, but were unable to get large improvements in performance.

1 Introduction

Given some set of locations and a distance function, the Traveling Salesman Problem (TSP) is to find the shortest *tour*, i.e., simple cycle through all of the locations. This problem has a long history (see, e.g. [11]) and is a famous example of an NP-hard problem. Accordingly, there is also a long history of heuristics for finding good tours and techniques for finding lower bounds on the length of the shortest tour.

In this paper we focus on one well-known technique for finding lower bounds. The basic idea is to formulate the TSP as an integer (linear) program (IP), but only solve a linear programming (LP) relaxation of it. The simplest such formulation is the following IP:

Variables: x_{ij} for each pair $\{i, j\}$ of cities

Objective: minimize $\sum_{i,j} x_{ij} \cdot distance(i, j)$

[⋆] This work was done while the author was at AT&T Labs–Research.

Constraints:

$$\forall i, j \quad x_{ij} \in \{0, 1\} \tag{1}$$

$$\forall i \quad \sum_j x_{ij} = 2 \tag{2}$$

$$\forall S \neq \emptyset \subset V \quad \sum_{i \in S, j \notin S} \geq 2 \tag{3}$$

The interpretation of this program is that x_{ij} will tell us whether or not we go directly from location i to location j. The first constraints say that we must either go or not go; the second say that we must enter and leave each city exactly once; and the third guarantee that we get one large cycle instead of several little (disjoint) ones. The third constraints are called *subtour elimination constraints*, and will be the main concern of our work.

We relax this IP to an LP in the standard way by replacing the first constraints with $0 \leq x_{ij} \leq 1$. Observe that any solution to the IP will be a solution to the LP, so the optimum we find can only be smaller than the original optimum. Thus we get a lower bound, which is known as the Held-Karp bound[5, 6]. Experimental analysis has shown that this bound is pretty good: for random two dimensional Euclidean instances, asymptotically the bound is only about 0.7% different from the optimal tour length, and for the "real-world" instances of TSPLIB [12], the gap is usually less than 2% [7]. And if the distances obey the triangle inequality, the bound will be at least 2/3 of the length of the optimal tour [13, 15]. It is possible to give more complicated IPs whose relaxations have smaller gaps, but we did not attempt to work with them for reasons that we will explain after we have reviewed the method in more detail.

Observe that it is not trivial to plug this linear program into an LP solver, because there are exponentially many subtour elimination constraints. Nevertheless, even in theory, there is still hope for efficiency, because the ellipsoid method [4] only requires an efficient *separation algorithm*, an algorithm that takes a solution and either decides that it is feasible or gives a violated constraint. For the subtour elimination constraints, if we construct a complete graph with the set of locations as vertices and the x_{ij} as edge weights, it suffices to determine whether or not the *minimum cut* of this graph, the way to separate the vertices into two groups so that the total weight of edges crossing between the groups is minimized, is less than two. If it is, the minimum cut gives us a violated constraint (take the smaller of the two groups as S in the constraint); if not we are feasible. Many algorithms for finding minimum cuts are known, ranging from algorithms that follow from the early work on maximum flows in the 1950s [3] to a recent Monte Carlo randomized algorithm that runs in $O(m \log^3 n)$ time on a graph with m edges and n vertices [9].

Even better, it is possible to find *all* near-minimum cuts (as long as the graph is connected) and thus find all the violated subtour elimination constraints. This leads us to ask an obvious question: which violated constraints do we want to use? When more than one constraint is violated, reporting certain violated constraints before others may lead to a shorter overall running time. The primary goal of this work is to explore this question.

There are several algorithms for finding all near-minimum cuts. They include a flow-based algorithm due to Vazirani and Yannakakis [14], a contraction-based algorithm due to Karger and Stein [10], and a tree-packing-based algorithm due to Karger [9]. We chose to use the Karger-Stein algorithm, primarily because of an implementation we had available that was not far from what we needed. We did not try the others. We believe that the time to find the cuts is small enough compared to the time spent by the LP solver that it would not change our results significantly.

At this point we should justify use of the simplest IP. Our decision was made partly on the grounds of simplicity and historical precedent. A better reason is that with the simplest IP we can use the Karger-Stein minimum cut algorithm and find *all* of the violated constraints. One can construct more complicated IPs that give better bounds by adding more constraints to this simple IP, and there are useful such constraints that have separation algorithms, but for none of the sets of extra constraints that people have tried is it known how to efficiently find all of the violated constraints, so it would be more difficult to determine which constraints we would like to use. It may still be possible to determine which constraints to use for a more complicated IP, but we leave that as a subject for further research. Note that the constraints of the more complicated IPs include the constraints of the simple IP, so answering the question for the simple IP is a reasonable first step towards answering the question for more complicated IPs.

We found that it is valuable to consider only sets of small, disjoint constraints. Relatedly, it seems to be better to fix violations in small areas of the graph first. This strategy reduces both the number of LPs we have to solve and the total running time. We note that it is interesting that we got this improvement using the Karger-Stein algorithm, because in the context of finding one minimum cut, experimental studies have found that other algorithms perform significantly better [2, 8]. So our results are a demonstration that the Karger-Stein algorithm can be useful in practice.

The rest of this paper is organized as follows. In Sect. 2 we give some important details of the implementations that we started with. In Sect. 3 we discuss the constraint selection strategies that we tried and the results we obtained. Finally, in Sect. 4 we summarize the findings and give some possibilities for future work.

2 Starting Implementation

In this section we give some details about the implementations we started with. We will discuss our attempts at improving them in Sect. 3. For reference, note that we will use n to denote the number of cities/nodes and will refer to the total edge weight crossing a cut as the *value* of the cut.

2.1 Main Loop

The starting point for our work is the TSP code `concorde` written by Applegate, Bixby, Chvátal, and Cook [1]. This code corresponds to the state of the art in lower bound computations for the TSP. Of course it wants to use far more that the subtour elimination constraints, but it has a mode to restrict to the simple IP. From now on, when we refer to "the way `concorde` works", we mean the way it works in this restricted mode. We changed the structure of this code very little, mainly just replacing the algorithm that finds violated constraints, but as this code differs significantly from the theoretical description above, we will review how it works.

First of all, `concorde` does not use the ellipsoid method to solve the LP. Instead it uses the simplex method, which has poor worst-case time bounds but typically works much better in practice. Simplex is used as follows:

1. start with an LP that has only constraints (1) and (2)
2. run the *simplex method* on the current LP
3. find some violated subtour elimination constraints and add them to the LP; if none terminate
4. repeat from 2

Observe that the initial LP describes the fractional 2-matching problem, so `concorde` gets an initial solution by running a fractional 2-matching code rather than by using the LP solver.

Second, it is important to pay attention to how cuts are added to the LP before reoptimizing. There is overhead associated with a run of the LP solver, so it would be inefficient to add only one cut at a time. On the other side, since not many constraints will actually constrain the optimal solution, it would be foolish to overwhelm the LP solver with too many constraints at one time. Notice also that if a constraint is not playing an active role in the LP, it may be desirable to remove it so that the LP solver does not have to deal with it in the future. Thus `concorde` uses the following general process for adding constraints to the LP, assuming that some have been found somehow and placed in a list:

1. go through the list, picking out constraints that are still violated until 250 are found or the end of the list is reached
2. add the above constraints to the LP and reoptimize
3. of the newly added constraints, only keep the ones that are in the basis

Thus at most 250 constraints are added at a time, and a constraint only stays in the LP if it plays an active role in the optimum when it is added. After a constraint is kept once, it is assumed to be sufficiently relevant that it is not allowed to be thrown away for many iterations. In our case, for simplicity, we never allowed a kept cut to leave again. (Solving this simple IP takes few enough iterations that this change shouldn't have a large impact.)

Third, just as it is undesirable to work with all of the constraints at once, it is also undesirable (in practice) to work with all of the variables at once, because most of them will be 0 in the optimal solution. So there is a similar process of selecting a few of the variables to work with, solving the LP with only those variables, and later checking to see if some other variables might be needed, i.e., might be non-zero in an optimal solution. The initial sparse graph comes from a greedy tour plus the 4 nearest neighbors with respect to the reduced costs from the fractional 2-matching.

Finally, it is not necessary to find minimum cuts to find violated constraints. If the graph is disconnected, then each connected component defines a violated constraint. In fact, any set of connected components defines a violated constraint, giving a number of violated constraints exponential in the number of components, so `concorde` only considers the constraints defined by one component. This choice makes sense, because if each connected component is forced to join with another one, we make good progress, at least halving the number of components.

Another heuristic `concorde` uses is to consider the cuts defined by a segment of a pretty good tour, i.e, a connected piece of a tour. `concorde` uses heuristics to find a pretty good tour at the beginning, and the authors noticed that cuts they found often corresponded to segments, so they inverted the observation as a heuristic. We mention this heuristic because it is used in the original implementation, which we compare against, but we do not use it in our modified code.

Finally, a full pseudo-code description of what the starting version of `concorde` does:

```
find an initial solution with a fractional 2-matching code
build the initial sparse graph, a greedy tour + 4 nearest in fractional
    matching reduced costs
do
    do
        add connected component cuts (*)
        add segment cuts (*)
        if connected, add flow cuts (*)
        else add connected component cuts
        if no cuts added OR a fifth pass through loop,
            check 50 nearest neighbor edges to see if they need to be added
    while cuts added OR edges added
    check all edges to see if they need to be added
while edges added
```

Note that the lines marked with (*) are where we make our changes, and the adding of cuts on these lines is as above, which includes calling the LP solver.

2.2 Karger-Stein Minimum Cut Algorithm

The starting implementation of the Karger-Stein minimum cut algorithm (KS) is the code written by Chekuri, Goldberg, Karger, Levine, and Stein [2]. Again, we did not make large modifications to this code, but it already differs significantly from the theoretical description of the algorithm. The original algorithm is easy to state:

```
if the graph has less than 7 nodes, solve by brute force
repeat twice:
    repeat until only $n/\sqrt{2}$ nodes remain:
        randomly pick an edge with probability proportional to edge weight
            and contract the endpoints
    run recursively on the contracted graph
```

Contracting two vertices means merging them and combining the resulting parallel edges by adding their weights. It is easy to see that contraction does not create any cuts and does not destroy a cut unless nodes from opposite sides are contracted.

The idea of the algorithm is that we are not so likely to destroy the minimum cut, because by definition there are relatively few edges crossing it. In particular, the random contractions to $n/\sqrt{2}$ nodes give at least a 50% chance of preserving the minimum cut. Thus if we repeat the contraction procedure twice, there is a reasonable chance that the minimum cut is preserved in one of the recursive calls, so there is a moderate ($\Omega(1/\log n)$) chance that the minimum cut is preserved in one of the base cases. By repeating the entire procedure $O(\log n)$ times, the success probability can be improved to $1 - 1/n$.

Of course we are not just interested in minimum cuts; we want all of the cuts of value less than 2. We can find these by doing fewer contractions at a time, that is, leaving more than $n/\sqrt{2}$ nodes. This modification makes cuts that are near-minimum (which a cut of value 2 hopefully is) also have a good chance of being found.

As KS is a Monte-Carlo algorithm (there is no easy way to be sure it has given the right answer) and we did not want to affect the correctness of `concorde`, whenever our implementation of KS found no cuts of value less than two, we always double-checked with `concorde`'s original flow-based cut finding algorithm. Later, when we refer to implementations that use only KS to find cuts, we will really mean that they always use KS, unless KS fails to find any cuts. Typically, by the time KS failed to find any cuts, we were either done or very close to it, so it is reasonable to ignore the fact that a flow algorithm is always still there.

An important thing to notice in KS is that we have two parameters to play with. One is how many contractions we do at a time, which governs the depth and success probability of the recursion. The other is how many times we run the whole procedure. In order to achieve a specific success probability, we can only choose one of these. If we are willing to do away with the theoretical analysis and make a heuristic out of this algorithm, we can choose both. Since we do have a correctness check in place, making KS a heuristic is a reasonable thing to do. In particular, we started with the first parameter set so that we would find all cuts of value less than two with probability $\Omega(1/\log n)$, and the second parameter set to three. We found that three iterations was sufficient to typically find a good fraction (approximately two thirds) of the cuts, and this performance seemed good enough for our purposes. Later, after we had gathered some information about the cuts, we worried about reducing the time spent by KS and set the first parameter such that we would only find cuts of value less than one with probability $\Omega(1/\log n)$. Note that regardless of the setting of the first parameter, the code will always report all the cuts of value less than two that it finds. So the later version of the code does not disregard higher value cuts as a result of changing the parameter, it merely has a lower chance of finding them.

The implemented version chooses edges for contraction in one pass, rather than one at a time. This modification can allow more contractions under certain good circumstances, but can cause trouble, because it is possible to get unlucky and have the recursion depth get large. See [2] for a thorough discussion of the implemented version. A change we made here was to repeat the contraction step if nothing gets contracted; while this change is an obvious one to make, it likely throws off the analysis a bit. Since we will make the algorithm a heuristic anyway, we chose not to worry about exactly what this little change does. Note that we had to disable many of the Padberg-Rinaldi heuristics used in the starting implementation, because they only work if we are looking for minimum cuts, not near-minimum cuts.

We also had to make some modifications so that we could run on disconnected graphs. If the graph is disconnected, there can be exponentially many minimum cuts, so we cannot hope to report them all. At first we worked around the problem of disconnected graphs by forcing the graph to be connected, as the starting implementation of `concorde` does. However, later in the study we wanted to try running KS earlier, so we had to do something about disconnected graphs. Our new workaround was to find the connected components, report those as cuts, and run KS in each component. This modification ignores many cuts, because a connected component can be added to any cut to form another cut of the same value. We chose this approach because 1) we had to do something, and 2) other aspects of our experiments, which we describe shortly, suggest that this approach is appropriate.

One last modification that we made was to contract out paths of edges of weight one at the beginning. The point of this heuristic is that if any edge on a path of weight one is in a small cut, then every such edge is in a small cut. So we would find many cuts that were very similar. Our experiments suggested that it is more useful to find violated constraints that are very different, so we used this heuristic to avoid finding some similar cuts.

3 Experiments and Results

3.1 Experimental Setup

Our experiments were run on an SGI multiprocessor (running IRIX 6.2) with 20 196 Mhz MIPS 10000 processors. The code was not parallelized, so it only ran on one processor, which it hopefully had to itself. The machine had 6 Gb of main memory and 1 Mb L2 cache. The code was compiled with SGI cc 7.2, with the -O2 optimization option and set to produce 64 bit executables. CPLEX 5.0 was used as the LP solver.

Several processes were run at once, so there is some danger that contention for the memory bus slowed the codes down, but there was nothing easy we could do about it, and we do not have reason to believe it was a big problem. In any case, all the codes were run under similar conditions, so the comparisons should be fair.

We used two types of instances. One was random two dimensional Euclidean instances generated by picking points randomly in a square. The running times we report later are averages over 3 random seeds. The second type of instance was "real-world", from TSPLIB. We tested on rl11849, usa13509, brd14051, pla33810, and pla85900.

3.2 Observations and Modifications

We started our study by taking `concorde`, disabling the segment cuts, and substituting KS for the flow algorithm. So the first time the algorithm's behavior changed was after the graph was connected, when KS was first called. At this point we gathered some statistics for a random 10000 node instance about the cuts that were found and the cuts that were kept. Figure 1 shows two histograms comparing cuts found to cuts kept. The first is a histogram of the size of the cuts found and kept, that is, the number of nodes on the smaller side. The second shows a similar histogram of the value of the cuts found and kept. Note that the scaling on the Y-axis of these histograms is unusual.

These histograms show several interesting features. First, almost all of the kept cuts are very small—fewer than 100 nodes. The found cuts are also very biased towards small cuts, but not as much. For example, many cuts of size approximately 2000 were found, but none were kept. A second interesting feature is that the minimum cut is unique (of value approximately .3), but the smallest kept cut is of value approximately .6, and most of the kept cuts have value one. This observation immediately suggests that it is not worthwhile to consider only minimum cuts, because they are few in number and not the cuts you want anyway. Furthermore, it appears that there is something special about cuts of value one, as a large fraction of them are kept.

To try to get a better idea of what was going on, we took a look at the fractional solution. Figure 2 shows the fractional 2-matching solution for a 200 node instance, which is what the cut finding procedures are first applied to. Not surprisingly, this picture has many small cycles, but the other structure that appears several times is a path, with some funny edge weights at the end that allow it to satisfy the constraints. The presence of these structures suggests looking at biconnected components in the graph induced by non-zero weight edges. A long path is in some sense good, because a tour looks locally like a path, but it is important that the two ends meet, which is to say that the graph must not only be connected but biconnected.

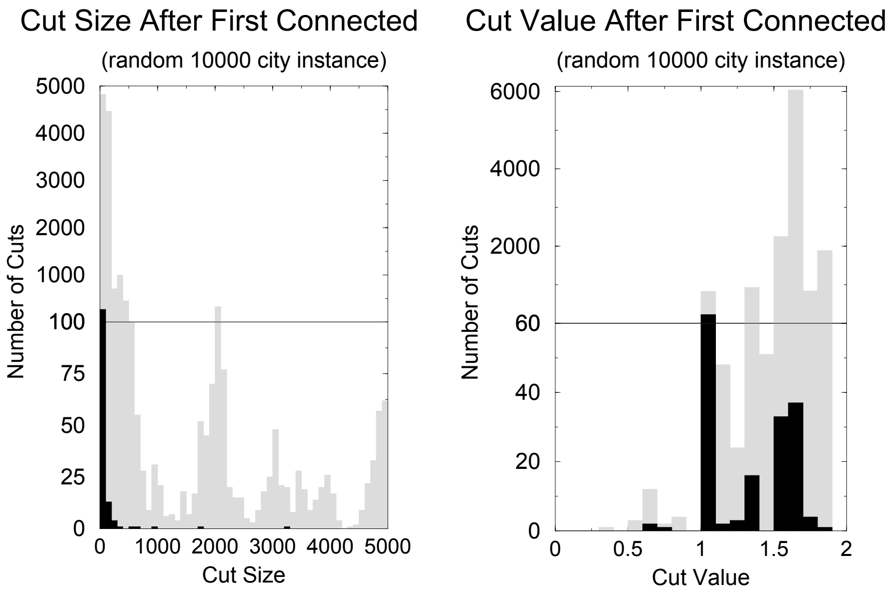


Fig. 1. Histograms showing properties of cuts found as compared to cuts kept. Gray bars represent found cuts and black bars represent kept cuts. Note that there is unusual scaling on the Y-axes.

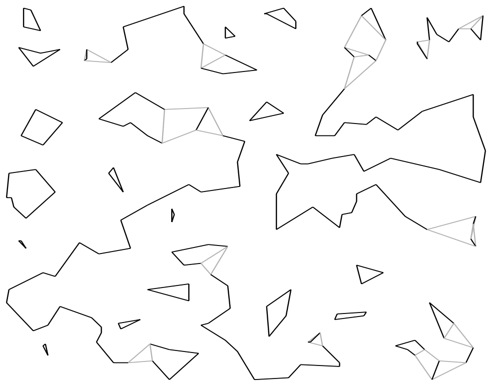


Fig. 2. Picture of a fractional 2-matching (initial solution) for a 200 node instance. Edge weights are indicated by shading, from white= 0 to black= 1.

We tried various ways of incorporating biconnected components before finding a method that worked well. There are two issues. One is when to try to find biconnected components. Should we wait until the graph is connected, or check for biconnectivity in the connected components? The second issue is what cuts to report. Given a long path, as above, there is a violated constraint corresponding to every edge of the path. Or stated more generally, there is a violated constraint corresponding to every articulation point (a node whose removal would disconnect the graph). Our first attempt was to look for biconnected components only once the graph was connected, and to report all the violated constraints. This approach reduced the number of iterations of the main loop, but took more time overall. Running biconnected components earlier reduced the number of iterations further, but also took too long.

So to reduce the number of cuts found, we modified the biconnected components code to report only constraints corresponding to biconnected components that had one or zero articulation points. (Note that a biconnected component with 0 articulation points is also a connected component.) The idea behind this modification is the same idea that was used to report constraints based on the connected components. In that context, we said that it made sense to consider only the constraints corresponding to the individual components, thus picking out small, disjoint constraints. Likewise, taking biconnected components with one or zero articulation points picks out small, disjoint constraints. This use of biconnected components proved valuable; it both reduced the number of iterations of the outer loop and reduced the overall running time. Accordingly, it is only this version that we give data for in the results section.

Our experience with KS was similar. Using it to find all of the violated constraints turned up so many that even though we saved iterations, the total running time was far worse. And it seemed to be faster to run KS right from the beginning, not waiting for the graph to become connected. So we generalized the idea above. We only report the smallest disjoint cuts; that is, we only report a cut if no smaller (in number of nodes) cut shares any nodes with it. It should be easy to see that the rules given above for selecting cuts from connected or biconnected components are special cases of this rule. So our eventual implementation with KS uses it right from the beginning, and always reports only the smallest cuts. Note that our handling of disconnected graphs is consistent with finding smallest cuts. Notice also that reporting only smallest cuts means we do not introduce a constraint that will force a connected component to join the other connected components until the component itself satisfies the subtour elimination constraints. This choice may seem foolish, not introducing a constraint we could easily find, but what often happens is that we connect the component to another in the process of fixing violations inside the component, so it would have been useless to introduce the connectivity constraint.

In this vein, we discovered that it could actually be harmful to force the graph to be connected before running KS. It would not be surprising if the time spent getting the graph connected was merely wasted, but we actually saw instances where the cut problem that arose after the graph was connected was much harder than anything KS would have had to deal with if it had been run from the beginning. The result was that finding connected components first actually cost a factor of two in running time.

Note that the implementation of selecting the smallest cuts was integrated into KS. We could have had KS output all of the cuts and then looked through them to pick out the small ones, but because KS can easily keep track of sizes as it contracts nodes, it is possible to never consider many cuts and save time.

There is also one observation that we failed to exploit. We noticed that the value histogram shows some preference for keeping cuts that have small denominator value; most kept cuts had value one, then several more had value 3/2, a few more had 4/3 and 5/3, etc. Unfortunately, we did not come up with a way to exploit this observation. We tried sorting the cuts by denominator before adding them to the LP, hoping that we would then get the cuts we wanted first and be able to quickly discard the others, but were unable to get a consistent improvement this way. Even when there was an improvement, it was not clear whether it was entirely due to the fact that cuts of value one got added first.

3.3 Results

We present our data in the form of plots, of which there are three types. One reports total time, another reports the number of LPs solved, and the third only considers the time to add cuts (as described above), which counts only the time to process the lists of cuts and reoptimize, but not the time to find the cuts. The total time includes everything: the time to add cuts, the time to find them, the time to add edges, the time to get an initial solution, etc. We also have two classes of plots for our two classes of inputs, random instances and TSPLIB instances. All times are reported relative to the square of the number of cities, as this function seems to be the approximate asymptotic behavior of the implementations. More precisely, the Y-axes of the plots that report times are always $1000000 * (\text{time in seconds})/n^2$. This scale allows us to see how the algorithms compare at both large and small problem sizes. Note also that the X-axes of the TSPLIB plots are categorical, that is, they have no scale. Table 1 summarizes the implementations that appear in the plots.

Short name	Description
starting point	original `concorde` implementation
w/o segments	original with segment cuts disabled
biconnect	using smallest biconnected components instead of connected
KS	all cut finding done by KS
KS1	all cut finding done by KS, probabilities set for cuts of value < 1

Table 1. Summary of the implementations for which we report data

There are several things to notice about the plots. First, looking at the time to add cuts for random instances (Fig. 3), we see that using either biconnected components or KS consistently improves the time a little bit. Furthermore, using the version of KS with looser probabilities causes little damage. Unfortunately, the gain looks like it may be disappearing as we move to larger instances. Looking at the number of LPs that we had to solve shows a clearer version of the same results (Fig. 4). Looking at the total time (Fig. 5), we see the difference made by adjusting the probabilities in KS. The stricter version of KS is distinctly worse than the original `concorde`, whereas the looser version of KS is, like the version with biconnected components, a bit better. Looking at the total time is looks even more like our gains are disappearing at larger sizes.

The story is similar on the TSPLIB instances (Figs. 6,8,7) Biconnected components continue to give a consistent improvement. KS gives an improvement on some instances, but not on the PLA instances.

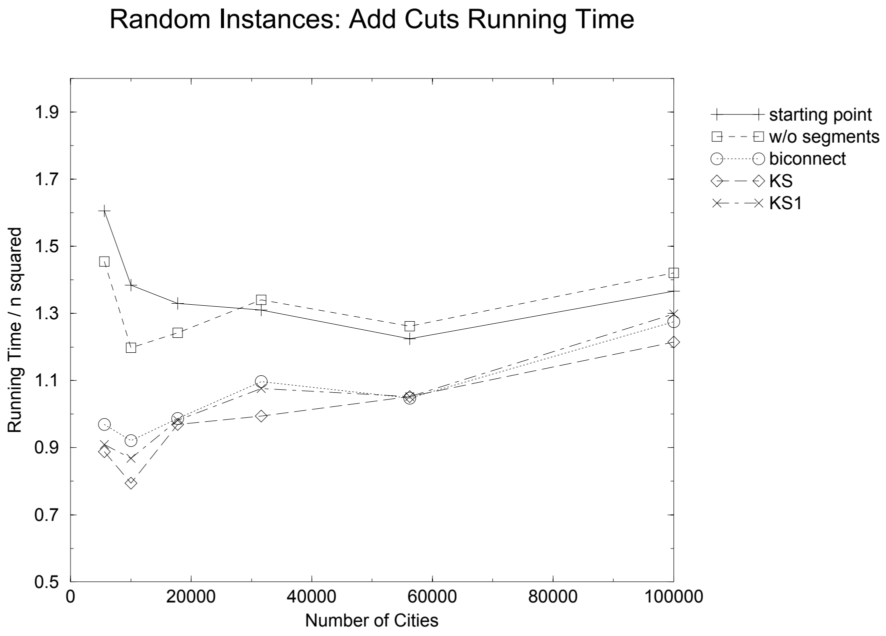


Fig. 3.

4 Conclusion

The strategy of looking for the smallest cuts seems to be a reasonable idea, in that it reduces the number of iterations and improves the running time a bit, but the gain is not very big. It also makes some intuitive sense that by giving

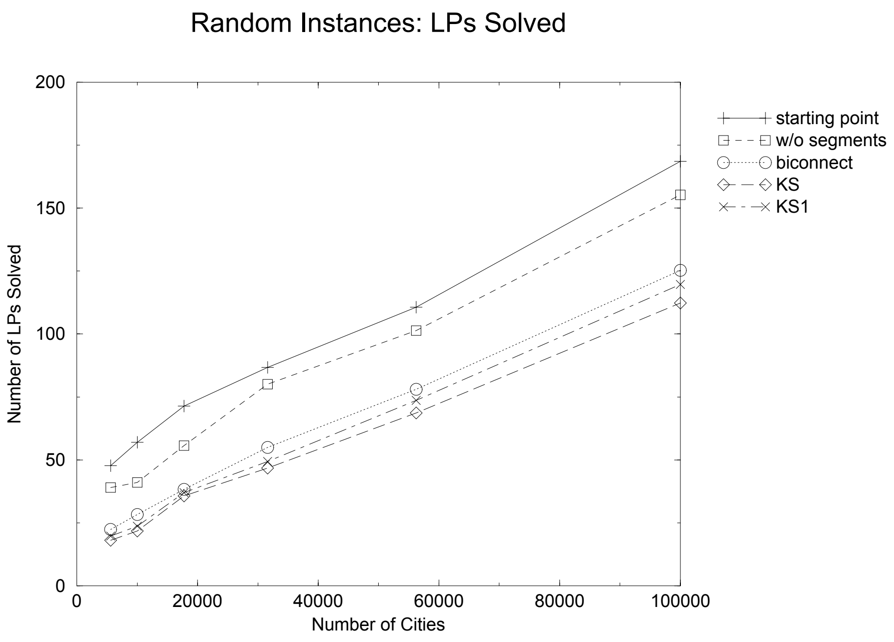

Random Instances: LPs Solved
Number of LPs Solved
Number of Cities
0
50
100
150
200
0
20000
40000
60000
80000
100000
starting point
w/o segments
biconnect
KS
KS1

Fig. 4.

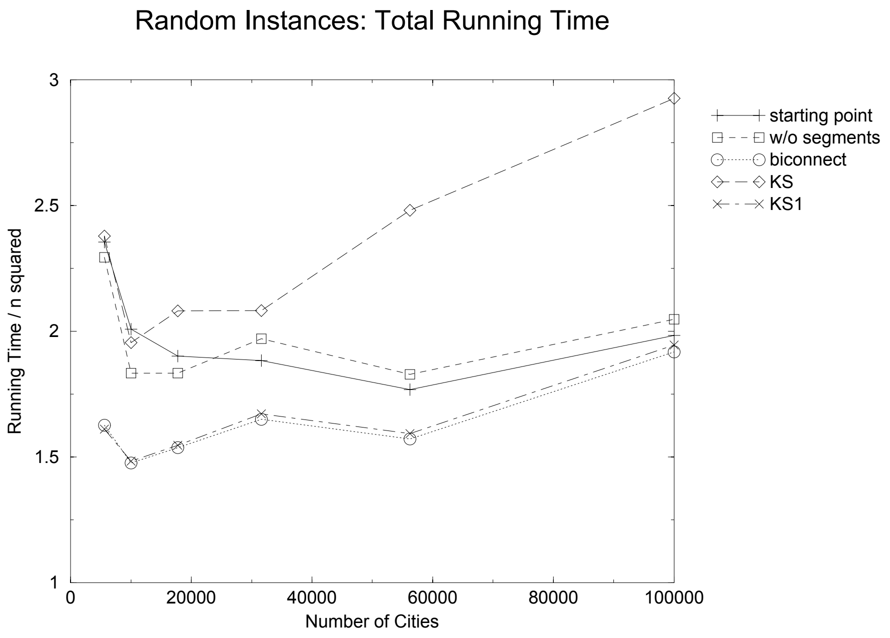

Random Instances: Total Running Time
Running Time / n squared
Number of Cities
1
1.5
2
2.5
3
0
20000
40000
60000
80000
100000
starting point
w/o segments
biconnect
KS
KS1

Fig. 5.

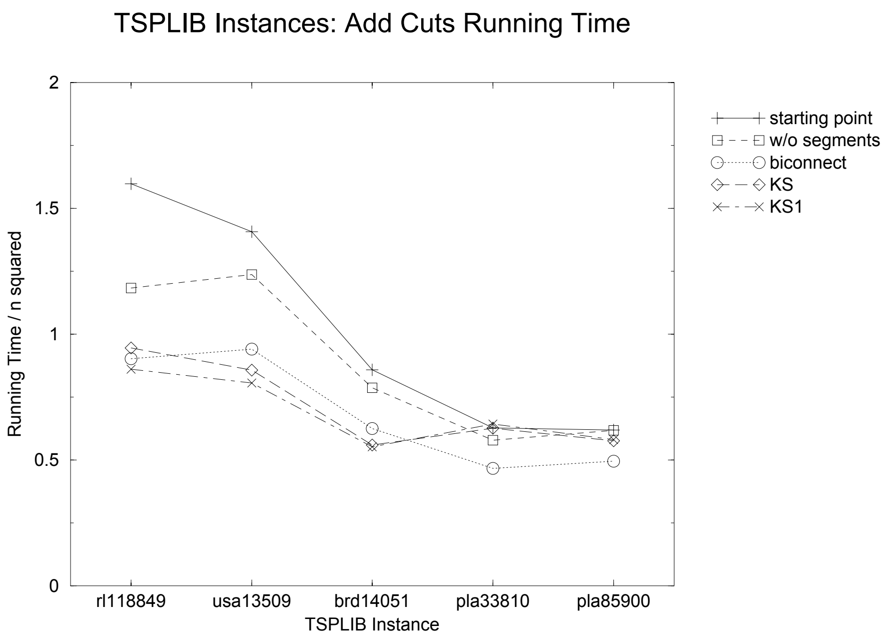

TSPLIB Instances: Add Cuts Running Time
Running Time / n squared
TSPLIB Instance
rl118849
usa13509
brd14051
pla33810
pla85900
0
0.5
1
1.5
2
starting point
w/o segments
biconnect
KS
KS1

Fig. 6.

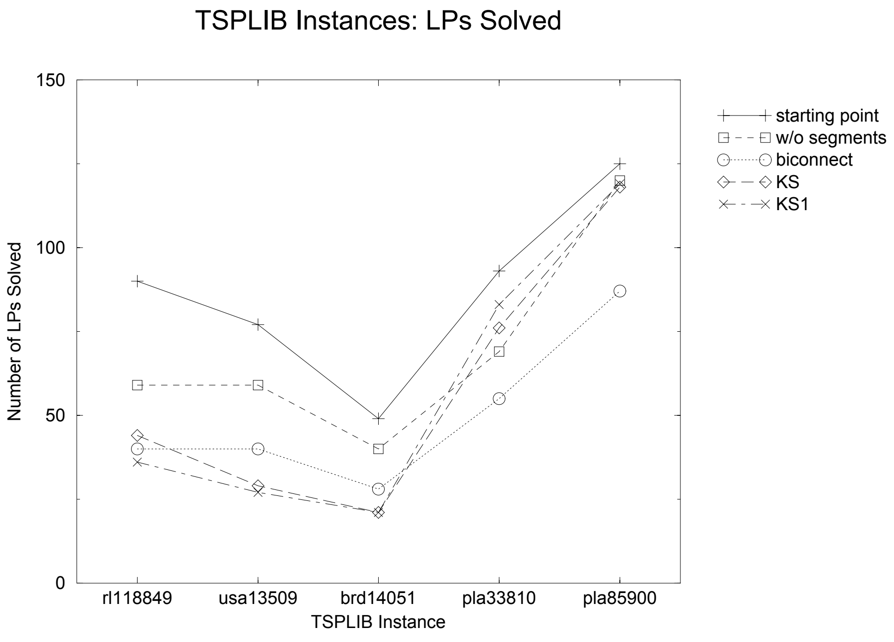

TSPLIB Instances: LPs Solved
Number of LPs Solved
TSPLIB Instance
rl118849
usa13509
brd14051
pla33810
pla85900
0
50
100
150
starting point
w/o segments
biconnect
KS
KS1

Fig. 7.

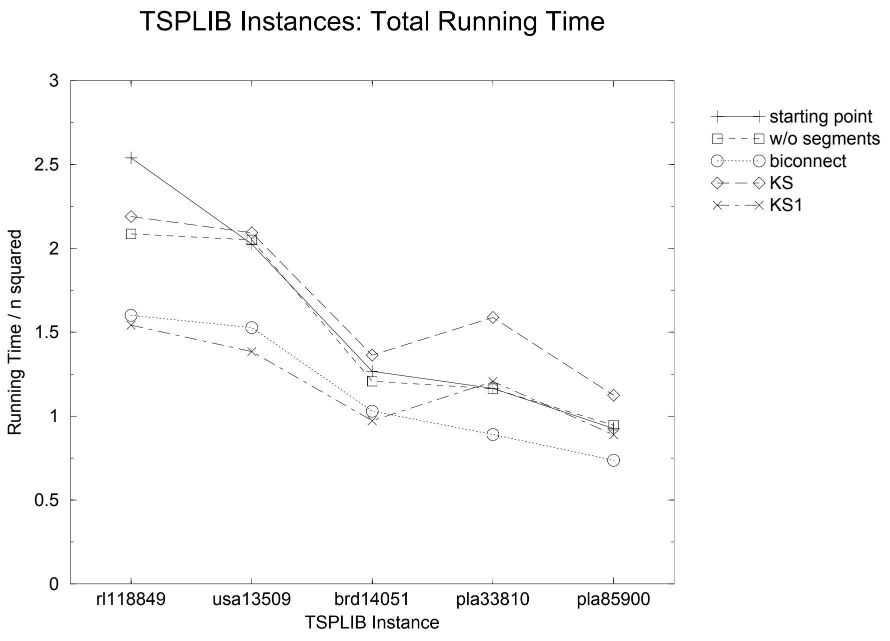


Fig. 8.

an LP solver a smallest region of the graph where a constraint is violated, it will encourage the solver to really fix the violation, rather than move the violation around.

It is worth noting that the right cuts are definitely not simply the ones that are easiest to find. As mentioned above, it was possible to slow the implementation down significantly by trying to use easy-to-find cuts first.

It is also interesting that it is possible to make any improvement with KS over a flow based code, because experimental studies indicate that for finding one minimum cut, it is generally much better to use the flow-based algorithm of Hao and Orlin. So our study suggests a different result: KS's ability to find all near-minimum cuts can in fact make it practical in situations where the extra cuts might be useful.

For future work, it does not seem like it would be particularly helpful to work harder at finding subtour elimination constraints for the TSP. However, studies of which constraints to find in more complicated IPs for the TSP could be more useful, and it might be interesting to investigate KS in other contexts where minimum cuts are used.

Acknowledgements

Many thanks to David Johnson for several helpful discussions and suggestions, including the suggestion that I do this work in the first place. Many thanks to

David Applegate for help with `concorde` and numerous helpful discussions and suggestions.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung*, ICM III:645–656, 1998.

[2] C. C. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the* 8th *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333. ACM-SIAM, Jan. 1997. A longer version appears as the fourth author's Masters thesis, available as a MIT Lab for Computer Science tech report, MIT-LCS-TR-719.

[3] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society of Industrial and Applied Mathematics*, 9(4):551–570, Dec. 1961.

[4] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer-Verlag, 1988.

[5] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Res.*, 18:1138–1162, 1970.

[6] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Math. Programming*, 1:6–25, 1971.

[7] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings of the* 7th *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 341–350, 1996.

[8] M. Jünger, G. Rinaldi, and S. Thienel. Practical performance of efficient minimum cut algorithms. Technical report, Informatik, Universität zu Köln, 1997.

[9] D. R. Karger. Minimum cuts in near-linear time. In G. Miller, editor, *Proceedings of the* 28th *ACM Symposium on Theory of Computing*, pages 56–63. ACM, ACM Press, May 1996.

[10] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, July 1996. Preliminary portions appeared in SODA 1992 and STOC 1993.

[11] E. L. Lawler, J. K. Lenstra, A. H. G. Rinooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[12] G. Reinelt. TSPLIB—a traveling salesman problem library. *ORSA J. Comput.*, 3:376–384, 1991.

[13] D. B. Shmoys and D. P. Williamson. Analyzing the held-karp tsp bound: A monotonicity property with applications. *Inform. Process. Lett.*, 35:281–285, 1990.

[14] V. V. Vazirani and M. Yannakakis. Suboptimal cuts: Their enumeration, weight, and number. In *Automata, Languages and Programming. 19th International Colloquium Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 366–377. Springer-Verlag, July 1992.

[15] L. Wolsey. Heuristic analysis, linear programming, and branch and bound. *Math. Prog. Study*, 13:121–134, 1980.

Obstacle-Avoiding Euclidean Steiner Trees in the Plane: An Exact Algorithm[⋆]

Martin Zachariasen and Pawel Winter

Dept. of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark
{martinz,pawel}@diku.dk

Abstract. The first exact algorithm for the obstacle-avoiding Euclidean Steiner tree problem in the plane (in its most general form) is presented. The algorithm uses a two-phase framework — based on the generation and concatenation of full Steiner trees — previously shown to be very successful for the obstacle-free case. Computational results for moderate size problem instances are given; instances with up to 150 terminals have been solved to optimality within a few hours of CPU-time.

1 Introduction

We consider the following topological network design problem:

- **Given:** A set Z of n terminals inside a simple polygon ω_0 with a set $\Omega = \{\omega_1, \ldots, \omega_h\}$ of h separated polygonal obstacles (holes).
- **Find:** The shortest (with respect to the L_2-metric) network T in ω_0 spanning Z, and avoiding obstacles in Ω.

The terminals are permitted to be on the boundaries of $\omega_0, \omega_1, \ldots, \omega_h$ but not in the interior of $\omega_1, \ldots, \omega_h$. The edges of T are permitted to meet at locations other than the given terminals. Such locations where more than two edges meet are referred to as *Steiner points*. The problem is referred to as the *obstacle-avoiding Euclidean Steiner tree problem*. T is referred to as the *obstacle-avoiding Euclidean Steiner minimum tree* (ESMTO). The problem is clearly NP-hard as it is a generalization of the well-known Euclidean Steiner tree problem (without obstacles) [2]. Whereas the obstacle-free problem has relatively few applications, the obstacle-avoiding problem is much more well-suited to model various real-life network design applications.

It is obvious that T must be a tree. Furthermore, no Steiner point in T can have more than three incident edges; this would not be the case if obstacles were not separated. A Steiner point not on a boundary of some obstacle has three edges making 120° with each other. A Steiner point on the boundary must be

⋆ Supported partially by the Danish Natural Science Research Council under contract 9701414 (project "Experimental Algorithmics").

located at a corner, and the 120° condition does not need to be fulfilled. We will refer to Steiner points on the boundaries of obstacles as *degenerate*.

ESMTOs are unions of *full Steiner trees* (FSTs) involving all terminals and some corners (Fig. 1). Terminals and corners have degree one in their FSTs. A terminal can appear in at most three FSTs while a corner must appear in two or three FSTs.

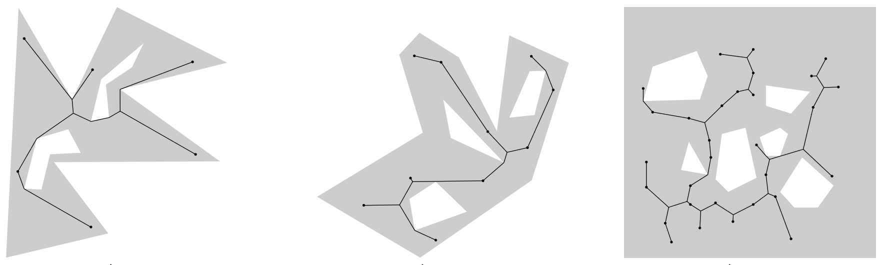


Fig. 1. Obstacle-avoiding Euclidean Steiner minimum tree examples.

Most efficient exact algorithms for the obstacle-free Euclidean Steiner tree problem are based on a straightforward framework. Subsets of terminals are considered one by one. For each subset, all its FSTs are determined one by one, and the shortest is retained. Several tests can be applied to these shortest FSTs in order to prune away those that cannot be in any optimal solution. The generation of FSTs can be substantially improved if FSTs are generated across various subsets of terminals (see [9,12] for details).

Surviving FSTs are concatenated in all possible ways to obtain trees spanning all terminals. The shortest of them yields an optimal solution. Warme [6] noticed that the concatenation of FSTs can in the obstacle-free case be formulated as a problem of finding a minimum spanning tree in a hypergraph with terminals as vertices and subsets spanned by FSTs as hyperedges. In addition, a pair of hyperedges must have at most one terminal in common. The length of a hyperedge is equal to the length of the corresponding FST.

This approach proved extremely powerful. Problem instances with more than 1000 terminals were solved in a reasonable amount of time [7].

The same general framework can be applied to the obstacle-avoiding Euclidean Steiner tree problem. In this case, all subsets of terminals *and* corners need to be considered. However, many FSTs will be excluded as their edges will be intersecting obstacles or will hit corners of obstacles at an inappropriate angle. Hence, the number of FSTs surviving the pruning tests will not explode, permitting to solve moderate size problem instances. Some pruning tests can be directly adapted from the obstacle-free case while others need considerable modifications.

The concatenation also needs to be modified since FSTs span both terminals *and* corners of obstacles. The concatenation problem can be formulated as a Steiner minimum tree problem in a hypergraph with terminals and corners as vertices. All terminals must be spanned while this is not required for corners. Hyperedges are given by terminals and corners spanned by FSTs and their length is equal to the length of corresponding FSTs. In addition, a pair of hyperedges must have at most one terminal or corner in common.

The purpose of this work is to present computational results for an exact algorithm for the obstacle-avoiding Steiner tree problem using the generation and concatenation techniques which proved so successful in the obstacle-free case. It should be noted that this is the first (implemented) exact algorithm for the obstacle-avoiding Steiner tree problem in its most general form. Previous contributions have focused on the exact solution of special cases [11] and heuristic algorithms [1,4,10].

The paper is organized as follows. The generation of FSTs is discussed in Section 2. This section focuses in particular on pruning tests and discusses preprocessing needed to speed up the generation. Our goal is to investigate how powerful pruning tests are; less attention is given to how these tests are implemented. The concatenation is described in Section 3. Computational results are given in Section 4. Concluding remarks including suggestions for further research are in Section 5.

2 Generation

An ESMTO is an union of FSTs, each spanning a subset of $W = Z \cup C$, where C is the set of obstacle corners of $\omega_0, \omega_1, \ldots, \omega_h$. Some corners may be eliminated from consideration as described below; the resulting set of *pseudo-terminals* is denoted $V \subseteq W$.

The algorithm uses the *equilateral point* generation strategy described by Winter and Zachariasen [12] (see this paper for an introduction to the overall strategy). An equilateral point corresponds to fixing the tree topology for a subset of pseudo-terminals. A fixed topology has a very limited region in which Steiner points may be placed, as explained below. A battery of tests is applied in order to reduce the feasible Steiner point regions. This often results in the complete elimination of an equilateral point.

Most pruning tests used in the obstacle-free case can be applied to the obstacle-avoiding case in a slightly altered version. In addition, new tests which consider the presence of obstacles are applied. In the following subsections we give some necessary definitions, describe the preprocessing phase which sets up data structures used by the generation algorithm, and finally we describe each of the pruning tests.

2.1 Definitions

Let ω_i, $0 \le i \le h$, denote both the polygon itself and the closed region bounded by it. The interior of ω_i is denoted by ω_i^I. The feasible region — in which the terminals and the ESMTO has to reside — is then $R = \omega_0 \setminus (\omega_1^I \cup \omega_2^I \cup \ldots \cup \omega_h^I)$.

The shortest (obstacle-avoiding) path between two points p and q in R is denoted by $sp(p,q)$. Shortest paths between points in W are contained in the *visibility graph* for W. This graph has W as its vertices; it has an edge between vertex p and q if and only if line segment pq is in R.

Any path $P(p,q)$ between two points p and q in R breaks into one or more *elementary paths* between terminals (from Z). The *Steiner distance* between p and q along $P(p,q)$ is the length of the longest elementary path in $P(p,q)$. The *bottleneck Steiner distance* b_{pq} between p and q is the minimum Steiner distance between p and q taken over all paths from p to q. Bottleneck distances between all points in W can be computed in polynomial time by using a modified Dijkstra algorithm for each vertex in the visibility graph as a root [2, p. 119].

Let p be a corner of some obstacle ω_i, $0 \le i \le h$. Denote by p^- (resp. p^+) the predecessor (resp. successor) of p on the boundary of ω_i such that the forbidden region is on the left when walking from p^- to p^+ via p.

The *equilateral point* e_{pq} of two points p and q is the third corner of the equilateral triangle $\triangle p e_{pq} q$ with the line segment pq as one of its sides, and such that the sequence of points $\{p, e_{pq}, q\}$ makes a right turn at e_{pq}. Points p and q are called the *base points* of e_{pq}. The *equilateral circle* of p and q is the circle circumscribing $\triangle p e_{pq} q$ and is denoted by C_{pq}. The counterclockwise arc from p to q on C_{pq} is called the *Steiner arc* from p to q. It is denoted by $\widehat{pq}$. The set of pseudo-terminals involved in the (recursive) construction of an equilateral point e will be denoted by $V(e)$.

Our interest in equilateral points and Steiner arcs is due to the fact that if two pseudo-terminals p and q are adjacent to a common Steiner point s_{pq}, and the sequence of points $\{p, s_{pq}, q\}$ makes a left turn at s_{pq}, then the location of s_{pq} is restricted to the Steiner arc $\widehat{pq}$. Furthermore, if p and q are Steiner points on Steiner arcs $\widehat{ac}$ and $\widehat{bd}$, and the sequence of points $\{p, s_{pq}, q\}$ makes a left turn, then the location of s_{pq} is restricted to the Steiner arc $\widehat{e_{ac}e_{bd}}$ where e_{ac} is the equilateral point with base points a and c, and e_{bd} is the equilateral point with base points b and d. Note that a, b, c, d can be pseudo-terminals or equilateral points.

2.2 Preprocessing

In the preprocessing phase we reduce the problem (if possible) and set up data structures used by the generation algorithm. The first step of the preprocessing phase is to *extend* obstacles iteratively by using the following procedure: Let p be a corner of some obstacle ω_i such that $\{p^-, p, p^+\}$ makes a right turn. The region given by $\triangle p^- p p^+$ is outside ω_i if $i \ge 1$ and inside otherwise. If $\triangle p^- p p^+$ contains no point from W we may delete the corner p from ω_i and add the direct segment between p^- and p^+, since it can be shown that no part of the ESMTO will be inside $\triangle p^- p p^+$.

The visibility graph for W is computed using a brute-force algorithm. Better algorithms exist for this problem [8], but the running time of this computation is insignificant compared to that used by the generation algorithm. Using the visibility graph, shortest paths $sp(p, q)$ and bottleneck Steiner distances b_{pq} are computed for every pair $p, q \in W$.

Finally, we identify the set of pseudo-terminals V. All terminals Z are obviously pseudo-terminals. In addition, we add each corner $p \in C$ such that $\{p^-, p, p^+\}$ makes a left turn; this means that the exterior angle at p is greater than 180°.

2.3 Projections

Consider an equilateral point e_{pq} of p and q. Assume that q itself is an equilateral point of a and c. Assume furthermore that the feasible locations of the Steiner point s_{ac} on the Steiner arc $\widehat{ac}$ has been restricted to the subarc $\widehat{a'c'}$ as shown in Fig. 2a. Consider the ray from q through c'. If it intersects $\widehat{pq}$, then the feasible locations of the Steiner point s_{pq} on $\widehat{pq}$ can be restricted to $\widehat{pq'}$ where q' is the intersection point. Similarly, if the ray from q through a' intersects $\widehat{pq'}$, then the feasible locations of s_{pq} can be restricted to $\widehat{p'q'}$ where p' is the intersection point. Finally, if $\widehat{a'c'}$ intersects $\widehat{p'q'}$, then q' can be moved to this intersection point.

If the rays from q through c' and a' do not intersect the Steiner arc $\widehat{pq}$ as shown in Fig. 2b, then the equilateral point e_{pq} can be pruned away; it is impossible to place s_{ab} on $\widehat{ac}$ such that its three incident edges make 120° with each other.

Above arguments apply in similar fashion when p is an equilateral point.

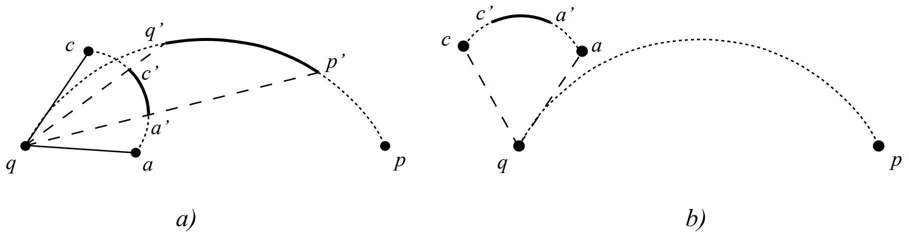


Fig. 2. Projections.

2.4 Bottleneck Steiner Distances

Consider the feasible subarc $\widehat{p'q'}$ of the Steiner arc $\widehat{pq}$. If q is an equilateral point e_{ac} based on a and c, let a' and c' denote respectively the rightmost and the leftmost feasible location of s_{ac} on $\widehat{ac}$ (if q is a pseudo-terminal, let $a' = c' = q$). If there are two pseudo-terminals $z_p \in V(p)$ and $z_q \in V(q)$ such that

$$b_{z_p z_q} < |a'p'|,$$

then p' can be moved toward q' until the equality is obtained. Similar argument can be applied to move q' toward p'.

2.5 Obstacle Intersections

Consider the feasible subarc $\widehat{p'q'}$ of the Steiner arc $\widehat{pq}$. Let a' and c' be as defined in Subsection 2.4. Suppose that there is a side s in some obstacle ω_i such that a' and p' are not visible to each other as shown in Fig. 3. Then p' can be moved toward q' until the ray from q through p' goes through an end-point of s (Fig. 3a) or until the intersection of the Steiner arc $\widehat{pq}$ with s (Fig. 3b).

If c' and q' are not visible to each other, then q' can be moved toward p' in analogous fashion.

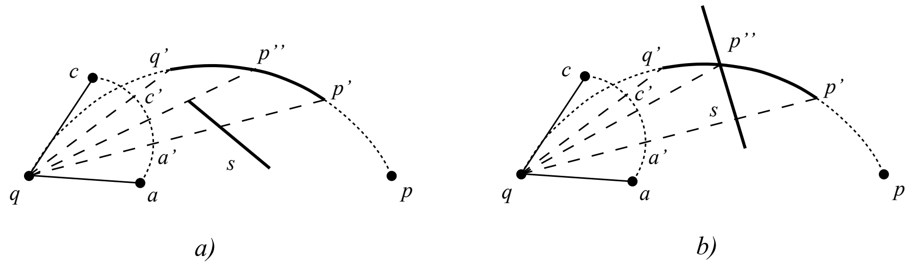


Fig. 3. Obstacle segment intersection.

2.6 Obstacle Hitting

Consider an equilateral point e_{pq} such that $q \in C$ is a corner of some obstacle ω_i. Let q^- (resp. q^+) denote the predecessor (resp. the successor) of q in ω_i (Fig. 4). The angles $\angle s_{pq}qq^+$ and $\angle q^-qs_{pq}$ must both be at least 120°. Consequently, if either p' or q' are infeasible locations for s_{pq} , they can be moved toward each other until either the 120° angle condition is satisfied, or one of q^-, q^+ becomes invisible from p' or q'.

2.7 Lune Test

Consider an equilateral point e_{pq} such that q is a pseudo-terminal. Let $\widehat{p'q'}$ denote the feasible subarc of the Steiner arc $\widehat{pq}$. Suppose that there exists a terminal $z \in Z$ such that $|qz| < |qp'|$ and $|p'z| < |qp'|$, and q, p', z are visible to each other.

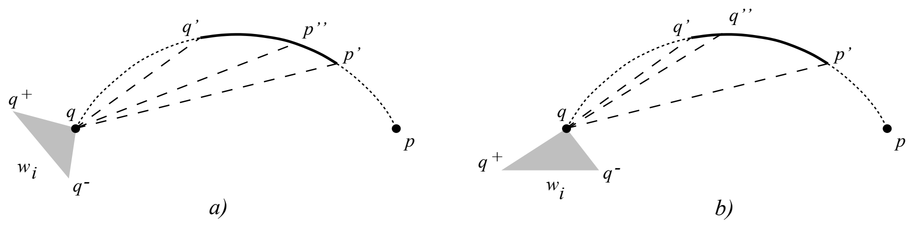


Fig. 4. Obstacle hitting.

Then p' can be moved toward q' until one of the above inequalities is not fulfilled. The determination of the new location of p' becomes somewhat complicated due to the presence of obstacles. To simplify this, it is required that q and z are visible from every point on the subarc $\widehat{p'q'}$. The new location of p' can then be determined as described in [12]. It should be noted that the lune test in [12] is much more general as it also applies in cases when q is an equilateral point.

Similar argument applies if p is a pseudo-terminal, and q' is attempted to be pushed toward p'.

2.8 Upper Bounds

Assume the ESMTO has an FST whose construction involves the equilateral point e_{pq}. The length of the portion F' of the FST "behind" the Steiner point s_{pq} on the Steiner arc $\widehat{pq}$ is equal to $|e_{pq}s_{pq}|$ [2]. If $\widehat{p'q'}$ is the feasible arc of the Steiner arc $\widehat{pq}$, a lower bound on the length of F' is $\min\{|e_{pq}p'|, |e_{pq}q'|\}$. Consequently, if a shorter tree spanning $V(e_{pq})$ and $\widehat{p'q'}$ can be constructed the equilateral point e_{pq} may be discarded.

In the obstacle-free case [12] the heuristic by Smith et al. [5] was used to provide such a good tree. The heuristic constructs FSTs for appropriately chosen subsets of up to four points taken from $V(e_{pq}) \cup \{p', q'\}$. These are then concatenated in a greedy fashion to obtain a solution. We used a simple modification of this heuristic in order to make it work for the obstacle-avoiding problem: When constructing FSTs spanning up to four points, those intersecting an obstacle are deleted. However, in some cases this heuristic may provide a forest rather than a tree; in these cases infinity was returned as the length of the heuristic tree.

2.9 Wedge Property

Suppose that $\widehat{p'q'}$ is the feasible subarc of the Steiner arc $\widehat{pq}$. Draw four rays (Fig. 5): l_1: rooted at e_{pq} through p', l_2: rooted at e_{pq} through q', l_3: rooted at p through q', l_4: rooted at q through p'. Consider the regions R_1: bounded by l_1, l_2 and $\widehat{p'q'}$, and possibly a polygonal boundary of an obstacle, R_2 (resp. R_3): bounded by l_2 and l_3 (resp. l_1 and l_4), and possibly a polygonal boundary of an obstacle.

Suppose that the region R_1 contains no pseudo-terminals. If R_2 or R_3 contains no pseudo-terminals, then no FST involving e_{pq} and with a Steiner point on $\widehat{p'q'}$ can exist.

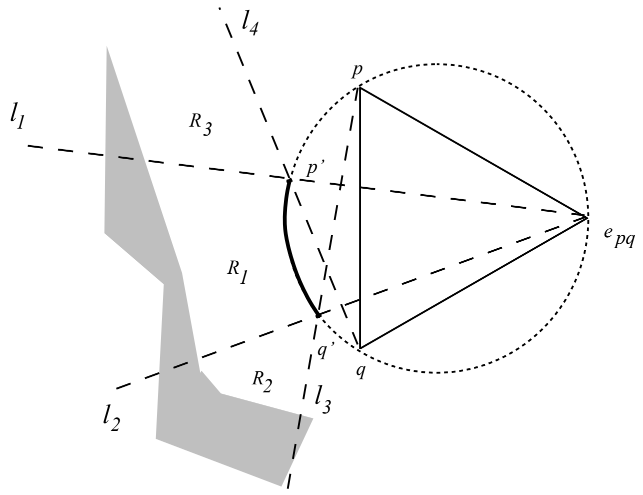


Fig. 5. Wedge property.

2.10 Full Steiner Tree Tests

Once an FST for an equilateral point e_{pq} and a pseudo-terminal r in the wedge region R_1 (Fig. 5) has been identified, modified versions of the tests described above can be applied. Since the locations of Steiner points of FSTs are at that stage determined, these tests can prove more effective then when applied to e_{pq} where the locations of Steiner points were not fixed (although they were restricted to subarcs of Steiner arcs).

Also edges in the subgraph of the visibility graph induced by V are valid FSTs. Some of them may be eliminated by the modified versions of tests described above.

3 Concatenation: Steiner Trees in Hypergraphs

Given the set $\mathcal{F} = \{F_1, F_2, ..., F_m\}$ of FSTs from the generation phase, the concatenation problem is to select a subset $\mathcal{F}^* \subseteq \mathcal{F}$ such that the FSTs in $\mathcal{F}^*$ interconnect Z and have minimum total length.

Consider a *hypergraph* $H = (V, \mathcal{F})$ with the set of pseudo-terminals V as its vertices and the set of FSTs $\mathcal{F}$ as its hyperedges, that is, each FST $F_i \in \mathcal{F}$ spans a subset of V denoted by $V(F_i)$. A *chain* in H from $z_{i_0} \in V$ to $z_{i_k} \in V$ is a sequence $z_{i_0}, F_{j_1}, z_{i_1}, F_{j_2}, z_{i_2}, \ldots, F_{j_k}, z_{i_k}$ such that all vertices and hyperedges are distinct and $z_{i_{l-1}}, z_{i_l} \in V(F_{j_l})$ for $l = 1, 2, \ldots, k$. A *tree* in H spanning $Z \subseteq V$ is a subset of hyperedges $\mathcal{F}' \subseteq \mathcal{F}$ such that there is a *unique* chain between every pair of vertices $z_i, z_j \in Z$. The weight of the tree is the total weight of hyperedges in the tree; each hyperedge $F_i \in \mathcal{F}$ has weight equal to its length $|F_i|$. Finding an ESMTO for Z is equivalent to finding a minimum-weight tree in H spanning Z. This problem is denoted the *Steiner tree problem in hypergraphs (SPHG)* since it is a natural generalization of the well-known NP-hard Steiner tree problem in graphs [2] .

In order to solve SPHG, we generalize a very powerful integer programming formulation for solving the minimum spanning tree problem in hypergraphs [6,7]. Denote by x an m-dimensional *binary* vector such that $x_i = 1$ if and only if F_i is selected to be part of the tree spanning Z. Let c be a vector in $\Re^m$ whose components are $c_i = |F_i|$. For any subset $U \subseteq V$, define the *cut* given by U as

$$(U : V \setminus U) = \{F_i \in \mathcal{F} \mid (V(F_i) \cap U \neq \emptyset) \wedge (V(F_i) \cap (V \setminus U) \neq \emptyset)\}.$$

A straightforward integer program (IP) for solving SPHG is now the following:

$$\mathbf{min}\ cx \tag{1}$$

$$\text{s.t.} \quad \sum_{F_i \in (U:V\setminus U)} x_i \geq 1, \quad \forall U \subset V,\ U \cap Z \neq \emptyset, (V \setminus U) \cap Z \neq \emptyset \tag{2}$$

The objective function (1) minimizes the total length of the chosen FSTs subject to connectivity constraints (2). The connectivity constraints ensure that every pair of *terminals* is connected, i.e., that there is no cut of total weight less than 1 separating them.

This integer program can be solved via branch-and-cut using lower bounds provided by linear programming (LP) relaxation. Connectivity constraints are added using separation methods [6]. However, a preliminary study indicated that the LP lower bounds provided by this formulation were very weak (the LP solution in the root node was typically 10-20% off the optimum integer solution).

In order to strengthen this formulation significantly a set of auxiliary variables must be added to the formulation. Let $S = V \setminus Z$ be the set of Steiner vertices in H; let y be an $|S|$-dimensional binary vector such that $y_s = 1$ if and only if Steiner vertex $s \in S$ is part of the tree spanning Z (we say that such a Steiner vertex is active). Note that y does not appear in the objective function. The following constraints may now be added to the formulation:

$$\sum_{F_i \in (\{s\}:V\setminus\{s\})} x_i \geq 2y_s, \quad \forall s \in S \tag{3}$$

$$\sum_{F_i \in (\{s\}:V\setminus\{s\})} x_i \leq 3y_s, \quad \forall s \in S \tag{4}$$

$$\sum_{F_i \in \mathcal{F}} (|V(F_i)| - 1)x_i = |Z| + \sum_{s \in S} y_s - 1 \tag{5}$$

$$\sum_{F_i \in \mathcal{F}} \max(0, |V(F_i) \cap U| - 1)x_i \leq |U \cap Z| + \sum_{s \in (U \cap S)} y_s - 1, \qquad \forall U \subset V,\ U \cap Z \neq \emptyset \tag{6}$$

Constraints (3) and (4) bound the degree of an active Steiner vertex to be either 2 or 3; non-active Steiner vertices must have degree 0. Equation (5) enforces the correct number and cardinality of hyperedges to construct a tree spanning Z. Constraints (6) eliminate cycles: For a given set of vertices U the total “edge material” inside U cannot exceed the cardinality of U minus 1; note that the actual cardinality of U is the sum of the number of terminals and active Steiner vertices residing in U.

The concatenation was implemented by modifying the code of Warme [6] for solving the minimum spanning tree in hypergraph problem. Only relatively small changes were needed in order to make this code work for the formulation given above; details are omitted.

4 Computational Experience

The exact algorithm for the obstacle-avoiding Steiner tree problem was experimentally evaluated on an HP9000 workstation[1] using the programming language C++ and class library LEDA (version 3.7) [3]. CPLEX 5.0 was used to solve linear programming (LP) relaxations.

In Table 1 we present computational results for the six instances shown in Fig. 1 and 6. This set includes four problem instances given in [1,10]. The instances have up to six polygonal obstacles, in addition to the boundary polygon.

The three smallest instances were solved within one minute and the other three within one hour. The number of generated FSTs increases only slightly faster than linear in the number of pseudo-terminals. The number of generated equilateral points, however, increases more rapidly. The integer program formulation for the concatenation is very tight; four out of six instances are solved in the root node of the branch-and-bound tree. The CPU-times spent in the concatenation phase are only a small fraction of the times needed for doing the generation of FSTs.

In order to test the new algorithm for larger instances we generated terminals randomly for a fixed set of obstacles as follows: A square $\bar{R}$ bounding all obstacles ($R \subseteq \bar{R}$) was constructed. Points were drawn with uniform distribution from $\bar{R}$; if a point belonged to R it was added to the set of terminals Z. This process was repeated until the desired number of terminals was generated.

In Table 2 and 3 we present results for the set of obstacles shown in Fig. 1a and 6b, respectively. Seven instances with 10 to 150 terminals were tested for each set of obstacles. It should be noted that the larger terminal sets contain the smaller as subsets (see also Fig. 7 and 8).

[1] Machine: HP Visualize Model C200. Processor: 200 MHz PA-8200. Main memory: 1 GB. Performance: 14.3 SPECint95 and 21.4 SPECfp95. Operating system: HP-UX 10.20. Compiler: GNU C++ 2.8.1 (optimization flag -O3).

The results for the smaller set of obstacles (Table 2) show a very regular pattern. As the number of terminals increases, the number of generated equilateral points increases quadratically and the number of surviving FSTs increases linearly in the number of pseudo-terminals. These results are fairly similar to results for the obstacle-free case. All instances are solved in less than one hour; the concatenation problem is solved in seconds.

For the larger set of obstacles (Table 3) the pattern is more complex. In general, the number of FSTs increases as the number of terminals increases, but only quite slowly. The running times also increase in general, but moderate size instances ($|Z| = 25$) apparently tend to be more difficult than larger instances ($|Z| = 75$). This can be explained by the considerably weaker bottleneck Steiner distance and lune tests for the moderately sized instances. The larger number of obstacles tends to weaken these tests for moderate size instances, but as the number of terminals increases these tests gradually become almost as powerful as in the obstacle-free case.

In order to further investigate the effect of increasing the number of obstacles, we generated a set of instances based on the socalled *Sierpinski triangle fractal*. Starting with an equilateral triangle (the boundary polygon), new triangles are iteratively added as shown in Fig. 9. In order to avoid obstacles touching each other (a requirement in the current implementation) all obstacles are shrunk by a small constant factor.

Four Sierpinski triangle fractal iterations result in 40 triangles (obstacles); using these obstacles a set of 10 randomly generated terminals were drawn as described above. Each of the four instances corresponding to iterations 1 to 4 were then solved using this set of terminals which by construction is inside the feasible region for each set of obstacles.

The results are shown in Table 4. The number of equilateral points and FSTs shows the same pattern as previously seen, while an interesting observation can be made for the concatenation phase. Apparently, the lower bound for the largest instance is quite weak; the result is a dramatic increase in the number of branch-and-bound nodes and the CPU-time. The weak lower bound can be explained by the symmetry of the problem: There exist more than one tree having shortest possible length. Some junctions (Steiner points) can be moved from one Sierpinski triangle to another without changing the length of the tree. This makes it difficult to avoid fractional variables in the LP-relaxation.

5 Conclusion

The first exact algorithm for the obstacle-avoiding Euclidean Steiner tree problem was presented. Moderate size problem instance with up to 150 terminals were solved to optimality within a few hours of CPU-time. Future research will focus on improving the FST generation algorithm, applying it to problem instances with special structure and to adapt the approach to the rectilinear Steiner tree problem with obstacles.

Table 1. Experimental results, selected instances. $|Z|$: Number of terminals. h: Number of obstacles. $|C|$: Number of obstacle corners. $|V|$: Number of pseudo-terminals. Fig/ref: Figure and/or reference. Eq-pts: Number of generated equilateral points. $|\mathcal{F}|$: Total number of generated FSTs (resp. number of 2-vertex FSTs). Gap: Root LP objective value vs. optimal value (gap in percent). Nds: Number of branch-and-bound nodes. Red: Reduction over MST in percent. CPU: CPU-time (seconds).

Instance					Generation			Concatenation			
$\|Z\|$	h	$\|C\|$	$\|V\|$	Fig/ref	Eq-pts	$\|\mathcal{F}\|$	CPU	Gap	Nds	Red	CPU
7	1	9	11	6a [1]	81	28 (13)	3.0	0.000	1	8.04	0.1
6	2	21	19	1a	230	76 (36)	11.2	0.000	1	10.70	1.3
10	3	21	23	1b	440	75 (36)	35.7	0.000	1	4.52	0.2
20	5	43	49	6b [1]	4833	250 (68)	807.2	0.552	1	5.45	4.0
30	6	34	58	6c [1]	4525	275 (74)	638.5	0.000	1	4.50	1.6
30	6	31	57	1c [10]	11414	388 (78)	1973.0	0.973	3	4.58	8.0

Table 2. Experimental results, randomly generated terminals. Note that the obstacles are the same for all instances ($h = 2$ and $|C| = 21$). See Table 1 for an explanation of the symbols.

Instance		Generation			Concatenation			
$\|Z\|$	$\|V\|$	Eq-pts	$\|\mathcal{F}\|$	CPU	Gap	Nds	Red	CPU
10	23	291	84 (33)	20.1	0.000	1	3.35	0.2
25	38	1261	130 (45)	96.2	0.000	1	3.22	0.4
50	63	2396	196 (67)	230.9	0.000	1	2.19	2.0
75	88	4100	230 (94)	494.9	0.000	1	1.70	2.5
100	113	6072	290 (115)	939.9	0.000	1	2.53	1.3
125	138	8887	390 (140)	1710.0	0.000	1	3.16	2.4
150	163	11992	514 (166)	2933.1	0.000	1	2.93	5.5

Table 3. Experimental results, randomly generated terminals. Note that the obstacles are the same for all instances ($h = 5$ and $|C| = 43$). See Table 1 for an explanation of the symbols.

Instance		Generation			Concatenation			
$\|Z\|$	$\|V\|$	Eq-pts	$\|\mathcal{F}\|$	CPU	Gap	Nds	Red	CPU
10	37	13792	406 (67)	1407.8	2.333	2	3.14	20.3
25	54	31400	556 (76)	6607.6	0.299	2	4.48	22.3
50	79	14688	405 (102)	3578.0	0.000	1	3.21	3.2
75	104	12840	383 (119)	3144.4	0.000	1	3.31	4.1
100	129	15640	494 (142)	4549.5	0.000	1	3.57	3.5
125	154	23127	728 (167)	9178.8	0.000	1	2.78	8.2
150	179	18267	700 (190)	7655.1	0.000	1	2.65	8.1

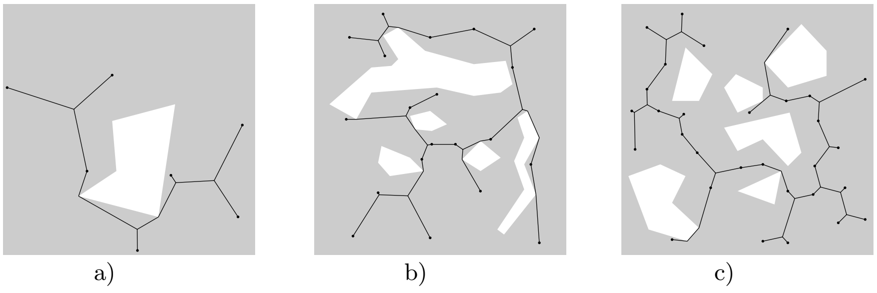


Fig. 6. Problem instances from [1].

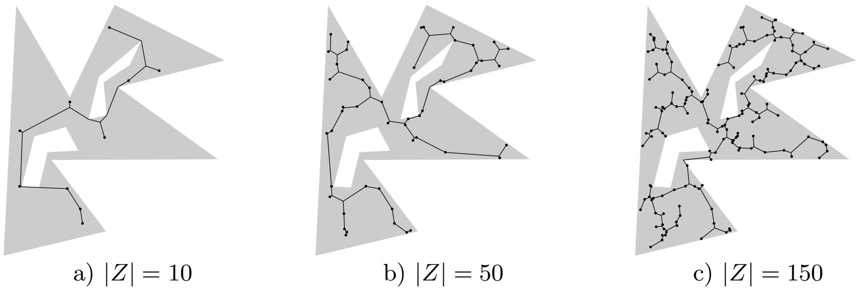


Fig. 7. Randomly generated terminals with fixed set of obstacles (shown on Fig. 1a.). Numeric results are given in Table 2.

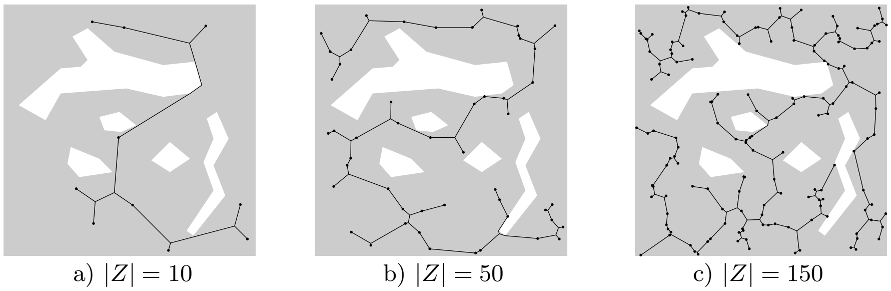


Fig. 8. Randomly generated terminals with fixed set of obstacles (shown on Fig. 6b.). Numeric results are given in Table 3.

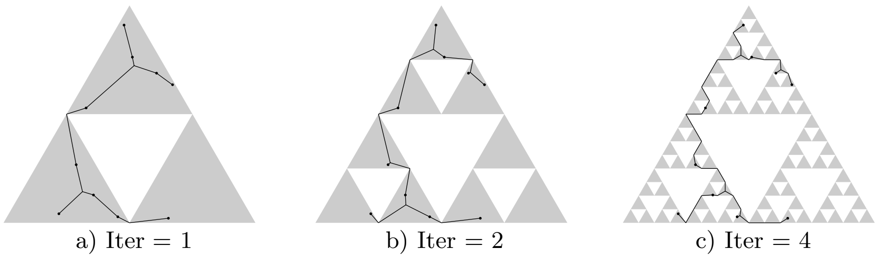


Fig. 9. Sierpinski triangle fractal. Numeric results are given in Table 4.

Table 4. Experimental results, Sierpinski triangle fractal. Note that the terminals are the same for all instances ($|Z| = 20$). See Table 1 for an explanation of the symbols.

Instance			Generation			Concatenation			
h	$\|C\|$	$\|V\|$	Eq-pts	$\|\mathcal{F}\|$	CPU	Gap	Nds	Red	CPU
1	6	13	43	15 (12)	0.7	0.000	1	1.07	0.0
4	15	22	82	37 (25)	2.3	0.000	1	1.52	0.1
13	42	49	245	109 (82)	14.4	0.000	1	3.77	1.4
40	123	130	1073	392 (308)	258.8	4.797	33	2.60	1194.6

Acknowledgement. The authors would like to thank David M. Warme for valuable discussions concerning the integer programming formulation and implementation.

References

1. A. Armillotta and G. Mummolo. A Heuristic Algorithm for the Steiner Problem with Obstacles. Technical report, Dipt. di Pregettazione e Produzione Industriale, Univ. degli Studi di Bari, Bari, 1988.
2. F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem.* Annals of Discrete Mathematics 53. Elsevier Science Publishers, Netherlands, 1992.
3. K. Mehlhorn and S. Näher. LEDA - A Platform for Combinatorial and Geometric Computing. Max-Planck-Institut für Informatik, Saarbrücken, Germany, `http://www.mpi-sb.mpg.de/LEDA/leda.html`, 1996.
4. J. S. Provan. An Approximation Scheme for Finding Steiner Trees with Obstacles. *SIAM Journal on Computing*, 17(5):920–934, 1988.
5. J. M. Smith, D. T. Lee, and J. S. Liebman. An $O(n \log n)$ Heuristic for Steiner Minimal Tree Problems on the Euclidean Metric. *Networks*, 11:23–29, 1981.
6. D. M. Warme. *Spanning Trees in Hypergraphs with Applications to Steiner Trees.* Ph.D. Thesis, Computer Science Dept., The University of Virginia, 1998.
7. D. M. Warme, P. Winter, and M. Zachariasen. Exact Algorithms for Plane Steiner Tree Problems: A Computational Study. In D.-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, Kluwer Academic Publishers, Boston, to appear.
8. E. Welzl. Constructing the Visibility Graph for n-line Segments in $O(n^2)$ Time. *Information Processing Letters*, 20:167–171, 1985.
9. P. Winter. An Algorithm for the Steiner Problem in the Euclidean Plane. *Networks*, 15:323–345, 1985.
10. P. Winter. Euclidean Steiner Minimal Trees with Obstacles and Steiner Visibility Graphs. *Discrete Applied Mathematics*, 47:187–206, 1993.
11. P. Winter and J. M. Smith. Steiner Minimal Trees for Three Points with One Convex Polygonal Obstacle. *Annals of Operations Research*, 33:577–599, 1991.
12. P. Winter and M. Zachariasen. Euclidean Steiner Minimum Trees: An Improved Exact Algorithm. *Networks*, 30:149–166, 1997.

Adaptive Algorithms for Cache-efficient Trie Search

Anurag Acharya, Huican Zhu, and Kai Shen

Dept. of Computer Science
University of California, Santa Barbara, CA 93106

Abstract. In this paper, we present cache-efficient algorithms for trie search. There are three key features of these algorithms. First, they use different data structures (partitioned-array, B-tree, hashtable, vectors) to represent different nodes in a trie. The choice of the data structure depends on cache characteristics as well as the fanout of the node. Second, they adapt to changes in the fanout at a node by dynamically switching the data structure used to represent the node. Third, the size and the layout of individual data structures is determined based on the size of the symbols in the alphabet as well as characteristics of the cache(s). We evaluate the performance of these algorithms on real and simulated memory hierarchies. Our evaluation indicates that these algorithms outperform alternatives that are otherwise efficient but do not take cache characteristics into consideration. A comparison of the number of instructions executed indicates that these algorithms derive their performance advantage primarily by making better use of the memory hierarchy.

1 Introduction

Tries are widely used for storing and matching strings over a given alphabet. Applications include dictionary lookup for text processing [6–8, 14, 22], itemset lookup for mining association rules in retail transactions [1, 2], IP address lookup in network routers [26, 27] and partial match queries [13, 24]. There has been much work on reducing the storage requirement and the instruction count for tries – for example [3, 5, 4, 6, 19, 20, 22]. These algorithms, however, do not take the memory hierarchy into account. Given the depth of memory hierarchies on modern machines, good cache performance is critical to the performance of an algorithm.

In this paper, we present cache-efficient algorithms for trie search. There are three key features of these algorithms. First, they use different data structures (partitioned-array, B-tree, hashtable, vector) to represent different nodes in a trie. The choice of the data structure depends on cache characteristics as well as the fanout of the node. Second, they adapt to changes in the fanout at a node by dynamically switching the data structure used to represent the node. Third, the size and the layout of individual data structures is determined based on the size of the symbols in the alphabet as well as characteristics of the cache(s).

We evaluate the performance of these algorithms on real and simulated memory hierarchies. To evaluate their performance on real machines, we ran them on three different machines with different memory hierarchies (Sun Ultra-2, Sun Ultra-30, and SGI Origin-2000). To evaluate the impact of variation in cache characteristics on the performance of these algorithms, we simulated architectures that differed in cache line size and cache associativity.[1]

To drive these experiments, we used two datasets from different application domains and with different alphabet sizes. The first dataset was from the text processing domain and consisted of a trie containing all the words in the Webster's Unabridged Dictionary. Against this dictionary, we ran searches using all the words in Herman Melville's *Moby Dick*. This dataset had an alphabet of 54 symbols (lower and upper case English characters, hyphen and apostrophe). The second dataset was from the datamining domain. For this application, the alphabet consists of items that can be purchased in a grocery store (beer, chips, bread etc) and a string consists of a single consumer transaction (the set of items purchased at one time). The task is to determine the set of items (referred to as *itemsets* that are frequently purchased together. Tries are used in this application to store the candidate itemsets and to help determine the frequent itemsets. The dataset used in our experiments was generated using the Quest datamining dataset generator which we obtained from IBM Almaden [23]. This dataset generator has been widely used in datamining research [1, 2, 15, 28, 29]. The alphabet for this dataset contained 10,000 symbols (corresponding to 10,000 items).

Our evaluation indicates that these algorithms out-perform alternatives that are otherwise efficient but do not take cache characteristics into consideration. For the dictionary dataset, our algorithm was 1.7 times faster on the SGI Origin-2000 and 4 times faster on both the Sun Ultras compared to the ternary search tree algorithm proposed by Bentley and Sedgewick [7]. For the datamining dataset, our algorithm was 1.4-1.9 times faster than a B-tree-trie and 1.2-1.5 times faster than a hashtable-trie.[2] A comparison of the number of instructions executed indicates that the algorithms presented in this paper derive their performance advantage over otherwise efficient algorithms[3] by making better use of the memory hierarchy. Furthermore, the perfomance advantage gained by the adaptive algorithms is not at the cost of additional space. Finally, simulation results indicate that the performance of our algorithms is not sensitive to the cache line size or the cache associativity.

The paper is organized as follows. A brief review of tries and modern memory hierarchies can be found in Appendix A. Section 2 presents the intuition

[1] Appendix A provides a brief introduction to caches in modern machines.

[2] Since our experiments indicated that the performance of the adaptive algorithms is insensitive to variations in the cache line size and associativity, we speculate that the difference in the speedups achieved for the Sun Ultra machines and the SGI Origin-2000 is due to the higher memory bandwidth in the Origin-2000.

[3] Ternary search tree for the dictionary dataset and the hashtable-trie for the datamining dataset.

behind our algorithms and describes them in some detail. Section 3 describes our experiments and Section 4 presents the results. Section 5 presents related work and Section 6 presents a summary and conclusions.

2 Algorithms

The design of the algorithms presented in this paper is based on three insights. First, there is a large variation in the fanout of the nodes in a trie. The nodes near the root usually have a large fanout; the nodes further from the root have a smaller fanout. This suggests that different data structures might be suitable for implementing different nodes in a trie. Second, only the first memory reference in a cache line has to wait for data to be loaded into the cache. This suggests that the data structures should be designed so as to pack as many elements in a cache line as possible.[4] Third, at each level of a trie, at most one link is accessed (the link to the successor if the match can be extended, no link if the match cannot be extended). This suggests that for nodes with a large fanout, the keys and the links should be stored separately. This reduces the number of links loaded and avoids loading any links for unsuccessful searches.

We present two algorithms. The first algorithm is for tries over large alphabets which require an integer to represent each symbol. The second algorithm is for tries over small alphabets whose symbols can be represented using a character. While a wide range of key sizes is possible, we believe that these two alternatives (integer and character) cover most trie applications.

2.1 Algorithm for large alphabets

This algorithm assumes that individual symbols are represented by integers and uses three alternative data structures, a *partitioned-array*, a bounded-depth B-tree and a hashtable, for representing trie nodes. It selects between them based on the fanout of the node and the cache line size.

The *partitioned-array* structure consists of two arrays – one to hold the keys and the other to hold the corresponding links. Each array is sized to fit within a single cache line. This assumes that integers and links are of the same size. For architectures on which links and integers are not the same size, the array containing the links is allowed to spread over multiple cache lines. The bounded-depth B-tree structure consists of a B-tree with at most k levels where k is a parameter. Individual nodes in this structure are represented using partitioned-arrays. The parameter k is a small constant (2-5) that is determined by the number of cache misses one is willing to tolerate (up to two cache misses per level – one for the key array and the other for the link array). For the experiments reported in this paper, we ran tests with $k = 2/3/4/5$ and found that $k = 4$

[4] Some memory hierarchies return the first reference first and load the rest of the cache line later. For such memory hierarchies, subsequent memory references may also need to wait.

yielded the best performance. The hashtable structure uses chains of partitioned-arrays to handle collisions. Figure 1 illustrates all three structures.

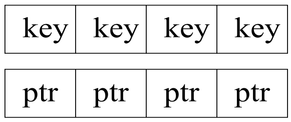
When a new node is created, the partitioned-array structure is used. As the fanout of a node increases, new keys are inserted into the partitioned-array. When the fanout of a node increases beyond the capacity of a partitioned-array, the partitioned-array is converted into a B-tree. Since partitioned-arrays are used to represent B-tree nodes, this conversion is cheap. Subsequent insertions into the node result in new keys being added to the B-tree. The bounded-depth B-tree structure is used as long as its depth is less than the depth threshold, k (as mentioned above, we used $k = 4$). When the fanout of the node increases such that the keys no longer fit within the bounded-depth B-tree, the B-tree is converted into a hashtable. This conversion requires substantial rearrangement. However, with a suitable depth bound for the B-tree, this conversion will be rare. Furthermore, since the hashtable uses the same basic data structure (the partitioned-array), it will be able to re-use the memory (and possibly cache lines) that were used by the B-tree.

2.2 Algorithm for small alphabets

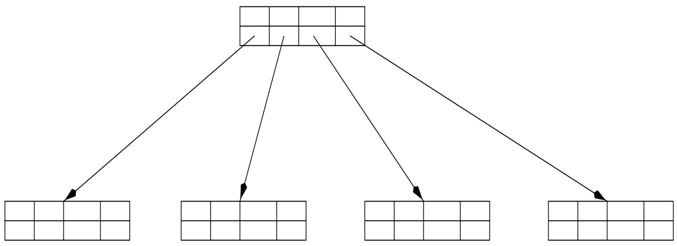
This algorithm assumes that individual symbols are represented by characters and uses partitioned-arrays of different sizes (1/2/4/8) and an m-way vector (where m is the size of the alphabet) to represent trie nodes. When a new node is created, a partitioned-array with just a single entry is used. As new keys are inserted, the partitioned-array is grown by doubling its size. When the number of keys exceeds a threshold, the partitioned-array is converted into an m-way vector (in order to reduce search time). In our algorithm, the threshold is selected by dividing the cache line size by the size of each link. For this algorithm, we use the size of links to size the partitioned-array instead of the size of the keys. This is because a large number of keys fit into cache lines available on current machines (32-byte/64-byte) and using link-arrays with 32/64 slots would greatly increase the space requirement for these nodes. The m-way vector consists of a link array that is indexed by the key. Conversion between these alternatives is cheap and requires only re-allocation of the space required to hold the arrays and copying the contents.

3 Experimental setup

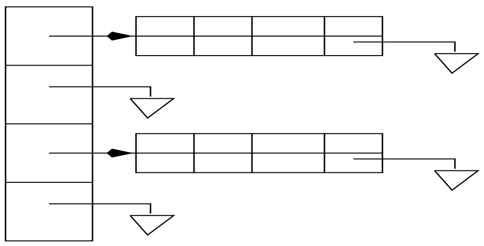
We evaluated the performance of these algorithms on real and simulated memory hierarchies. To evaluate their performance on real machines, we ran them on three different machines with different memory hierarchies – a Sun Ultra-2, a Sun Ultra-30, and an SGI Origin-2000. Table 1 provides details about these machines. We ran each experiment five times and selected the lowest time to minimize potential interference due to daemons and other processes running during the experiments.



(a) **Partitioned-array:** links and keys in separate arrays.



(b) **B-tree:** each node in the B-tree is a partitioned-array.



(c) **Hashtable:** each element in the overflow chain is a partitioned-array.

Fig. 1. Data structures used to represent trie nodes for large alphabets. Each partitioned-array is sized such that the array containing the keys fits in a single cache line.

To perform a controlled study of the impact of variation in cache characteristics on the performance of these algorithms, we simulated architectures that differed in cache line size (32/64-byte) and cache associativity (direct-mapped/2-way/4-way/8-way). We used the *msim* simulator [25] and assumed a 32KB L1 cache and a 2MB L2 cache. We assumed zero delay fetching data from L1 cache, a 5 cycle delay from L2 cache and a 36 cycle delay from main memory.

To drive these experiments, we used two datasets from different application domains and with different alphabet sizes. The first dataset was from the text processing domain and consisted of a trie containing all the words in the Web-

Machine	Cache line size	L1 size	L2 size	Proc. Clock	Mem BW
Sun Ultra-2	32-byte	16KB (direct-map)	2MB (direct-map)	167MHz	200MB/s
Sun Ultra-30	32-byte	32KB (direct-map)	2MB (direct-map)	250MHz	200MB/s
SGI Origin-2000	64-byte	32KB (direct-map)	2MB (2-way assoc)	195MHz	296MB/s

Table 1. Details of the real memory hierarchies used in experiments. The associativity of each cache is in parentheses. The memory bandwidth numbers have been measured using the STREAM *copy* benchmark written by John McCalpin (*http://www.cs.virginia.edu/stream*). The primary difference between the Ultra-2 and the Ultra-30 is the speed of the processor (the L1 cache is too small to impact the results of these experiments). The differences between the Sun machines and the SGI machine are: processor speed, cache linesize, L2 associativity and the memory bandwidth.

ster's Unabridged Dictionary.[5] The words were inserted in the order of their occurrence in the dictionary. Against this dictionary, we ran searches using all the words (212933) in Herman Melville's *Moby Dick*.[6] The words were searched for in their order of occurrence. This dataset had an alphabet of 54 symbols (lower and upper case English characters, hyphen and apostrophe). The average length of the words in the trie was 6.7 characters and the average length of the words in *Moby Dick* was 5.4 characters.

For this dataset, we used the algorithm for small alphabets and compared its performance to that of the ternary search tree proposed by Bentley and Sedgewick [7].[7] Bentley et al demonstrate that ternary search trees are somewhat faster than hashing for an English dictionary and up to five times faster than hashing for the DIMACS library call number datasets [7]. Clement et al [10] analyze the ternary search tree algorithm as a form of trie and conclude that it is an efficient data structure from an information-theoretic point of view. We did not consider the B-tree-based and the hashtable-based algorithms for this dataset as the maximum fanout for this dataset was only 54 (lower and upper case English characters, hyphen and apostrophe).

The second dataset was from the datamining domain and consisted of sets of retail transactions generated by Quest datamining dataset generator which we obtained from IBM Almaden [23]. For this application, the alphabet consists of the items that can be purchased in a grocery store and a string consists of a single retail transaction. Recall that the task is to determine the set of items that are frequently purchased together. Tries are used in this application to store

[5] An online version is available at *ftp://uiarchive.cso.uiuc.edu/pub/etext/gutenberg/-etext96/*.

[6] An online version is available at *ftp://uiarchive.cso.uiuc.edu/pub/etext/gutenberg/-etext91/moby.zip*.

[7] We obtained the code for the ternary search tree algorithm from *http://www.cs.princeton.edu/~rs/strings*.

the candidate itemsets and to help determine the frequent itemsets. The alphabet for this dataset contained 10,000 symbols (corresponding to 10,000 items); the average length of each string (i.e., the average number of items purchased per transaction) was 4 with a maximum length of 10. We created four pairs of transaction-sets. One transaction-set in each pair was used to construct the trie and the other was used for searching the trie. In each pair, the transaction-set used to search the trie was four times as large as the transaction-set used to create the trie and included it as a subset. The trie for the first pair of transaction-sets contained 20,000 transactions; the trie for the second pair of transaction-sets contained 40,000 transactions; the trie for the third pair of transaction-sets contained 80,000 transactions; and the trie for the fourth pair of transaction-sets contained 160,000 transactions. Compactly represented, these transactions can be stored in 1MB, 2MB, 4MB and 8MB respectively. For the machines used in our experiments, this corresponds to 0.5*cache_size, cache_size, 2*cache_size and 4*cache_size. This allowed us to explore the space of ratios between the cache size and the dataset size.

For this dataset, we used the algorithm for large alphabets and compared its performance against two alternatives. The first algorithm was a non-adaptive variant of our algorithms which used B-trees for all nodes (we refer to this as the B-tree-trie algorithm). This algorithm does not bound the depth of the B-tree. Our goal in this comparison was to determine how much advantage is provided by adaptivity. The second algorithm used variable-sized hashtables for all nodes (we refer to this as the hashtable-trie algorithm). The hashtable used for the root node had 1024 buckets; the hashtables used for nodes at every subsequent level reduced the number of buckets by a factor of two. We selected this algorithm for our experiments as an algorithm similar to this is proposed by several researchers for maintaining frequent itemsets for mining association rules [1, 2, 15]. We did not consider ternary search trees for this dataset due to the large alphabet – given that each ternary node matches two values (the third link is for "all else"), ternary search trees for such large alphabets are likely to be very deep (and therefore, likely to have poor memory performance).

We coded all our algorithms in C++ and compiled them using `g++ -O2`. For timing, we used the high-resolution `gethrtime()` call on the Sun machines and `gettimeofday()` on the SGI machine. We used these calls to determine the total time needed to perform *all* the trie lookups for each workload.

4 Results

Figure 2 compares the performance of the adaptive algorithm with that of the ternary search tree for the dictionary dataset. The adaptive algorithm significantly outperforms the ternary search tree on all three machines – by a factor of 1.7 on the Origin-2000 and by a factor of 4 on the Ultra-2 and the Ultra-30.

Figure 3 compares the performance of the adaptive algorithm with that of the B-tree-trie and the hashtable-trie algorithms for the datamining dataset. We note that the adaptive algorithm is faster than both the alternatives for all inputs

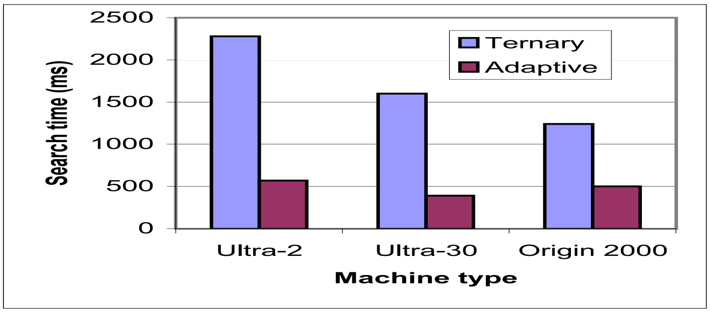


Fig. 2. Performance of the adaptive algorithm and the ternary search tree for the dictionary dataset.

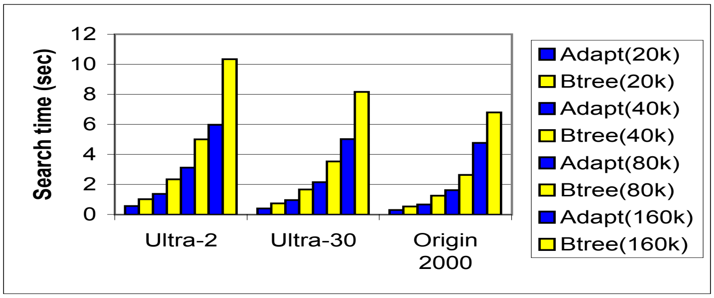
(by a factor of 1.4-1.9 over B-tree-trie and a factor of 1.2-1.5 over hashtable-trie). Note that the speedup falls off as the dataset size grows to several times the cache size. This indicates that the input is not localized to a small part of the trie.

There are three possible reasons for the performance advantage of the adaptive algorithms over the non-adaptive algorithms: (1) they execute fewer instructions, or (2) they spend less time waiting for data to be loaded into cache, or (3) both. For each of the dataset-platform-algorithm combination, we determined the number of instructions executed. For this, we built a cost model for each platform-algorithm combination and inserted code in each program to maintain an instruction counter. To build the cost model for each platform-algorithm combination, we disassembled the optimized binary code for that platform-algorithm combination and determined the number of instructions in each basic block.

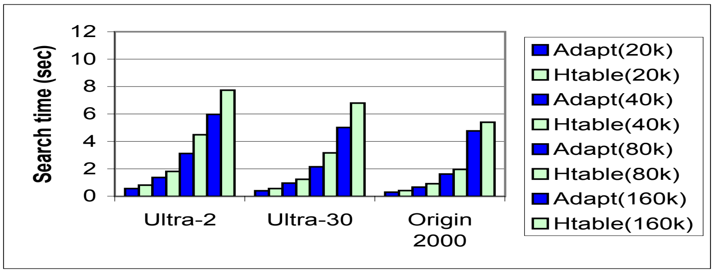
For the dictionary dataset, we found that the ternary search tree executes about 64 billion instructions whereas the adaptive algorithm executes about 62 billion instructions. Figure 4 compares the number of instructions executed by all three algorithms (adaptive, B-tree-trie and hashtable-trie) for the four datamining inputs. We note that the adaptive algorithm executes about the same number of instructions as the hashtable-trie and close to half the number of instructions executed by the B-tree trie. From these results, we conclude that the adaptive algorithms derive their performance advantage over otherwise efficient algorithms (ternary search tree for the dictionary dataset and the hashtable-trie for the datamining dataset) by making better use of the memory hierarchy.

We also compared the amount of space used by the different algorithms. For the dictionary dataset, the ternary search tree used 6.2 MB (388K nodes) and the adaptive algorithm used 6.0MB (217K nodes).[8] Table 2 presents the space used by different algorithms for the datamining dataset. From these results, we

[8] These numbers are for the Sun Ultras.



(a) Adaptive vs B-tree-trie



(b) Adaptive vs hashtable-trie

Fig. 3. Performance of the adaptive algorithm, the B-tree-trie and the hashtable-trie for the datamining dataset.

conclude that the perfomance advantage gained by the adaptive algorithms is not at the cost of additional space.

Num Transactions →	20K	40K	80K	160K
Adaptive	3.6MB	7.1MB	13.6MB	41.3MB
B-tree-trie	8MB	14.9MB	27.9MB	62.3MB
Hashtable-trie	36MB	60MB	95MB	190MB

Table 2. Space used by different algorithms for the datamining dataset.

One of the main features of the adaptive algorithms is that they change the representation of a node as its fanout changes. Figure 5 presents the distribution

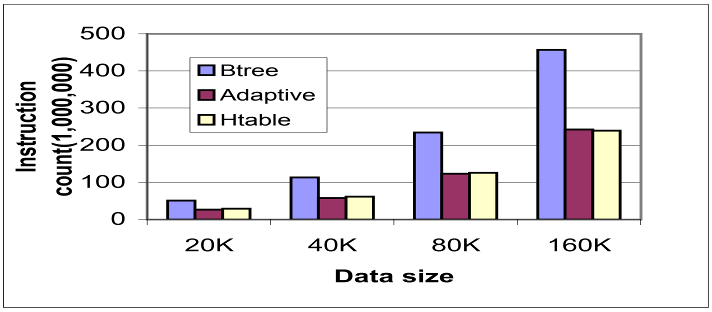


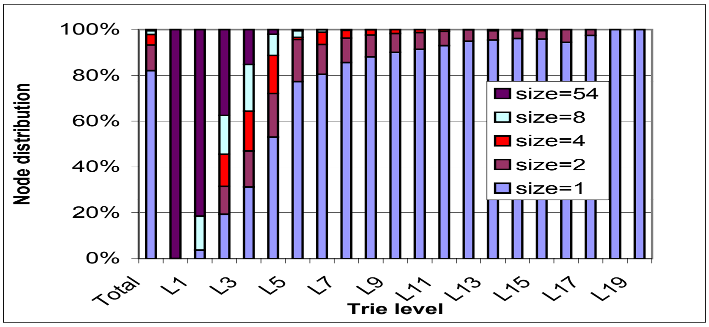
Fig. 4. Number of instructions executed by different algorithms for the datamining dataset.

of the different representations used for both datasets. The first bar in both graphs depicts the distribution of the three data structures in the entire trie; the bars to its right provide a level-by-level breakdown. We note that there is a wide variation in the distribution of data structures (and therefore fanout) on different levels and that on the whole, tries for both datasets are dominated by nodes with very small fanouts.

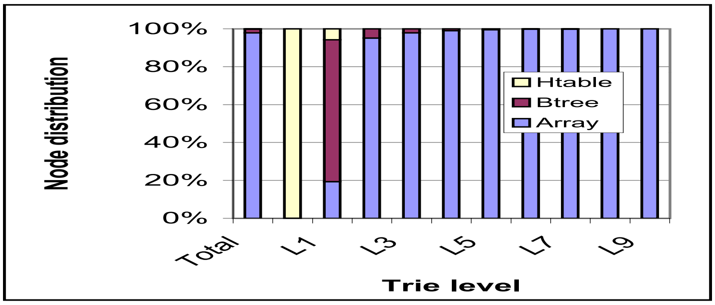
4.1 Impact of variation in cache characteristics

To evaluate the impact of variation in cache characteristics, we used the *msim* memory hierarchy simulator. Figure 6(a) examines the variation in simulated execution time as the cache line size changes. We used 32B and 64B as the alternatives as these are considered the most suitable cache line sizes. Smaller cache lines result in more frequent fetches from the main memory; larger cache lines require the processor to wait longer. In addition, larger cache lines are more likely to result in unnecessary data movement from/to the main memory. For this experiment, we assumed that both L1 and L2 caches were direct-mapped. We note that the performance of the adaptive algorithm does not change much with a change in the cache line size. This is not surprising as the line size is an explicit parameter of the algorithm.

Figure 6(b) examines the variation in simulated execution time as the cache associativity changes. For this experiment, we assumed 32-byte cache lines, and a direct-mapped L1 cache. We varied the associativity of the L2 cache with direct-mapped, 2-way, 4-way and 8-way associative configurations. Note that most modern L1 caches are direct-mapped and L2 caches are either direct-mapped or 2-way associative. We note that the the performance of the adaptive algorithm is insensitive to changes in cache associativity.



(a) Dictionary dataset

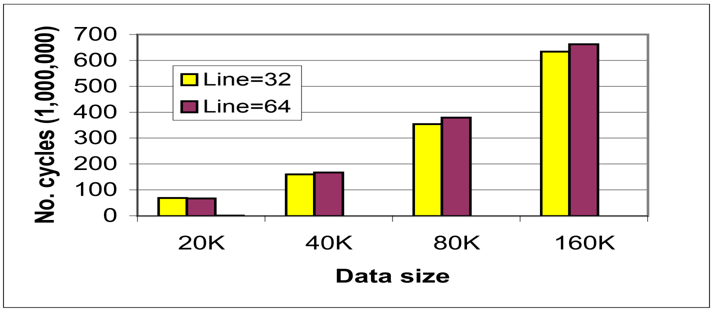


(b) Datamining dataset

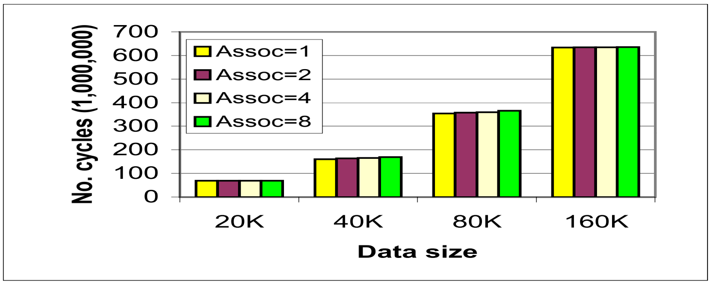
Fig. 5. Distribution of the data structures used by the adaptive algorithms. For the datamining dataset, the graphs present the distribution for the input with 160K transactions.

5 Related work

Previous work on adapting trie structures has taken one of four approaches. The first approach focuses on reducing the number of instructions executed by reducing the number of nodes and levels in the trie [5, 4, 6, 12, 19, 22]. The second approach views tries as collections of m-way vectors and focuses on reducing the space used by these vectors using a list-based representation or a sparse-matrix representation [3, 17, 20]. The third approach focuses on the data structures used to represent trie nodes with the goal of reducing both the space required and the number of instructions executed [7, 9]. Finally, there has been much recent interest in optimizing the tries used for address lookups in network routers [11, 21, 26]. The algorithms proposed by these researchers focus on reducing the number of memory accesses by reducing the number of levels in the trie and the fanout



(a) Variation in line size



(b) Variation in associativity

Fig. 6. Impact of variation in cache characteristics on the performance of the adaptive algorithm. For the results presented in (a), both L1 and L2 are assumed to be direct-mapped; for the results presented in (b), the cache line size is assumed to be 32-bytes and L1 is assumed to be direct-mapped.

at individual nodes. Our approach is closest to these in that we share their goal of reducing memory accesses; however, there are two main differences. First, these algorithms assume that the "symbols" to be matched at each level (so to speak) are not fixed and that the strings that form the trie can be arbitrarily subdivided. This assumption is correct for application that they focus on – IP address lookup in network routers.[9] We assume a fixed alphabet which is applicable to most other applications tries are used for. Second, these algorithms

[9] IP addresses (currently) are 32-bits long and arbitrary-length prefixes can appear in network routing tables.

focus on re-structuring the trie, while we focus on selecting the data structure for individual nodes.

6 Summary and conclusions

In this paper, we presented cache-efficient algorithms for trie search. The key features of these algorithms are: (1) they use multiple alternative data structures for representing trie nodes; (2) they adapt to changes in the fanout at a node by dynamically switching the data structure used to represent the node; (3) they determine the size and the layout of individual data structures based on the size of the symbols in the alphabet as well as characteristics of the cache(s).

Our evaluation indicates that these algorithms out-perform alternatives that are otherwise efficient but do not take cache characteristics into consideration. A similar conclusion is reached by Lamarca&Ladner in their paper on cache-efficient algorithms for sorting [18].

For the dictionary dataset, our algorithm was 1.7 times faster on the SGI Origin-2000 and 4 times faster on both the Sun Ultras compared to the ternary search tree algorithm proposed by Bentley and Sedgewick [7]. For the datamining dataset, our algorithm was 1.4-1.9 times faster than a B-tree-trie and 1.2-1.5 times faster than a hashtable-trie. A comparison of the number of instructions executed indicates that the algorithms presented in this paper derive their performance advantage over otherwise efficient algorithms by making better use of the memory hierarchy.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large data bases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–16, Washington, D.C., May 1993.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, 1994.
3. E. Ai-Sunwaiyel and E. Horowitz. Algorithms for trie compaction. *ACM Transactions on Database Systems*, 9(2):243–63, 1984.
4. J. Aoe, K. Marimoto, and T. Sato. An efficient implementation of trie structure. *Software Practice and Experience*, 22(9):695–721, 1992.
5. J. Aoe, K. Morimoto, M. Shishibori, and K. Park. A trie compaction algorithm for a large set of keys. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):476–91, 1996.
6. A. Appel and G. Jacobson. The world's fastest scrabble program. *Communications of the ACM*, 31(5):572–8, 1988.
7. J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of SODA '97*, 1997.
8. A. Blumer, J. Blumer, D. Haussler, and R. McConnel. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–95, 1987.
9. H. Clampett. Randomized binary searching with tree structures. *Communications of the ACM*, 7(3):163–5, 1964.

10. J. Clement, P. Flajolet, and B. Vallee. The analysis of hybrid trie structures. Technical Report 3295, INRIA, Nov 1997.
11. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *Computer Communication Review*, October 1997.
12. J. Dundas. Implementing dynamic minimal-prefix tries. *Software Practice and Experience*, 21(10):1027–40, 1991.
13. P. Flajolet and C. Puech. Partial match retrieval of multidimensional data. *Journal of the ACM*, 33(2):371–407, 1986.
14. G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: in Pascal and C.* Addison-Wesley, second edition, 1991.
15. E. Han, V. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proceedings of SIGMOD'97*, 1997.
16. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, second edition, 1996.
17. D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.
18. A. Lamarca and R. Ladner. The influence of caches on the performance of sorting. In *Proceedings of SODA'97*, 1997.
19. C. Lucchesi and T. Knowaltowski. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, 1993.
20. K. Maly. Compressed tries. *Communications of the ACM*, 19:409–15, 1976.
21. S. Nilsson and G. Karlsson. Fast address lookup for internet routers. In *Proceedings of IEEE Broadband Communications'98*, 1998.
22. J. Peterson. *Computer Programs for Spelling Correction.* Lecture Notes in Computer Science, Springer Verlag, 1980.
23. IBM Quest Data Mining Project. The Quest retail transaction data generator[10], 1996.
24. R. Rivest. Partial match retrieval algorithms. *SIAM Journal on Computing*, 5:19–50, 1976.
25. S. Sharma and A. Acharya. The *msim* memory hierarchy simulator. Personal Communication, 1997.
26. S. Venkatachary and G. Varghese. Faster IP Lookups Using Controlled Prefix Expansion. In *Proceedings of SIGMETRICS'98*, pages 1–10, 1998.
27. M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proceedings of SIGCOMM'97*, 1997.
28. M. Zaki, M. Ogihara, S. Parthasarthy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Proceedings of Supercomputing'96*, 1996.
29. M. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997.

A Background

A.1 Tries

Knuth [17] defines a trie as an m-way tree whose nodes are m-place vectors with components corresponding to individual symbols in the alphabet. Each node

[10] Available at *http://www.almaden.ibm.com/cs/quest/syndata.html.*

on level l represents the set of all strings that begin with certain sequence of l symbols; the node specifies an m-way branch depending on the $(l+1)th$ symbol. Figure 7 presents a trie that contains the words *and, an, at, dig* and *dot.*

The search for a string in a trie starts from the root node and proceeds as a descent in the tree structure. At each level, the fanout of the node is examined to determine if the the next symbol in the string appears as a label for one of the links. The search is successful is all the symbols in the strings are matched; the search is unsuccessful if at any node, the next symbol in the string does not appear as a label for one of the outgoing links.

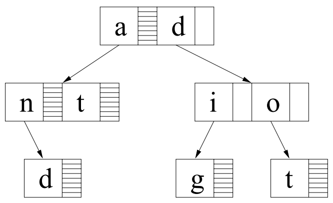


Fig. 7. Example of a trie. A shaded box next to a key indicates that a string ends at that key. For example, the shaded box next to n at the second level indicates that the string "an" ends at that node.

A.2 Modern memory hierarchies

Modern memory hierarchies attempt to bridge the difference between processor cycle time and memory access time by inserting one to three caches (referred to as L1, L2 and L3) between the processor and the main memory. In contemporary systems, data in the L1 cache can typically be accessed in a single processor cycle, data in the L2 cache in 5-13 processor cycles and data in the main memory in 25-150 processor cycles.

The size of caches usually grows with their distance from the processor – the caches closer to the processor are usually smaller than the caches further in the hierarchy. In contemporary systems, the L1 cache is typically between 8KB and 32KB; the L2 cache is between 96KB and 4MB and the L3 cache (if it exists) is multiple megabytes. Modern memory hierarchies satisfy the *inclusion* property – i.e., the cache at level i contains all the data contained in the cache at level $(i-1)$.

Caches are divided into *lines*; a reference to any memory location in a cache line results in the entire line being brought in from the next level in the memory hierarchy. Newer implementations include an optimization that brings in referenced memory location first and allows the processor to resume execution while the rest of the cache line is being brought in. Larger cache lines are useful for programs with strong spatial locality; however, each cache miss takes longer to satisfy. Typically, cache lines are 32 or 64 bytes long.

Associativity of a cache is defined to be the number of locations in this cache that a memory location in the next level in the memory hierarchy can be mapped to. A cache that has m such locations is referred to as *m-way associative*. Caches with ($m = 1$) are referred to as *direct-mapped*; caches with ($m = num\_lines\_in\_cache$) are referred to as *fully associative*. Caches in modern memory hierarchies are, typically, either direct-mapped or 2-way associative.

For more information on modern memory hierarchies, see [16].

Fast Priority Queues for Cached Memory

Peter Sanders

Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken, Germany
`sanders@mpi-sb.mpg.de`, Fax: (49) 681-9325 199

Abstract. The cache hierarchy prevalent in todays high performance processors has to be taken into account in order to design algorithms which perform well in practice. We advocates the approach to adapt external memory algorithms to this purpose. We exemplify this approach and the practical issues involved by engineering a fast priority queue suited to external memory and cached memory which is based on k-way merging. It improves previous external memory algorithms by constant factors crucial for transferring it to cached memory. Running in the cache hierarchy of a workstation the algorithm is at least two times faster than an optimized implementation of binary heaps and 4-ary heaps for large inputs.

1 Introduction

The mainstream model of computation used by algorithm designers in the last half century [18] assumes a sequential processor with unit memory access cost. However, the mainstream computers sitting on our desktops have increasingly deviated from this model in the last decade [10,11,13,17,19]. In particular, we usually distinguish at least four levels of memory hierarchy: A file of multi-ported *registers*, can be accessed in parallel in every clock-cycle. The *first-level cache* can still be accessed every one or two clock-cycles but it has only few parallel ports and only achieves the high throughput by pipelining. Therefore, the instruction level parallelism of super-scalar processors works best if most instructions use registers only. Currently, most first-level caches are quite small (8–64KB) in order to be able to keep them on chip and close to the execution unit. The *second-level cache* is considerably larger but also has an order of magnitude higher latency. If it is off-chip, its size is mainly constrained by the high cost of fast static RAM. The *main memory* is build of high density, low cost dynamic RAM. Including all overheads for cache miss, memory latency and translation from logical over virtual to physical memory addresses, a main memory access can be two orders of magnitude slower than a first level cache hit. Most machines have separate caches for data and code so that we can disregard instruction reads as long as the programs remain reasonably short.

Although the technological details are likely to change in the future, physical principles imply that fast memories must be small and are likely to be more expensive than slower memories so that we will have to live with memory hierarchies when talking about sequential algorithms for large inputs.

The general approach of this paper is to model one cache level and the main memory by the single disk single processor variant of the external memory model [22]. This model assumes an internal memory of size M which can access the external memory by transferring blocks of size B. We use the word pairs "cache line" and "memory block", "cache" and "internal memory", "main memory" and "external memory" and "I/O" and "cache fault" as synonyms if the context does not indicate otherwise. The only formal limitation compared to external memory is that caches have a fixed replacement strategy. In another paper, we show that this has relatively little influence on algorithm of the kind we are considering. Nevertheless, we henceforth use the term *cached memory* in order to make clear that we have a different model.

Despite of the far-reaching analogy between external memory and cached memory, a number of additional differences should be noted: Since the speed gap between caches and main memory is usually smaller than the gap between main memory and disks, we are careful to also analyze the work performed internally. The ratio between main memory size and first level cache size can be much larger than that between disk space and internal memory. Therefore, we will prefer algorithms which use the cache as economically as possible. Finally, we also discuss the remaining levels of the memory hierarchy but only do that informally in order to keep the analysis focussed on the most important aspects.

In Section 2 we present the basic algorithm for our *sequence heaps* data structure for priority queues[1]. The algorithm is then analyzed in Section 3 using the external memory model. For some m in $\Theta(M)$, k in $\Theta(M/B)$, any constant $\gamma > 0$ and $R = \lceil \log_k \frac{I}{m} \rceil \leq \mathcal{O}(M/B)$ it can perform I insertions and up to I `deleteMins` using $I(2R/B + \mathcal{O}(1/k + (\log k)/m))$ I/Os and $I(\log I + \log R + \log m + \mathcal{O}(1))$ key comparisons. In another paper, we show that similar bounds hold for cached memory with a-way associative caches if k is reduced by $\mathcal{O}(B^{1/a})$. In Section 4 we present refinements which take the other levels of the memory hierarchy into account, ensure almost optimal memory efficiency and where the amortized work performed for an operation depends only on the current queue size rather than the total number of operations. Section 5 discusses an implementation of the algorithm on several architectures and compares the results to other priority queue data structures previously found to be efficient in practice, namely binary heaps and 4-ary heaps.

Related Work

External memory algorithms are a well established branch of algorithmics (e.g. [21,20]). The external memory heaps of Teuhola and Wegner [23] and the fishspear data structure [9] need $\Theta(B)$ less I/Os than traditional priority queues like binary heaps. Buffer search trees [1] were the first external memory priority

[1] A data structure for representing a totally ordered set which supports insertion of elements and deletion of the minimal element.

queue to reduce the number of I/Os by another factor of $\Theta(\log \frac{M}{B})$ thus meeting the lower bound of $\mathcal{O}((I/B)\log_{M/B} I/M)$ I/Os for I operations (amortized). But using a full-fledged search tree for implementing priority queues may be considered wasteful. The heap-like data structures by Brodal and Katajainen, Crauser et. al. and Fadel et. al. [3,7,8] are more directly geared to priority queues and achieve the same asymptotic bounds, one [3] even per operation and not in an amortized sense. Our sequence heap is very similar. In particular, it can be considered a simplification and reengineering of the "improved array-heap" [7]. However, sequence heaps are more I/O-efficient by a factor of about three (or more) than [1,3,7,8] and need about a factor of two less memory than [1,7,8].

2 The Algorithm

Merging k sorted sequences into one sorted sequence (k-way merging) is an I/O efficient subroutine used for sorting – both for external [14] and cached memory [16]. The basic idea of sequence heaps is to adapt k-way merging to the related but more dynamical problem of priority queues.

Let us start with the simple case, that at most km insertions take place where m is the size of a buffer which fits into fast memory. Then the data structure could consist of k sorted sequences of length up to m. We can use k-way merging for deleting a batch of the m smallest elements from k sorted sequences. The next m deletions can then be served from a buffer in constant time.

To allow an arbitrary mix of insertions and deletions, we maintain a separate binary heap of size up to m which holds the recently inserted elements. Deletions have to check whether the smallest element has to come from this *insertion buffer*. When this buffer is full, it is sorted and the resulting sequence becomes one of sequences for the k-way merge.

Up to this point, sequence heaps and the earlier data structures [3,7,8] are almost identical. Most differences are related to the question how to handle more than km elements. We cannot increase m beyond M since the insertion heap would not fit into fast memory. We cannot arbitrarily increase k since eventually k-way merging would start to incur cache faults. Sequence heaps use the approach to make room by merging all the k sequences producing a larger sequence of size up to km [3,7].

Now the question arises how to handle the larger sequences. We adopt the approach used for *improved array-heaps* [7] to employ R *merge groups* $G_1, \ldots, G_R$ where G_i holds up to k sequences of size up to mk^{i-1}. When group G_i overflows, all its sequences are merged and the resulting sequence is put into group G_{i+1}.

Each group is equipped with a *group buffer* of size m to allow batched deletion from the sequences. The smallest elements of these buffers are deleted in batches of size $m' \ll m$. They are stored in the *deletion Buffer*. Fig. 1 summarizes the data structure. We now have enough information to explain how deletion works:

DeleteMin: The smallest elements of the deletion buffer and insertion buffer are compared and the smaller one is deleted and returned. If this empties the deletion buffer, it is refilled from the group buffers using an R-way merge. Before

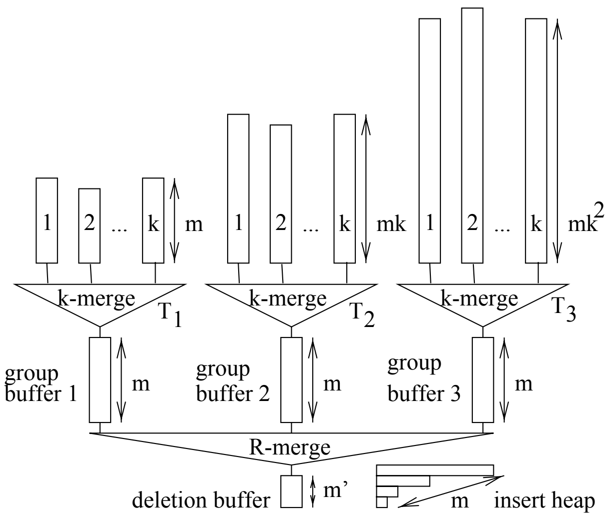


Fig. 1. Overview of the data structure for sequence heaps for $R = 3$ merge groups.

the refill, group buffers with less than m' elements are refilled from the sequences in their group (if the group is nonempty).

DeleteMin works correctly provided the data structure fulfills the heap property, i.e., elements in the group buffers are not smaller than elements in the deletion buffer, and, in turn, elements in a sorted sequence are not smaller than the elements in the respective group buffer. Maintaining this invariant is the main difficulty for implementing insertion:

Insert: New elements are inserted into the insert heap. When its size reaches m its elements are sorted (e.g. using merge sort or heap sort). The result is then merged with the concatenation of the deletion buffer and the group buffer 1. The smallest resulting elements replace the deletion buffer and group buffer 1. The remaining elements form a new sequence of length at most m. The new sequence is finally inserted into a free slot of group G_1. If there is no free slot initially, G_1 is emptied by merging all its sequences into a single sequence of size at most km which is then put into G_2. The same strategy is used recursively to free higher level groups when necessary. When group G_R overflows, R is incremented and a new group is created. When a sequence is moved from one group to the other, the heap property may be violated. Therefore, when G_1 through G_i have been emptied, the group buffers 1 through $i+1$ are merged, and put into G_1.

The latter measure is one of the few differences to the improved array heap [7] where the invariant is maintained by merging the new sequence and the group buffer. This measure almost halves the number of required I/Os.

For cached memory, where the speed of internal computation matters, it s also crucial how to implement the operation of k-way merging. We propose to use the "loser tree" variant of the selection tree data structure described by Knuth [14, Section 5.4.1]: When there are k' nonempty sequences, it consists of a binary tree with k' leaves. Leaf i stores a pointer to the current element of sequence i. The current keys of each sequence perform a tournament. The winner is passed up the tree and the key of the loser and the index of its leaf are stored in the inner node. The overall winner is stored in an additional node above the root. Using this data structure, the smallest element can be identified and replaced by the next element in its sequence using $\lceil \log k \rceil$ comparisons. This is less than the heap of size k assumed in [7,8] would require. The address calculations and memory references are similar to those needed for binary heaps with the noteworthy difference that the memory locations accessed in the loser tree are predictable which is not the case when deleting from a binary heap. The instruction scheduler of the compiler can place these accesses well before the data is needed thus avoiding pipeline stalls, in particular if combined with loop unrolling.

3 Analysis

We start with an analysis for of the number of I/Os in terms of B, the parameters k, m and m' and an arbitrary sequence of `insert` and `deleteMin` operations with I insertions and up to I `deleteMins`. We continue with the number of key comparisons as a measure of internal work and then discuss how k, m and m' should be chosen for external memory and cached memory respectively. Adaptions for memory efficiency and many accesses to relatively small queues are postponed to Section 4.

We need the following observation on the minimum intervals between tree emptying operations in several places:

Lemma 1. *Group G_i can overflow at most every $m(k^i - 1)$ insertions.*

Proof. The only complication is the slot in group G_1 used for invalid group buffers. Nevertheless, when groups G_1 through G_i contain k sequences each, this can only happen if $\sum_{j=1}^{R} m(k-1)k^{j-1}So = m(k^i - 1)$ insertions have taken place. ■

In particular, since there is room for m insertions in the insertion buffer, there is a very simple upper bound for the number of groups needed:

Corollary 1. $R = \lceil \log_k \frac{I}{m} \rceil$ *groups suffice.*

We analyze the number of I/Os based on the assumption that the following information is kept in internal memory: The insert heap; the deletion buffer; a merge buffer of size m; group buffers 1 and R; the loser tree data for groups G_R, G_{R-1} (we assume that $k(B+2)$ units of memory suffice to store the blocks of the k sequences which are currently accessed and the loser tree information itself); a corresponding amount of space shared by the remaining $R-2$ groups and data for merging the R group buffers.[2]

Theorem 1. *If* $R = \lceil \log_k(I/m) \rceil$, $4m+m'+(3k+R)(B+2) < M$ *and* $k(B+2) \leq m-m'$ *then*

$$I\left(\frac{2R}{B} + \mathcal{O}\left(\frac{1}{k} + \frac{\log k}{m}\right)\right)$$

I/Os suffice to perform any sequence of I `insert`*s and up to I* `deleteMin`*s on a sequence heap.*

Proof. Let us first consider the I/Os performed for an element moving on the following *canonical* data path: It is first inserted into the insert buffer and then written to a sequence in group G_1 in a batched manner, i.e, we charge $1/B$ I/Os to the insertion of this element. Then it is involved in emptying groups until it arrives in group G_R. For each emptying operation it is involved into one batched read and one batched write, i.e., we charge $2(R-1)/B$ I/Os for tree emptying operations. Eventually, it is read into group buffer R. We charge $1/B$ I/Os for this. All in all, we get a charge of $2R/B$ I/Os for each insertion.

What remains to be shown is that the remaining I/Os only contribute lower order terms or replace I/Os done on the canonical path. When an element travels through group G_{R-1} then $2/B$ I/Os must be charged for writing it to group buffer $R-1$ and later reading it when refilling the deletion buffer. However, the $2/B$ I/Os saved because the element is not moved to group G_R can pay for this charge. When an element travels through group buffer $i \leq R-2$, the additional $c \geq 2/B$ I/Os saved compared to the canonical path can also pay for the cost of swapping loser tree data for group G_i. The latter costs $2k(B+2)/B$ I/Os which can be divided among at least $m - m' \geq k(B+2)$ elements removed in one batch.

When group buffer $i \geq 2$ becomes invalid so that it must be merged with other group buffers and put back into group G_1, this causes a direct cost of $\mathcal{O}(m/B)$ I/Os and we must charge a cost of $\mathcal{O}(im/B)$ I/Os because these elements are thrown back $\mathcal{O}(i)$ steps on their path to the deletion buffer. Although an element may move through all the R groups we do not need to charge $\mathcal{O}(Rm/B)$ I/Os for small i since this only means that the shortcut originally taken by this element compared to the canonical path is missed. The remaining overhead can be charged to the $m(k-1)k^{j-2}$ insertions which have filled group G_{i-1}. Summing over all groups, each insertions gets an additional charge of $\sum_{i=2}^{R} \mathcal{O}(im/B)/(m(k-1)k^{j-2}) = \mathcal{O}(1/k)$. Similarly, invalidations of group buffer 1 give a charge $\mathcal{O}(1/k)$ per insertion.

We need $\mathcal{O}(\log k)$ I/Os for inserting a new sequence into the loser tree data structure. When done for tree 1, this can be amortized over m insertions. For tree $i > 1$ it can be amortized over $m(k^{i-1}-1)$ elements by Lemma 1. For an

[2] If we accept $\mathcal{O}(1/B)$ more I/Os per operation it would suffice to swap between the insertion buffer plus a constant number of buffer blocks and one loser tree with k sequence buffers in internal memory.

element moving on the canonical path, we get an overall charge of $\mathcal{O}(\log k/m) + \sum_{i=2}^{R} m(k^{i-1} - 1)\log k = \mathcal{O}((\log k)/m)$ per insertion.

Overall we get a charge of $2R/B + \mathcal{O}(1/k + \log k/m)$. per insertion. ■

We now estimate the number of key comparisons performed. We believe this is a good measure for the internal work since in efficient implementations of priority queues for the comparison model, this number is close to the number of unpredictable branch instructions (whereas loop control branches are usually well predictable by the hardware or the compiler) and the number of key comparisons is also proportional to the number of memory accesses. These two types of operations often have the largest impact on the execution time since they are the most severe limit to instruction parallelism in a super-scalar processor. In order to avoid notational overhead by rounding, we also assume that k and m are powers of two and that I is divisible by mk^{R-1}. A more general bound would only be larger by a small additive term.

Theorem 2. *With the assumptions from Theorem 1 at most* $I(\log I + \lceil \log R \rceil + \log m + 4 + m'/m + \mathcal{O}((\log k)/k))$ *key comparisons are needed. For average case inputs "*$\log m$*" can be replaced by* $\mathcal{O}(1)$.

Proof. Insertion into the insertion buffer takes $\log m$ comparisons at worst and $\mathcal{O}(1)$ comparisons on the average. Every `deleteMin` operation requires a comparison of the minimum of the insertion buffer and the deletion buffer. The remaining comparisons are charged to insertions in an analogous way to the proof of Theorem 1. Sorting the insertion buffer (e.g. using merge sort) takes $m \log m$ comparisons and merging the result with the deletion buffer and group buffer 1 takes $2m + m'$ comparisons. Inserting the sequence into a loser tree takes $\mathcal{O}(\log k)$ comparisons. Emptying groups takes $(R-1)\log k + \mathcal{O}(R/k)$ comparisons per element. Elements removed from the insertion buffer take up to $2 \log m$ comparisons. But those need not be counted since we save all further comparisons on them. Similarly, refills of group buffers other than R have already been accounted for by our conservative estimate on group emptying cost. Group G_R only has degree $I/(mk^{R-1})$ so $\lceil \log I - (R-1)\log k - \log m \rceil$ comparisons per element suffice. Using similar arguments as in the proof of Theorem 1 it can be shown that inserting sequences into the loser trees leads to a charge of $\mathcal{O}((\log k)/m)$ comparisons per insertion and invalidating group buffers costs $\mathcal{O}((\log k)/k)$ comparisons per insertion. Summing all the charges made yields the bound to be proven. ■

For external memory one would choose $m = \Theta(M)$ and $k = \Theta(M/B)$. In another paper we show that k should be a factor $\mathcal{O}(B^{1/a}/\delta)$ smaller on a-way associative caches in order to limit the number of cache faults to $(1+\delta)$ times the number of I/Os performed by the external memory algorithm. This requirement together with the small size of many first level caches and TLBs[3] explains why

[3] *T*ranslation *L*ook-aside *B*uffers store the physical position of the most recently used virtual memory pages.

we may have to live with a quite small k. This observation is the main reason why we did not pursue the simple variant of the array heap described in [7] which needs only a single merge group for all sequences. This merge group would have to be about a factor R larger however.

4 Refinements

Memory Management: A sequence heap can be implemented in a memory efficient way by representing sequences in the groups as singly linked lists of memory pages. Whenever a page runs empty, it is pushed on a stack of free pages. When a new page needs to be allocated, it is popped from the stack. If necessary, the stack can be maintained externally except for a single buffer block. Using pages of size p, the external sequences of a sequence heap with R groups and N elements occupy at most $N + kpR$ memory cells. Together with the measures described above for keeping the number of groups small, this becomes $N + kp \log_k N/m$. A page size of m is particularly easy to implement since this is also the size of the group buffers and the insertion buffer. As long as $N = \omega(km)$ this already guarantees asymptotically optimal memory efficiency, i.e., a memory requirement of $N(1 + o(1))$.

Many Operations on Small Queues: Let N_i denote the queue size before the i-th operation is executed. In the earlier algorithms [3,7,8] the number of I/Os is bounded by $\mathcal{O}(\sum_{i \le I} \log_k N_i/m)$. For certain classes of inputs, $\sum_{i \le I} \log_k N_i/m$ can be considerably less than $I \log_k I/m$. However, we believe that for most applications which require large queues at all, the difference will not be large enough to warrant significant constant factor overheads or algorithmic complications. We have therefore chosen to give a detailed analysis of the basic algorithm first and to outline an adaption yielding the refined asymptotic bound here: Similar to [7], when a new sequence is to be inserted into group G_i and there is no free slot, we first look for two sequences in G_i whose sizes sum to less than mk^{i-1} elements. If found, these sequences are merged, yielding a free slot. The merging cost can be charged to the `deleteMins` which caused the sequences to get so small. Now G_i is only emptied when it contains at least $mk^i/2$ elements and the I/Os involved can be charged to elements which have been inserted when G_i had at least size $mk^{i-1}/4$. Similarly, we can "tidy up" a shrinking queue: When there are R groups and the total size of the queue falls below $mk^{R-1}/4$, empty group G_R and insert the resulting sequence into group G_{R-1} (if there is no free slot in group G_{R-1} merge any two of its sequences first).

Registers and Instruction Cache: In all realistic cases we have $R \le 4$ groups. Therefore, instruction cache and register file are likely to be large enough to efficiently support a fast R-way merge routine for refilling the deletion buffer which keeps the current keys of each stream in registers.

Second Level Cache: So far, our analysis assumes only a single cache level. Still, if we assume this level to be the first level cache, the second level cache may have some influence. First, note that the group buffers and the loser trees with their group buffers are likely to fit in second level cache. The second level cache may also be large enough to accommodate all of group G_1 reducing the costs for $2/B$ I/Os per insert. We get a more interesting use for the second level cache if we assume its bandwidth to be sufficiently high to be no bottleneck and then look at inputs where deletions from the insertion buffer are rare (e.g. sorting). Then we can choose $m = \mathcal{O}(M_2)$ if M_2 is the size of the second level cache. Insertions have high locality if the $\log m$ cache lines currently accessed by them fit into first level cache and no operations on deletion buffers and group buffers use random access.

High Bandwidth Disks: When the sequence heap data structure is viewed as a classical external memory algorithm we would simply use the main memory size for M. But our measurements in Section 5 indicate that large binary heaps as an insertion buffer may be too slow to match the bandwidth of fast parallel disk subsystems. In this case, it is better to modify a sequence heap ooptimized for cache and main memory by using specialized external memory implementations for the larger groups. This may involve buffering of disk blocks, explicit asynchronous I/O calls and perhaps prefetching code and randomization for supporting parallel disks [2]. Also, the number of I/Os may be reduced by using a larger k inside these external groups. If this degrades the performance of the loser tree data structure too much, we can insert another heap level, i.e., split the high degree group into several low degree groups connected together over sufficiently large level-2 group buffers and another merge data structure.

Deletions of non-minimal elements can be performed by maintaining a separate sequence heap of deleted elements. When on a `deleteMin`, the smallest element of the main queue and the delete-queue coincide, both are discarded. Hereby, insertions and deletions cost only one comparison more than before, if we charge a delete for the costs of one insertion and two `deleteMin`s (note that the latter are much cheaper than an insertion). Memory overhead can be kept in bounds by completely sorting both queues whenever the size of the queue of deleted elements exceeds some fraction of the size of the main queue. During this sorting operation, deleted keys are discarded. The resulting sorted sequence can be put into group G_R. All other sequences and and the deletion heap are empty then.

5 Implementation and Experiments

We have implemented sequence heaps as a portable C++ template class for arbitrary key-value-pairs. Currently, sequences are implemented as a single array. The performance of our sequence heap mainly stems on an efficient implementation of the k-way merge using loser trees, special routines for 2-way, 3-way and 4-way merge and binary heaps for the insertion buffer. The most important optimizations turned out to be (roughly in this order): Making live for the compiler easy; use of *sentinels*, i.e., dummy elements at the ends of sequences and heaps which save special case tests; loop unrolling.

5.1 Choosing Competitors

When an author of a new code wants to demonstrate its usefulness experimentally, great care must be taken to choose a competing code which uses one of the best known algorithms and is at least equally well tuned. We have chosen implicit binary heaps and aligned 4-ary heaps. In a recent study [15], these two algorithms outperform the pointer based data structures splay tree and skew heap by more than a factor two although the latter two performed best in an older study [12]. Not least because we need the same code for the insertion buffer, binary heaps were coded perhaps even more carefully than the remaining components – binary heaps are the only part of the code for which we took care that the assembler code contains no unnecessary memory accesses, redundant computations and a reasonable instruction schedule. We also use the *bottom up heuristics* for `deleteMin`: Elements are first lifted up on a min-path from the root to a leaf, the leftmost element is then put into the freed leaf and is finally bubbled up. Note that binary heaps with this heuristics perform only $\log N + \mathcal{O}(1)$ comparisons for an insertions plus a `deleteMin` on the average which is close to the lower bound. So in flat memory it should be hard to find a comparison based algorithm which performs significantly better for average case inputs. For small queues our binary heaps are about a factor two faster than a more straightforward non-recursive adaption of the textbook formulation used by Cormen, Leiserson and Rivest [5].

Aligned 4-ary heaps have been developed at the end using the same basic approach as for binary heaps, in particular, the bottom up heuristics is also used. The main difference is that the data gets aligned to cache lines and that more complex index computations are needed.

All source codes are available electronically under `http://www.mpi-sb.mpg.de/~sanders/programs/`.

5.2 Basic Experiments

Although the programs were developed and tuned on SPARC processors, sequence heaps show similar behavior on all recent architectures that were available for measurements. We have run the same code on a SPARC, MIPS, Alpha and Intel processor. It even turned out that a single parameter setting – $m' = 32$, $m = 256$ and $k = 128$ works well for all these machines.[4] Figures 2, 3, 4 and 5 respectively show the results.

All measurements use random 32 bit integer keys and 32 bit values. For a maximal heap size of N, the operation sequence $(\texttt{insert deleteMin insert})^N$ $(\texttt{deleteMin insert deleteMin})^N$ is executed. We normalize the amortized execution time per `insert`-`deleteMin`-pair – $T/(6N)$ – by dividing by $\log N$. Since all algorithms have an "flat memory" execution time of $c \log N + \mathcal{O}(1)$ for some constant c, we would expect that the curves have a hyperbolic form and converge

[4] By tuning k and m, performance improvements around 10 % are possible, e.g., for the Ultra and the PentiumII, $k = 64$ are better.

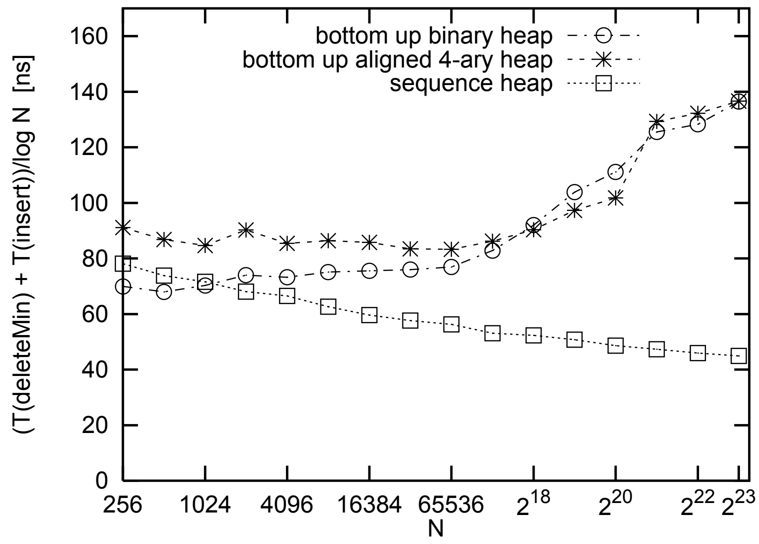


Fig. 2. Performance on a Sun Ultra-10 desktop workstation with 300 MHz Ultra-SparcIIi processor (1st-level cache: $M = 16\text{KByte}$, $B = 16\text{Byte}$; 2nd-level cache: $M = 512\text{KByte}$, $B = 32\text{Byte}$) using Sun Workshop C++ 4.2 with options `-fast -O4`.

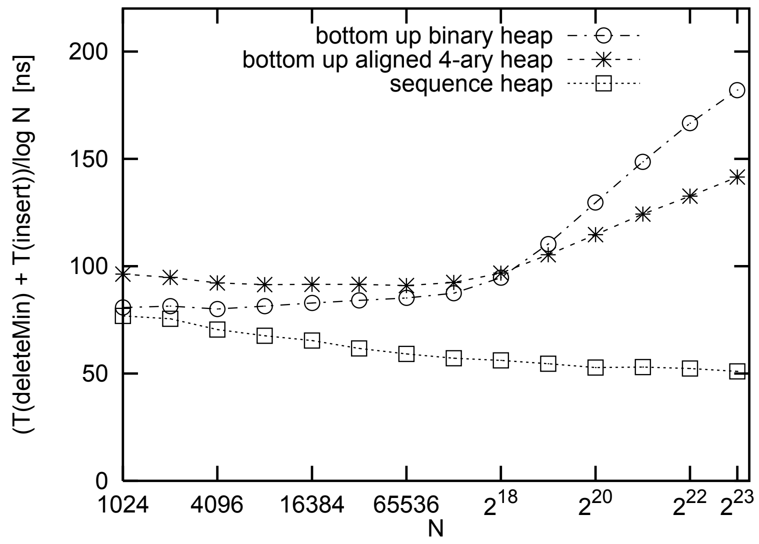


Fig. 3. Performance on a 180 MHz MIPS R10000 processor. Compiler: `CC -r10000 -n32 -mips4 -O3`.

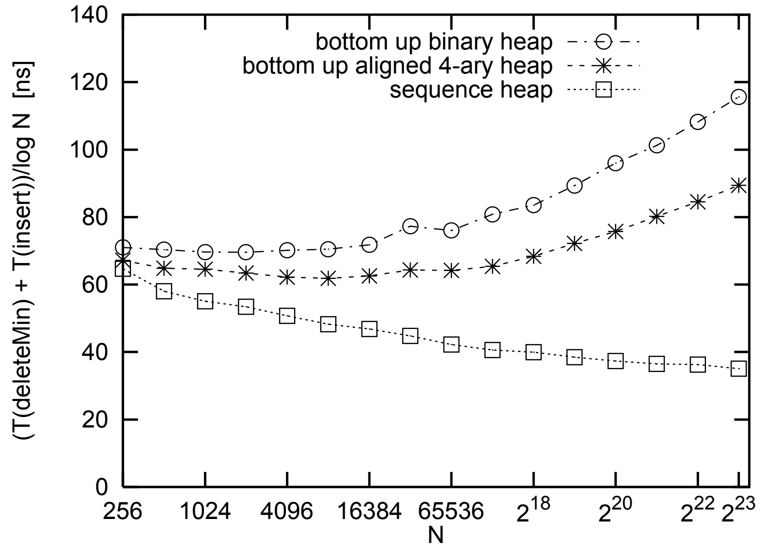


Fig. 4. Performance on a 533 MHz DEC-Alpha-21164 processor. Compiler: `g++ -O6`.

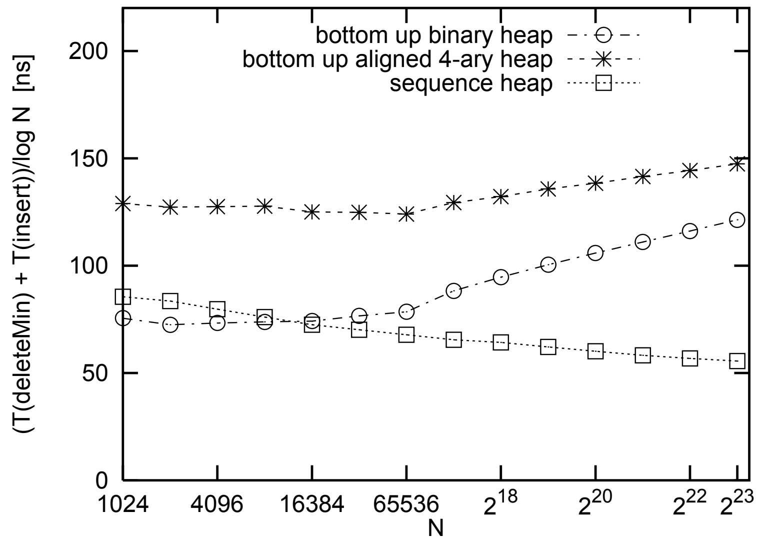


Fig. 5. Performance on a 300 MHz Intel Pentium II processor. Compiler: `g++ -O6`.

to a constant for large N. The values shown are averages over at least 10 trials. (More for small inputs to avoid problems due to limited clock resolution.) In order to minimize the impact of other processes and virtual memory management, a warm-up run is made before each measurement and the programs are run on (almost) unloaded machines.

Sequence heaps show the behavior one would expect for flat memory – cache faults are so rare that they do not influence the execution time very much. In Section 5.4, we will see that the decrease in the "time per comparison" is not quite so strong for other inputs.

On all machines, binary heaps are equally fast or slightly faster than sequence heaps for small inputs. While the heap still fits into second level cache, the performance remains rather stable. For even larger queues, the performance degradation accelerates. Why is the "time per comparison" growing about linearly in $\log n$? This is easy to explain. Whenever the queue size doubles, there is another layer of the heap which does not fit into cache, contributing a constant number of cache faults per `deleteMin`. For $N = 2^{23}$, sequence heaps are between 2.1 and 3.8 times faster than binary heaps.

We consider this difference to be large enough to be of considerable practical interest. Furthermore, the careful implementation of the algorithms makes it unlikely that such a performance difference can be reversed by tuning or use of a different compiler.[5] (Both binary heaps and sequence heaps could be slightly improved by replacing index arithmetics by arithmetics with address offsets. This would save a single register-to-register shift instruction per comparison and is likely to have little effect on super-scalar machines.) Furthermore, the satisfactory performance of binary heaps on small inputs shows that for large inputs, most of the time is spent on memory access overhead and coding details have little influence on this.

5.3 4-ary Heaps

The measurements in figures 2 through 5 largely agree with the most important observation of LaMarca and Ladner [15]: since the number of cache faults is about halved compared to binary heaps, 4-ary heaps have a more robust behavior for large queues. Still, sequence heaps are another factor between 2.5 and 2.9 faster for very large heaps since they reduce the number of cache faults even more. However, the relative performance of our binary heaps and 4-ary heaps seems to be a more complicated issue than in [15]. Although this is not the main concern of this paper we would like to offer an explanation:

Although the bottom up heuristics improves both binary heaps and 4-ary heaps, binary heaps profit much more. Now, binary heaps need less instead of more comparisons than 4-ary heaps. Concerning other instruction counts, 4-ary

[5] For example, in older studies, heaps and loser trees may have looked bad compared to pointer based data structures if the compiler generates integer division operations for halving an index or integer multiplications for array indexing.

heaps only save on memory write instructions while they need more complicated index computations.

Apparently, on the Alpha which has the highest clock speed of the machines considered, the saved write instructions shorten the critical path while the index computations can be done in parallel to slow memory accesses (spill code).

On the other machines, the balance turns into the other direction. In particular, the Intel architecture lacks the necessary number of registers so that the compiler has to generate a large number of additional memory accesses. Even for very large queues, this handicap is never made up for.

The most confusing effect is the jump in the execution time of 4-ary heaps on the SPARC for $N > 2^{20}$. Nothing like this is observed on the other machines and this effect is hard to explain by cache effects alone since the input size is already well beyond the size of the second level cache. We suspect some problems with virtual address translation which also haunted the binary heaps in an earlier version.

5.4 Long Operation Sequences

Our worst case analysis predicts a certain performance degradation if the number of insertions I is much larger than the size of the heap N. However, in Fig. 6 it can be seen that the contrary can be true for random keys.

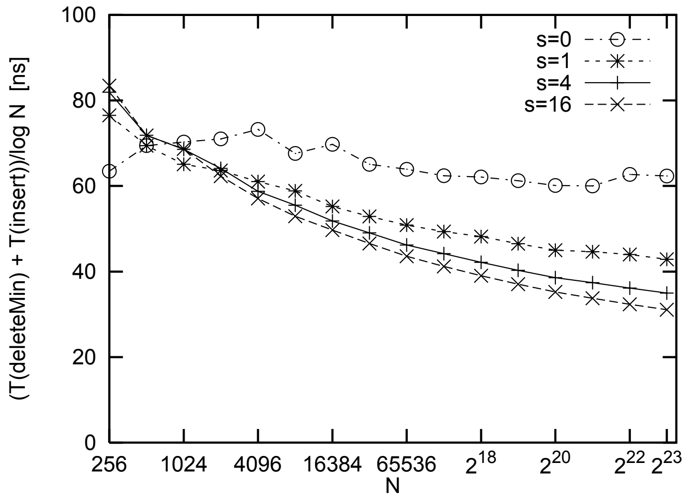


Fig. 6. Performance of sequence heaps using the same setup as in Fig. 2 but using different operation sequences: (`insert` (`deleteMin insert`)^s)^N (`deleteMin` (`insert deleteMin`)^s)^N for $s \in \{0, 1, 4, 16\}$. For $s = 0$ we essentially get heap-sort with some overhead for maintaining useless group and deletion buffers. In Fig. 2 we used $s = 1$.

For a family of instances with $I = 33N$ where the heap grows and shrinks very slowly, we are almost two times faster than for $I = N$. The reason is that new elements tend to be smaller than most old elements (the smallest of the old elements have long been removed before). Therefore, many elements never make it into group G_1 let alone the groups for larger sequences. Since most work is performed while emptying groups, this work is saved. A similar locality effect has been observed and analyzed for the Fishspear data structure [9]. Binary heaps or 4-ary heaps do not have this property. (They even seem to get slightly slower.) For $s = 0$ this locality effect cannot work. So that these instances should come close to the worst case.

6 Discussion

Sequence heaps may currently be the fastest available data structure for large comparison based priority queues both in cached and external memory. This is particularly true, if the queue elements are small and if we do not need deletion of arbitrary elements or decreasing keys. Our implementation approach, in particular k-way merging with loser trees can also be useful to speed up sorting algorithms in cached memory.

In the other cases, sequence heaps still look promising but we need experiments encompassing a wider range of algorithms and usage patterns to decide which algorithm is best. For example, for monotonic queues with integer keys, radix heaps look promising. Either in a simplified, average case efficient form known as calendar queues [4] or by adapting external memory radix heaps [6] to cached memory in order to reduce cache faults.

We have outlined how the algorithm can be adapted to multiple levels of memory and parallel disks. On a shared memory multiprocessor, it should also be possible to achieve some moderate speedup by parallelization (e.g. one processor for the insertion and deletion buffer and one for each group when refilling group buffers; all processors collectively work on emptying groups).

Acknowledgements

I would like to thank Gerth Brodal, Andreas Crauser, Jyrki Katajainen and Ulrich Meyer for valuable suggestions. Ulrich Rüde from the University of Augsburg provided access to an Alpha processor.

References

1. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *4th Workshop on Algorithms and Data Structures*, number 955 in LNCS, pages 334–345. Springer, 1995.
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. Gerth Stølting Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In *6th Scandinavian Workshop on Algorithm Theory*, number 1432 in LNCS, page 107ff. Springer Verlag, Berlin, 1998.
4. R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.
6. A. Crauser and K. Mehlhorn et. al. On the performance of LEDA-SM. Technical Report MPI-I-98-1-028, Max Planck Institute for Computer Science, 1998.
7. A. Crauser, P. Ferragina, and U. Meyer. Efficient priority queues in external memory. working paper, October 1997.
8. R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. External heaps combined with effective buffering. In *4th Australasian Theory Symposium*, volume 19-2 of *Australian Computer Science Communications*, pages 72–78. Springer, 1997.
9. M. J. Fischer and M. S. Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM*, 41(1):3–30, 1994.
10. J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, 1996.
11. Intel Corporation, P.O. Box 5937, Denver, CO, 80217-9808, `http://www.intel.com`. *Intel Archtecture Software Developer's Manual. Volume I: Basic Architecture*, 1997. Ordering Number 243190.
12. D. Jones. An empirical comparison of priority-queue and event set implementations. *Communications of the ACM*, 29(4):300–311, 1986.
13. J. Keller. The 21264: A superscalar alpha processor with out-of-order execution. In *Microprocessor Forum*, October 1996.
14. D. E. Knuth. *The Art of Computer Programming — Sorting and Searching*, volume 3. Addison Wesley, 1973.
15. A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1(4), 1996.
16. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *8th ACM-SIAM Symposium on Discrete Algorithm*, pages 370–379, 1997.
17. MIPS Technologies, Inc. *R10000 Microprocessor User's Manual*, 2.0 edition, 1998. `http://www.mips.com`.
18. J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.
19. Sun Microsystems. *UltraSPARC-IIi User's Manual*, 1997.
20. D. E. Vengroff. *TPIE User Manual and Reference*, 1995. `http://www.cs.duke.edu/~dev/tpie_home_page.html`.
21. J. S. Vitter. External memory algorithms. In *6th European Symposium on Algorithms*, number 1461 in LNCS, pages 1–25. Springer, 1998.
22. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12(2–3):110–147, 1994.
23. L. M. Wegner and J. I. Teuhola. The external heapsort. *IEEE Transactions on Software Engineering*, 15(7):9–925, July 1989.

Efficient Bulk Operations on Dynamic R-trees (Extended Abstract)

Lars Arge[1]*, Klaus H. Hinrichs[2], Jan Vahrenhold[2]**, and Jeffrey S. Vitter[1]***

[1] Center for Geometric Computing, Department of Computer Science
Duke University, Durham, NC 27708, USA
large@cs.duke.edu, jsv@cs.duke.edu
[2] Westfälische Wilhelms-Universität, Institut für Informatik
48149 Münster, Germany
khh@math.uni-muenster.de, jan@math.uni-muenster.de

Abstract. We present a simple lazy buffering technique for performing bulk operations on multidimensional spatial indexes (data structures), and show that it is efficient in theory as well as in practice. We present the technique in terms of the so-called R-tree and its variants, as they have emerged as practically efficient indexing methods for spatial data.

1 Introduction

In recent years there has been an upsurge of interest in spatial databases in the commercial and research database communities. Spatial databases are systems designed to store, manage, and manipulate spatial data like points, polylines, polygons, and surfaces. Geographic information systems (GIS) are a popular incarnation. Spatial database applications often involve massive data sets, such as for example EOS satellite data [13]. Thus the need for efficient handling of massive spatial data sets has become a major issue, and a large number of disk based multidimensional index structures (data structures) have been proposed in the database literature (see e.g. [21] for a recent survey). Typically, multidimensional index structures support insertions, deletions, and updates, as well as a number of proximity queries like window or nearest-neighbor queries. Recent research in the database community has focused on supporting *bulk operations*, in which a large number of operations are performed on the index at the same time. The increased interest in bulk operations is a result of the ever-increasing size of the manipulated spatial data sets and the fact that performing a large number of single operations one at a time is simply too inefficient to be of practical use. The most common bulk operation is to create an index for a given data set from scratch—often called *bulk loading* [14].

* Supported in part by the U.S. Army Research Office through MURI grant DAAH04–96–1–0013 and by the National Science Foundation through ESS grant EIA–9870734.
** Part of this work was done while a visiting scholar at Duke University.
*** Supported in part by the U.S. Army Research Office through MURI grant DAAH04–96–1–0013 and by the National Science Foundation through grants CCR–9522047 and EIA–9870734.

In this paper we present a simple lazy buffering technique for performing bulk operations on multidimensional indexes, and show that it is efficient in theory as well as in practice. We present our results in terms of the R-tree and its variants [16, 15, 26, 9, 18], which have emerged as especially practically efficient indexing methods for spatial data.

1.1 Model of computation and previous results on I/O-efficient algorithms

Since objects stored in a spatial database can be rather complex, they are often approximated by simpler objects, and the index is built on these approximations. The most commonly used approximation is the *minimal bounding box*; the smallest d-dimensional rectangle that includes the whole object. The resulting indexing problem is to maintain a dynamically changing set of d-dimensional rectangles on disk such that, for example, all rectangles containing a query point can be efficiently found. For simplicity we restrict our attention in this paper to the two-dimensional case; the boxes are called *minimal bounding rectangles.*

For our theoretical considerations we use the standard two-level I/O model [1] and define the following parameters:

$$\begin{aligned} N &= \#\text{ of rectangles,}\\ M &= \#\text{ of rectangles fitting in internal memory,}\\ B &= \#\text{ of rectangles per disk block,} \end{aligned}$$

where $N \gg M$ and $1 \leq B \leq M/2$. An *input/output operation* (or simply *I/O*) consists of reading a block from disk into internal memory or writing a block from internal memory to disk. Computation can only be performed on rectangles in internal memory. Our measures of performance of an algorithm are the number of I/Os it performs, the amount of disk space it uses (in units of disk blocks), and the internal memory computation time.[1] More sophisticated measures of disk performance involve analysis of seek and rotational latencies and caching issues [25]; however, the simpler standard model has proven quite useful in identifying first-order effects [31].

I/O efficiency has always been a key issue in database design, but has only recently become a central area of investigation in the algorithms community. Aggarwal and Vitter [1] developed matching upper and lower I/O bounds for a variety of fundamental problems such as sorting and permuting. For example, they showed that sorting N items in external memory requires $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. Subsequently, I/O-efficient algorithms have been developed for several problem domains, including computational geometry, graph theory, and string processing. Refer to recent surveys for references [3, 4, 32]. The practical merits of the developed algorithms have been explored by a number of authors [11, 31, 6, 5].

[1] For simplicity we concentrate on the two first measures in this paper. It can be shown that the asymptotic internal memory computation time of our new R-tree algorithms is the same as for the traditional algorithms.

Much of this work uses the *Transparent Parallel I/O programming Environment* (TPIE) [29–31]. TPIE is a set of C++ functions and templated classes that allow for simple, efficient, and portable implementation of I/O algorithms.

1.2 Previous results on bulk operations on R-trees

The R-tree, originally proposed by Guttman [16], is a height-balanced multiway tree similar to a B-tree [12, 7]. The leaf nodes contain $\Theta(B)$ data rectangles each, while internal nodes contain $\Theta(B)$ entries of the form (Ptr, R), where Ptr is a pointer to a child node and R is the minimal bounding rectangle covering all rectangles in the subtree rooted in that child.

An R-tree occupies $O(N/B)$ disk blocks and has height $O(\log_B N)$; insertions can be performed in $O(\log_B N)$ I/Os. There is no unique R-tree for a given set of data rectangles, and minimal bounding rectangles stored within an R-tree node can overlap. In order to query an R-tree to find all rectangles containing a given point p, all internal nodes whose minimal bounding rectangle contains p have to be visited. Intuitively, we thus want the minimal bounding rectangles stored in a node to overlap as little as possible. An insertion of a new rectangle can increase the overlap, and several heuristics for choosing which leaf to insert a new rectangle into, as well as for splitting nodes during rebalancing, have been proposed [15, 26, 9, 18].

Bulk loading an R-tree with N rectangles using the naive method of repeated insertion takes $O(N \log_B N)$ I/Os, which has been recognized to be abysmally slow. Several bulk loading algorithms using $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os have been proposed [24, 17, 14, 20, 28, 10]. These algorithms are more than a factor of B faster than the repeated insertion algorithm. Most of the proposed algorithms [24, 17, 14, 20] work in the same basic way; the input rectangles are sorted according to some global one-dimensional criterion (such as x-coordinate [24], the Hilbert value of the center of the rectangle [17, 14], or using an order obtained from a rectangular tiling of the space [20]) and placed in the leaves in that order. The rest of the index is then built recursively in a bottom-up, level-by-level manner. The algorithm developed in [28] also builds the index recursively bottom-up, but it utilizes a lazy buffering strategy. (Since this buffer strategy is similar to the technique we will describe later, we will give a more detailed description of the algorithm of [28] in Section 2.3.) Finally, the method proposed in [10] works recursively in a top-down way by repeatedly trying to find a good partition of the data.

Even though a major motivation for designing bulk loading algorithms is the slowness of the repeated insertion algorithm, another and sometimes even more important motivation is the possibility of obtaining better space utilization and/or query performance.[2] Most of the bulk loading algorithms mentioned

[2] A simple approach for building an R-tree using only $O(N/B)$ I/Os is to form the leaves by grouping the input rectangles B at a time, and then to build the tree in a bottom-up level-by-level manner. However, if the grouping of rectangles is done in an arbitrary manner, the resulting R-tree will likely have extremely bad query

above are capable of obtaining almost 95% space utilization (meaning that only $\frac{1}{0.95}\lceil \frac{N}{B} \rceil$ disk blocks are used), while empirical results show that average utilization of the repeated insertion algorithm is around 70% [9]. However, empirical results also indicate that packing an R-tree too much can lead to poor query performance, especially when the data is not uniformly distributed [14]. Bulk loading algorithms typically produce R-trees with better query performance than the repeated insertion algorithm. But no one algorithm is best for all cases (all data distributions) [20, 10]: On mildly skewed low-dimensional data the algorithm in [20] outperforms the algorithm in [17], while the opposite is the case on highly skewed low-dimensional data [20]. Both algorithms perform poorly on higher-dimensional data [10], where the algorithm developed in [10] achieves the best query performance.

Despite the common perception (as reported in [10], for example) that the algorithm developed in [28] (which utilizes a buffer technique similar to the one we use in this paper) produces an R-tree index identical to the index obtained using repeated insertion, this is definitely not the case. In [28] empirical results are only reported on the performance of the construction of multiversion B-trees [8]. The order of elements in the leaves of a multiversion B-tree is unique and thus the buffer algorithm will always construct the same order as the repeated insertion algorithm. This equivalence does not hold for R-tree construction and thus it remains an open problem as to what quality the index is produced using the buffer method.

All algorithms mentioned above are inherently "static" in the sense that they can only be used to bulk load an index with a given static data set. None of them efficiently supports bulk updates. Only a few attempts have been made on designing bulk update algorithms [23, 19], and efficient bulk updating is mentioned in [10] as an important open problem. The most successful attempt seems to be an algorithm by Kamel et al. [19]. In this algorithm the rectangles to be inserted are first sorted according to their spatial proximity (Hilbert value of the center), and then packed into blocks of B rectangles. These blocks are then inserted one at a time using standard insertion algorithms. Intuitively, the algorithm should give an insertion speedup of B (as blocks of B rectangles are inserted together), but it is likely to increase overlap and thus produces a worse index in terms of query performance. Empirical results presented in [19] support this intuition.

1.3 Our results

In this paper we present a simple lazy buffering technique for performing bulk operations on multidimensional indexes. As mentioned, we describe the technique in terms of the R-tree family.

In the first part of the paper (Section 2) we present our technique and analyze its theoretical performance. Unlike previous methods our algorithm does not

performance, since the minimal bounding rectangles in the nodes of the index will have significant overlaps. Such R-tree algorithms are therefore of no interest. At a minimum, constructing good R-trees is at least as hard as sorting, and thus we refer to algorithms that use $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os as *optimal.*

need to know in advance all the operations to be performed, something which is important in the many applications where updates and queries arrive continuously. Furthermore, our method is general enough to handle a wide variety of bulk operations:

– Our algorithm can *bulk insert* N' rectangles into an R-tree containing N rectangles using $O(\frac{N'}{B}\log_{M/B}\frac{N+N'}{B}+\frac{N}{B})$ I/Os in the worst case.
– Using the bulk insertion algorithm an R-tree can be *bulk loaded* in the optimal number of I/O operations $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.
– Given N' queries that require $O(Q\log_B\frac{N}{B})$ I/Os (for some Q) using the normal (one by one) query algorithm, our algorithm answers the queries in $O(\frac{Q}{B}\log_{M/B}\frac{N}{B}+\frac{N}{B})$ I/Os.
– A set of N' rectangles can be deleted from an R-tree containing N rectangles using $O(\frac{N'}{B}\log_{M/B}\frac{N}{B}+\frac{N}{B}+Q(N'))$ I/Os, where $Q(N')$ is the number of I/Os needed to locate the leaves containing the N' rectangles to be deleted.

In most cases our algorithms represents an improvement of more than a factor of B over known methods. For most indexes our technique can even handle a batch of intermixed inserts, deletes, and queries. As discussed in [23], being able to do so is extremely important in many environments where queries have to be answered while the index is being updated.

In the second part of the paper (Section 3) we present the results of a detailed set of experiments on real-life data. The data sets we use are the standard benchmark data set for spatial indexes, namely, the TIGER/Line data [27]. Our experiments were designed to test our theoretical predictions and to compare the performance of our algorithms with previously known bulk update algorithms:

– To investigate the general effects of our technique we used it in conjunction with the standard R-tree heuristics to bulk load an R-tree. We compared its performance with that of the repeated insertion algorithm. Our experiments show that we obtain a huge speedup in construction time and at the same time the query performance remains good. Even though bulk loading using our technique does not yield the same index as would be obtained using repeated insertion, the quality of the index remains about the same.
– As discussed, special-purpose bulk loading algorithms often produce significantly better indexes than the one obtained using repeated insertion, especially in terms of space utilization. We are able to capitalize on a certain laziness in our algorithm, via a simple modification, and achieve dramatically improved space utilization. The modification uses a heuristic along the lines of [17, 14]. Our bulk loading experiments with the modified algorithm show that we can obtain around 90% space utilization while maintaining or even improving the good query performance. As an added benefit the modification also improve construction time by up to 30%.
– Finally, in order to investigate the practical efficiency of our technique when performing more general bulk operations, we performed experiments with bulk insertion of a large set of rectangles in an already existing large R-tree. We compared the performance of our algorithm with that of the previously

best known algorithm [19]. Our algorithm performs better than the algorithm in [19], in terms of both the number of I/Os used to do the insertions and the query performance of the resulting R-tree index.

One especially nice feature of our algorithms is that from a high level point of view, the set of bulk operations are performed precisely as in the standard on-line algorithms. For example, our bulk insertion algorithm is conceptually identical to the repeated insertion algorithm (except of course that the insertions are done lazily). From an implementation point of view, our algorithms admit a nice modular design because they access the underlying index (in our case, the R-tree) only via standard routing and balancing routines. Having implemented our algorithms we can thus combine them with most existing indexes very easily.

2 Performing Bulk Operations on R-trees using Buffers

In this section we present our technique for performing bulk operations on R-trees and analyze it theoretically. In Section 2.1 we review the standard R-tree insertion and query algorithms and present the general idea in our buffer technique. In Section 2.2 we discuss the details in how a R-tree index can be bulk loaded, and in Section 2.3 how a bulk of insertions, deletions, or queries can be performed on an existing index using the technique. Final remarks (including a discussion of how our method compares with the previously proposed buffer technique [28]) are given in Section 2.4.

2.1 R-tree basics and sketch of our technique

As mentioned, the R-tree is a height-balanced tree similar to a B-tree; all leaf nodes are on the same level of the tree, and a leaf contains $\Theta(B)$ rectangles. Each internal node v (except maybe for the root) has $\Theta(B)$ children. For each of its children, v contains a rectangle that covers all the rectangles in the child. We assume that each leaf and each internal node fit in one disk block. An R-tree has height $O(\log_B \frac{N}{B})$ and occupies $O(\frac{N}{B})$ blocks. Guttman [16] introduced the R-tree, and several researchers have subsequently proposed different update heuristics designed to minimize the overlap of rectangles stored in a node [15, 26, 9, 18]. All variants of R-tree insertion algorithms (heuristics) [16, 15, 26, 9, 18] work in the same basic way (similar to B-tree algorithms) and utilize two basic functions:

- *Route*(r, v), which given an R-tree node v and a rectangle r to be inserted, returns the best (according to some heuristic) subtree v_s to insert r into. If necessary, the function also updates (extends) the rectangle stored in v that corresponds to v_s.
- *Split*(v), which given a node v, splits v into two new nodes v' and v''. The function also updates the entry for v in *father*(v) to correspond to v', as well as insert a new entry corresponding to v''. (If v is the root a new root with two children is created.)

Queries are handled in the same way in all R-tree variants using the following basic function:

– *Search*(r, v), which given a rectangle r and a node v, returns a set of subtrees V_s whose associated rectangles in v intersect r.

The abstract algorithms for inserting a new rectangle into an R-tree and for querying an R-tree with a rectangle are given in Figure 1. (Other queries can be handled with similar algorithms.) When performing a query, many subtrees may need to be visited, and thus it is not possible to give a better than linear I/O bound on the worst-case complexity of a query. An insertion can be performed in $O(\log_B \frac{N}{B})$ I/Os, since only nodes on a single root-to-leaf path are visited by the (routing as well as the rebalancing) algorithm.

Our technique for efficiently performing bulk operations on R-trees is a variant of the general *buffer tree technique* introduced by Arge [2, 3]. Here we modify the general technique in a novel way, since a straightforward application of the technique would result in an R-tree with an (impractically large) fan-out of $\frac{M}{B}$ [28]. The main idea is the following: We attach *buffers* to all R-tree nodes on every $\lfloor \log_B \frac{M}{4B} \rfloor$th level of the tree; more precisely, we define the leaves to be on level 0 and assign buffers of size $\frac{M}{2B}$ blocks to nodes on level $i \cdot \lfloor \log_B \frac{M}{4B} \rfloor$, for $i = 1, 2, \ldots$. We call a node with an associated buffer a *buffer node*. Operations on the structure are now done in a "lazy" manner. For example, let us assume that we are performing a batch of insertions. In order to insert a rectangle r, we do not immediately use *Route*(r, v) to search down the tree to find a leaf to insert r into. Instead, we wait until a block of rectangles to be inserted has been collected and then we store this block in the buffer of the root (which is stored on disk). When a buffer "runs full" (contains $\frac{M}{4}$ or more rectangles) we perform what we call a *buffer emptying process*: For each rectangle r in the buffer we repeatedly use *Route*(r, v) to route r down to the next buffer node v'. Then we insert r into the buffer of v', which in turn is emptied when it eventually runs full. When a rectangle reaches a leaf it is inserted there, and if necessary the index is restructured using *Split*. In order to avoid "cascading" effects we only route the first $\frac{M}{4}$ rectangles from the buffer in a buffer emptying process. Since

```
Insert(r, v)

1. WHILE v is not a leaf DO
     $v := Route(r, v)$
2. Add $r$ to the rectangles of $v$
3. WHILE too many rectangles in $v$
   DO
     $f := father(v)$; $Split(v)$; $v := f$
```

```
Query(r, v)

1. IF $v$ is not a leaf THEN
     $V_s := Search(r, v)$
     Recursively call $Query(r, v')$
     for all $v' \in V_s$
2. IF $v$ is a leaf THEN
     Report all rectangles in $v$
     that intersect $r$
```

Fig. 1. Abstract algorithms for inserting and querying in an R-tree rooted at node v.

all the buffers that these rectangles are routed to are non-full (i.e., they contain less than $\frac{M}{4}$ rectangles), no buffer overflow ($> \frac{M}{2}$ rectangles) can occur even if all rectangles are routed to the same buffer. The abstract buffer emptying process is sketched in Figure 2.

Because of the lazy insertion, some insertions do not cost any I/Os at all, while others may be very expensive (by causing many buffer emptying processes). However, the introduction of buffers allows us to take advantage of the big block and internal memory sizes and perform the whole batch of insertions with fewer I/Os than we would use if we performed the insertions in the normal way. The key to this improvement is that when we empty a buffer we route many rectangles through a relatively small set of R-tree nodes to the next level of buffer nodes. In fact, the set of nodes is small enough to fit in half of the internal memory; the number of different buffer nodes the rectangles can be routed to is bounded by $B^{\lfloor \log_B \frac{M}{4B} \rfloor} \leq \frac{M}{4B}$, since the fan-out of the nodes is bounded by B. Thus the maximal number of nodes that needs to be loaded is bounded by $2 \cdot \frac{M}{4B} = \frac{M}{2B}$. This means that we can route the rectangles from the buffer *in main memory*: Before we perform the buffer emptying process sketched in Figure 2, we simply load all the relevant nodes, as well as the first $\frac{M}{4}$ rectangles from the buffer, into main memory using $O(\frac{M}{B})$ I/Os. Then we can perform the routing without using any I/Os at all, before using $O(\frac{M}{B})$ I/Os to write the R-tree nodes back to disk, and $O(\frac{M}{B})$ I/Os to write the rectangles to the new buffers. (Since the number of buffers we write rectangles to is $< \frac{M}{4B}$, we only use $O(\frac{M}{B})$ I/Os to write non-full blocks.) In total we use $O(\frac{M}{B})$ I/Os to push $\frac{M}{4}$ rectangles $\lfloor \log_B \frac{M}{4B} \rfloor$ levels down, which amounts to $O(\frac{1}{B})$ I/Os per rectangle. To route all the way down the $O(\log_B \frac{N}{B})$ levels of the tree we thus use $O((1/(B \cdot \lfloor \log_B \frac{M}{4B} \rfloor)) \cdot \log_B \frac{N}{B}) = O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os per rectangle. By contrast, the normal insertion algorithm uses $O(\log_B \frac{N}{B})$ I/Os per rectangle.

The above is just a sketch of the main idea of the buffer technique. There are still many issues to consider. In the next two subsections we discuss some of the details of how the buffer technique can be used to bulk load an R-tree, as well as to perform more general bulk operations.

On other than level $\lfloor \log_B \frac{M}{4B} \rfloor$:

```
FOR the first $M/4$ rectangles $r$ in
$v$'s buffer DO

  1. $n := v$
  2. WHILE $n$ not buffer node DO
       $n := Route(r, n)$
  3. Insert $r$ in the buffer of $n$
```

On level $\lfloor \log_B \frac{M}{4B} \rfloor$:

```
FOR all rect. $r$ in $v$'s buffer DO

  1. $n := v$
  2. WHILE $n$ is not a leaf DO
       $n := Route(r, n)$
  3. Add $r$ to the rectangle in $n$
  4. WHILE too many rect. in $n$ DO
       $f := father(n)$; $Split(n)$, $n := f$
```

Fig. 2. Sketch of main idea in buffer emptying process on node v.

2.2 Bulk loading an R-tree

Our bulk loading algorithm is basically the standard repeated insertion algorithm, where we use buffers as described in the previous subsection. A number of issues still have to be resolved in order to make the algorithm I/O-efficient. For example, since a buffer emptying process can trigger other such processes, we use an external memory stack to hold all buffer nodes with full buffers (i.e., buffers containing more than $\frac{M}{4}$ rectangles). We push a reference to a node on this stack as soon as its buffer runs full, and after performing a buffer-emptying on the root we repeatedly pop a reference from the stack and empty the relevant buffer. Special care needs to be taken when the root is not a buffer node, that is, when it is not on level $j \cdot \lfloor \log_B \frac{M}{4B} \rfloor$, for some j. In such a situation we simply store the top portion of the tree without buffers in internal memory instead of on the disk. Note that since an R-tree (like a B-tree) only grows (shrinks) at the top, a node stays at the level of the tree it is on when it is created. This means that if a node is a buffer node when it is created it stays a buffer node throughout its lifetime.

We also need to fill in the details of how restructuring is performed, that is, how precisely the buffer of a node v on level $\lfloor \log_B \frac{M}{4B} \rfloor$ is emptied. The index is restructured using the *Split* function, and we need to be a little careful when splitting buffer nodes. The detailed algorithm for emptying the buffer of a node on level $\lfloor \log_B \frac{M}{4B} \rfloor$ is given in Figure 3: We first load the R-tree rooted at v (including the leaves containing the data rectangles) into internal memory. Then all rectangles in the buffer are loaded (blockwise) and each of them is routed (using *route*) to a leaf and inserted. If an insertion causes a leaf l to overflow, *Split* is used to split l as well as all the relevant nodes on the path from l to v (these nodes are all in internal memory). If the split propagates all the way up to v, we need to propagate it further up the part of the tree that is not in internal memory. In order to do so, we stop the buffer emptying process; we first write the trees rooted in the two (buffer) nodes v' and v'', produced by splitting v, back to disk. In order to distribute the remaining rectangles in the buffer of v we then load all of them into internal memory. For each rectangle r, we use *Route* to decide which buffer to insert r into, and finally we write each rectangle back to the relevant buffer on disk. It is easy to realize that we use $O(\frac{M}{B})$ I/Os on the above process, regardless of whether we stop the emptying process or not. Finally, the restructuring is propagated recursively up the index using *Split*, with the minor modification of the normal R-tree restructuring algorithm that rectangles in the buffer of a buffer node being split are distributed using *Route* as above.

Lemma 1. *A set of N rectangles can be inserted into an initially empty buffered R-tree using $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O operations.*

Proof. By the argument in Section 2.1 we use $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os per rectangle, that is, $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os in total, *not* counting I/Os used on emptying buffers on level $\lfloor \log_B \frac{M}{4B} \rfloor$, which in turn may result in splitting of R-tree nodes (restructuring). When emptying a buffer on level $\lfloor \log_B \frac{M}{4B} \rfloor$ we either push $\frac{M}{2}$

```
• Load R-tree (including leaves) rooted in $v$ into internal memory.
• WHILE (buffer of $v$ contains rectangle $r$) AND ($v$ is not split) DO
    1. $l := v$
    2. WHILE $l$ is not leaf DO
         $l := Route(r, l)$
    3. Insert $r$ in $l$
    4. WHILE (too many rectangles in $l$) AND ($v$ is not split) DO
         $f := father(l)$; $Split(l)$; $l := f$
• IF $v$ did not split THEN write R-tree rooted in $v$ back to disk.
• IF $v$ split into $v'$ and $v''$ THEN
    1. Write R-trees rooted in $v'$ and $v''$ back to disk.
    2. Load remaining rectangles in buffer of $v$ into internal memory.
    3. Use $Route$ to compute which of the two buffers to insert each
       rectangle in.
    4. Write rectangles back to relevant buffers.
    5. $v := father(v)$
    6. WHILE too many rectangles in $v$ DO
         $f := father(v)$; $Split(v)$
         IF $v$ is a buffer node THEN
           (a) Load rectangles in buffer of $v$ into internal memory.
           (b) Use $Route$ to compute which buffer to insert each
               rectangle in.
           (c) Write rectangles back to buffers.
         $v := f$
```

Fig. 3. Buffer emptying process on buffer nodes v on level $\lfloor \log_B \frac{M}{4B} \rfloor$.

rectangles down to the leaves and the argument used in Section 2.1 applies, or we suspend the emptying process in order to rebalance the tree. In the latter case we may spend a constant number of I/Os to split R-tree nodes on each of the $O(\log_B \frac{N}{B})$ levels and $O(\frac{M}{B})$ I/Os to distribute rectangles on each of the $O((\log_B \frac{N}{B})/\log_B \frac{M}{4B}) = O(\log_{M/B} \frac{N}{B})$ levels with buffers, including the $O(\frac{M}{B})$ I/Os we used on the suspended buffer emptying. However, we only spend these I/Os when new nodes are created. During the insertion of all N rectangles, a total of $O(\frac{N}{B}/B)$ R-tree nodes and a total of $O(\frac{N}{B}/\frac{M}{B}) = O(\frac{N}{M})$ buffer nodes are created. Thus the overall restructuring cost adds up to at most $O(\frac{N}{B})$ I/Os.

□

The only remaining issue is that after all the insertions have been performed it is very likely that we still have many nonempty buffers that need to be emptied in order to obtain the final R-tree. To do so we simply perform a buffer-emptying process on all buffer nodes in a breadth first manner, starting with the root.

Lemma 2. *All buffers in a buffered R-tree on N rectangles can be emptied in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O operations.*

Proof. The cost of emptying full buffers can be accounted for by the same argument as the one used in the proof of Lemma 1, and thus $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O operations are needed. The number of I/Os used on emptying non-full buffers is bounded by $O(\frac{M}{B})$ times the number of buffer nodes. As there are $O(\frac{N}{B}/\frac{M}{B})$ buffer nodes in a tree on N rectangles the bound follows.

□

The above two lemmas immediately imply the following.

Theorem 1. *An R-tree can be bulk loaded with N rectangles in* $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ *I/O operations.*

2.3 Performing bulk insertions, queries, and deletes

After having discussed our bulk loading algorithm it is easy to describe how bulk inserts can be performed efficiently on an already existing R-tree: We simply attach buffers to the tree, insert the rectangles lazily one by one, and perform a final emptying of all buffers. Using the same arguments as in the proofs of Theorem 1, we obtain the following.

Theorem 2. *A set of* N' *rectangles can be bulk inserted into an existing R-tree containing* N *rectangles in* $O(\frac{N'}{B}\log_{M/B}\frac{N+N'}{B}+\frac{N}{B})$ *I/O operations.*

In order to answer a large set of queries on an existing R-tree (such a situation often arises when performing a so-called spatial join [22]), we simply attach buffers to the R-tree and perform the queries in a lazy manner in the same way as when we perform insertions: To perform one query we insert the query rectangle in the buffer of the root. When a buffer is emptied and a query needs to be recursively performed in several subtrees, we simply route a copy of the query to each of the relevant buffers. When a query rectangle reaches a leaf the relevant data rectangles are reported.

Theorem 3. *A set of* N' *queries can be performed on an existing R-tree containing* N *rectangles in* $O(\frac{Q}{B}\log_{M/B}\frac{N}{B}+\frac{N}{B})$ *I/O operations, where* $Q\log_B N$ *is the number of nodes in the tree the normal R-tree query algorithm would visit.*

Conceptually, bulk deletions can be handled as insertions and queries and in the full version of this paper we prove the following.

Theorem 4. *A set of* N' *rectangles can be bulk deleted from an R-tree containing* N *rectangles using* $O(\frac{N'}{B}\log_{M/B}\frac{N}{B}+\frac{N}{B}+Q(N'))$ *I/O operations, where* $Q(N')$ *is the number of I/O operations needed to locate the rectangles in the tree using batched query operations.*

2.4 Further remarks

In the previous sections we have discussed how to perform insertions, deletions, and queries separately using buffers. Our method can also be modified such that a batch of *intermixed* updates and queries can be performed efficiently (even if they are not all present at the beginning of the algorithm, but arrive in an on-line manner). Being able to do so is extremely important in many on-line environments, where queries have to be answered while the index is being updated [23]. Buffers are attached and the operations are performed by inserting them block-by-block into the buffer of the root. Buffer emptying is basically performed as discussed in the previous sections. The buffer emptying process for nodes on level $\lfloor \log_B \frac{M}{4B} \rfloor$ have to be modified slightly and the interested reader is referred to [2] for details.

As mentioned in the introduction, a buffering method similar to our approach has previously been presented by van den Bercken, Seeger, and Widmayer [28]. However, while our approach supports all kinds of bulk operations, the approach in [28] only supports bulk loading. To bulk load an R-tree, the algorithm in [28] first constructs an R-tree with fanout $\frac{M}{B}$ by attaching buffers to *all* nodes and performing insertions in a way similar to the one used in our algorithm. The leaves in this R-tree are then used as leaves in the R-tree with fanout B that will eventually be constructed. The rest of this tree is produced in a bottom-up manner by successively applying the buffer algorithm to the set of rectangles obtained when replacing the rectangles in the nodes on the just constructed level with their minimal bounding rectangle. The number of I/Os used on performing the bulk loading is dominated by the construction of the leaf level, which asymptotically is the same as the algorithm we develop. In practice, however, the additional passes over the data, even though they involve data of geometrically decreasing size, often represent a significant percentage of the total time. The bottom-up construction is inherently off-line, since all data needs to be known by the start of the algorithm, and the algorithm is therefore unsuitable for bulk operations other than bulk loading. In summary, both the method proposed in [28] and our method utilize a lazy buffering approach, but our technique seems to be more efficient than the one in [28], and at the same time it allows for much more general bulk operations.

3 Empirical Results

In this section we discuss our implementation of the algorithms presented in the last section and give empirical evidence for their efficiency when compared to existing methods. In Section 3.1 we describe our implementation and the experimental setup. Section 3.2 is dedicated to an empirical analysis of the effects of buffering, and in Section 3.3 we discuss how to improve our algorithms using heuristics similar to the ones used in [17, 14]. Finally, in Section 3.4 we compare the I/O cost and query performance of the different bulk insertion algorithms.

3.1 Our implementation

In order to evaluate the practical significance of our buffer algorithm we implemented the original repeated insertion algorithm for R-tree construction [16], the bulk insertion algorithm developed in [19], and our proposed buffered bulk insertion algorithm. We also implemented the normal query algorithm [16]. Our implementations were done in C++ using TPIE [29, 30].[3] TPIE supports both a *stream-oriented* and a *block-oriented* style of accessing secondary storage. A TPIE stream represents a homogeneous list of objects of an arbitrary type, and the system provides I/O-efficient algorithms for scanning, merging, distributing, and sorting streams. The block-oriented part of TPIE supports random accesses to specific blocks. TPIE supports several methods for actually performing the I/Os. All these methods work on standard UNIX files. For example, one method uses the standard I/O system calls `fread` and `fwrite`, while another relies on memory mapping (the `mmap` and `munmap` calls). In our experiments we used the method based on memory mapping, and when we refer to the number of I/Os performed by an algorithm, we refer to TPIE's count of how many `mmap` operations were performed (both reads and writes involve mapping a block). The actual physical number of I/Os performed is very likely to be less than this count, as the operating system can choose to keep a block in internal memory even after it is unmapped.

A conceptual benefit of our algorithms is that from an abstract point of view they are similar to the normal algorithms where the operations are performed one by one. Our algorithms admit a nice modular design because they only access the underlying R-tree through the standard routing and restructuring procedures. We used the block-oriented part of TPIE to implement a standard R-tree [16] that served as a base for realizing the different update approaches. Using the stream-oriented part of TPIE we stored all the buffers of an index in one separate stream. As a consequence of this modular design we are immediately able to attach buffers to any existing index, regardless of how it has been created. After all of the bulk operations have been performed (and all buffers have been emptied) we can even decide to detach the buffers again without affecting the updated index. Therefore our buffer algorithm can be regarded as a generic "black box" that takes an arbitrary index and returns an updated version while only accessing the public interface of the index.

Our experiments were done on a machine with a block size of 4 Kbytes (Sun SparcStation20 running Solaris 2.5) which allowed for an R-tree fanout of 100. However, following recommendations of previous empirical studies [16, 9], we only used a maximal fanout of 50. Similarly, we used a minimal fanout of 50/6 which has previously been found to give the best query performance. For simplicity, we added buffers in our implementation to all nodes instead of only to nodes on every $\lfloor \log_B \frac{M}{4B} \rfloor$th level. Theoretically, by using a buffer size of only B blocks, we obtain a bulk loading I/O bound of $O(\frac{N}{B} \log_B \frac{N}{B})$. In practice, this is not significantly different from the theoretical $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ bound obtained in Section 2.

As test data we used the standard benchmark data used in spatial databases, namely, rectangles obtained from the TIGER/Line data set [27] for the states of Rhode Island, Connecticut, New Jersey and New York. In all our bulk loading and updating experiments we used rectangles obtained from the road data of

[3] TPIE is available at http://www.cs.duke.edu/TPIE. The implementations described in this paper will be incorporated in a future release of TPIE.

State	Category	Size	Objects	Category	Queries	Results
Rhode Island (RI)	Roads	4.3 MB	68,278	Hydrography	701	1,887
Connecticut (CT)	Roads	12.0 MB	188,643	Hydrography	2,877	8,603
New Jersey (NJ)	Roads	26.5 MB	414,443	Hydrography	5,085	12,597
New York (NY)	Roads	55.7 MB	870,413	Hydrography	15,568	42,489

Table 1. Characteristics of test data.

these states. Our query experiments consisted of overlap queries with rectangles obtained from hydrographic data for the same states. The queries reflect typical queries in a spatial join operation. In order to work with a reasonably small but still characteristic set of queries, we used every tenth object from the hydrographic data sets. In the construction and updating experiments, we used the entire road data sets. The sizes of the data sets are given in Table 1. The third column shows the size of the road data set *after* the bounding rectangles have been computed, that is, it is the actual size of the data we worked with.

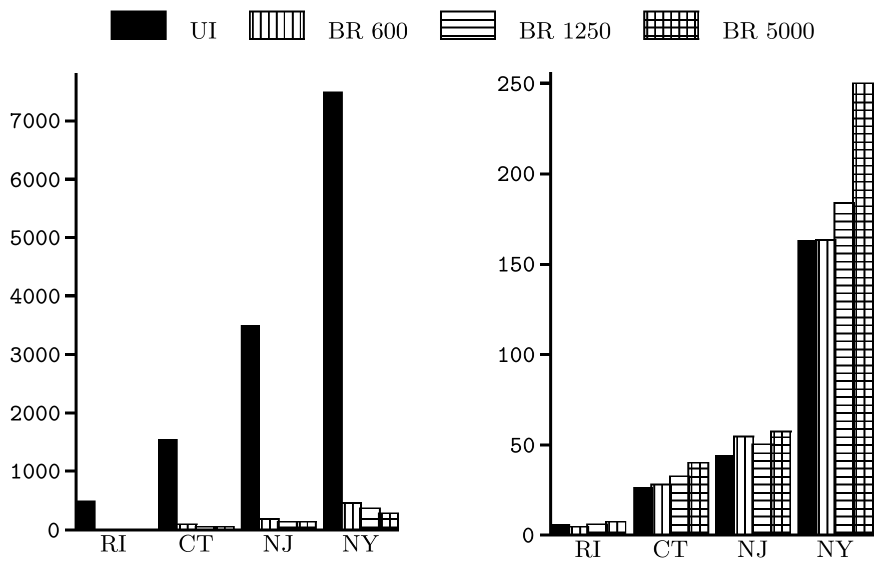
Finally, it should be mentioned that for simplicity we in the following only present the results of query experiments performed *without* buffers. If buffers are used we observe a speed-up similar to the speed-ups observed in our bulk loading experiments that we present in the next section.

3.2 Effects of adding buffers to multidimensional indexes

In order to examine the general effects of adding buffers to a multidimensional index, we first performed a set of bulk loading experiments where we compared the performance of our algorithm (in the following referred to as BR for $\underline{\text{b}}$uffer $\underline{\text{R}}$-tree) with that of the standard repeated insertion algorithm. (Since the repeated insertion algorithm takes $\underline{\text{u}}$nsorted data and $\underline{\text{i}}$nserts it, we will use UI when referring to it.) We performed experiments with buffers of size $\lfloor B/4 \rfloor$, $\lfloor B/2 \rfloor$ and $\lfloor 2B \rfloor$ blocks. With our choice of $B = 50$ this corresponds to buffers capable of storing 600, 1250 and 5000 rectangles, respectively.

Figure 4 (a) shows that our algorithm (BR) dramatically outperforms the repeated insertion algorithm (UI) as expected. Depending on the buffer size we reduce the number of I/O operations by a factor of 16–24. Our experiments on query performance given in Figure 4 (b) show that, in contrast to popular belief, the introduction of buffers does affect the structure of the resulting index. However, they also show that by carefully choosing the buffer size we are able to produce an index of the same quality as the index produced by UI.

Our conclusion from this first set of experiments is that buffers can indeed be used to speed up spatial index algorithms without sacrificing query performance. Since our main objective was to design and test algorithms capable of performing more general bulk operations than just bulk loading, we did not compare our bulk loading algorithm with other such algorithms. The purpose of this first set of experiments was just to examine the effects of buffering on the behavior of index structures where the query performance is sensitive to the order in which data elements are inserted. Specialized bottom-up R-tree bulk loading algorithms are



(a) Costs for building the R-tree (b) Costs for querying the R-tree

Fig. 4. Effects of buffering on build and query performance (I/O in thousands)

likely to outperform our algorithm because of the smaller number of random I/Os (as opposed to sequential I/Os) performed in such algorithms.

3.3 Improving the space utilization

As discussed in the introduction, special-purpose bulk loading algorithms often produce significantly better indexes than the ones obtained using repeated insertion, especially in terms of space utilization. There exist several heuristics for improving the space utilization of two-dimensional R-trees from approximately 70% [9] to almost 95% [17, 14, 20, 19, 10]. One key question is therefore whether we can take advantage of some of these heuristics in our buffer algorithm in order to improve space utilization, and without sacrificing the conceptual advantage of not having to know all updates by the start of the algorithm.

It turns out that by modifying our buffer emptying algorithm for nodes on the level just above the leaves, we are indeed able to combine the advantages of our buffering method and the so-called Hilbert heuristic [17]: As in the algorithm discussed in Section 2, we start by loading all leaves into internal memory. Instead of repeatedly inserting the rectangles from the buffer using the standard algorithm, we then sort all the rectangles from the buffer and the leaves according to the Hilbert values of their center. The rectangles are then grouped into new leaves that replace the old ones. Following the recommendations in [14], we are careful not to fill the leaves completely. Instead we fill them up to 75% of capacity and include the next candidate rectangle only if it increases the area of the minimal bounding rectangle of the rectangles in the leaf by no more than

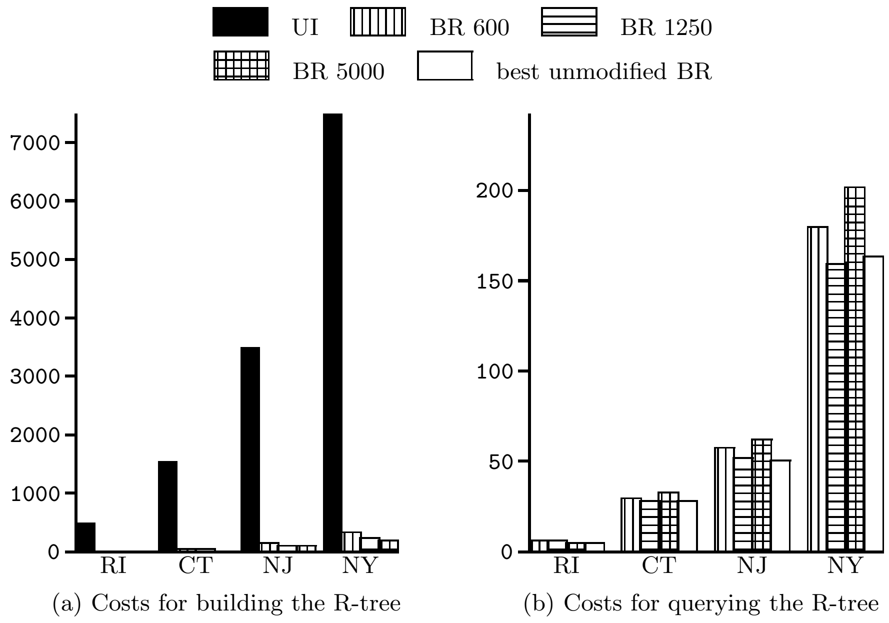


Fig. 5. Experiments with modified leaf buffer emptying algorithm (I/O in thousands)

20%. Since the space utilization of the tree is mainly determined by how much the leaves are filled, this modification improves our space utilization significantly.

In order to evaluate the modified algorithm, we repeated the bulk loading experiments from Section 3.2. The results of these experiments were very encouraging: With all buffer sizes we improved space utilization to approximately 90%, and at the same time we improve the construction time by around 30% compared with our original algorithm. We also noticeably reduced the query time of the algorithm using the largest buffer size, and for each data set we were able to produce at least one index matching the query performance obtained by the original buffer algorithm. In most cases we were even able to produce a better index than the one produced using repeated insertions. The results are summarized in Figure 5. The detailed I/O costs of the experiments, as well as of the experiments from Section 3.2, are presented in Table 2.

3.4 Bulk updating existing R-trees

Finally, in order to investigate the practical efficiency of our technique when performing more general bulk operations, we performed experiments with bulk insertion of a large set of rectangles into an already existing R-tree. We compared the performance of our buffer algorithm (with a buffer size of 5000 rectangles) with the naive repeated insertion algorithm (UI), as well as with the bulk update algorithm of Kamel et al. [19]. As mentioned, this algorithm sorts the rectangles to be bulk inserted according to the Hilbert value of their centers and groups them into blocks. These blocks are then inserted using the repeated insertion

Data Set	Buffer Size	Building Standard	Building Modified	Querying Standard	Querying Modified	Packing Standard	Packing Modified
RI	0	495,909	495,909	5,846	5,846	56%	56%
	600	32,360	23,801	5,546	5,858	60%	88%
	1,250	26,634	16,140	6,429	6,632	60%	89%
	5,000	21,602	11,930	8,152	5,322	59%	90%
CT	0	1,489,278	1,489,278	27,699	27,699	56%	56%
	600	94,286	74,244	28,569	28,947	59%	88%
	1,250	76,201	50,983	32,586	27,591	59%	90%
	5,000	62,584	38,588	40,600	32,352	60%	91%
NJ	0	3,376,188	3,376,188	43,156	43,156	56%	56%
	600	211,467	167,401	53,874	56,844	59%	88%
	1,250	168,687	117,971	50,183	50,743	59%	90%
	5,000	142,064	92,578	57,571	61,485	59%	91%
NY	0	7,176,750	7,176,750	160,235	160,235	56%	56%
	600	448,645	345,038	160,232	175,424	59%	88%
	1,250	357,330	250,704	180,205	155,338	59%	89%
	5,000	299,928	203,165	243,704	196,972	59%	90%

Table 2. Summary of the I/O costs for all construction experiments.

algorithm. (Since this algorithm sorts the rectangles and groups them into nodes, we refer to it as SN.) To allow for a fair competition we used the I/O-efficient external sorting algorithm in TPIE to sort the rectangles. We allowed the sort to use 4 Mbytes of main memory. This should be compared to the maximal internal memory use of around 300 Kbytes of the buffer algorithm.

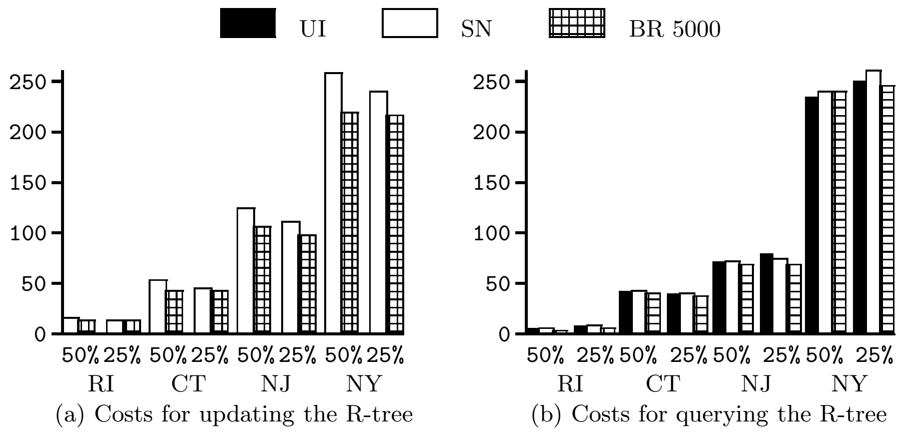


Fig. 6. Bulk updating results for UI, SN and BR (buffer 5000) (I/O in thousands)

For each state we constructed two base R-trees with 50% and 75% of the objects from the road data set using BR and inserted the remaining objects with the three algorithms. Experiments showed that the algorithms SN and BR outperform UI with respect to updating by the same order of magnitude,

Data Set	Update Method	Update with 50% of the data Building	Querying	Packing	Update with 25% of the data Building	Querying	Packing
RI	UI	259,263	6,670	64%	145,788	8,148	66%
	SN	15,865	7,262	92%	14,490	8,042	91%
	BR	13,484	5,485	90%	13,301	6,981	91%
CT	UI	805,749	40,910	66%	428,163	39,016	69%
	SN	51,086	40,593	92%	44,236	39,666	91%
	BR	42,774	37,798	90%	42,429	35,968	90%
NJ	UI	1,777,570	70,830	66%	943,992	66,715	71%
	SN	120,034	69,798	92%	106,712	71,383	91%
	BR	101,017	65,898	91%	95,823	66,030	91%
NY	UI	3,736,601	224,039	66%	1,988,343	238,666	71%
	SN	246,466	230,990	92%	229,923	249,908	91%
	BR	206,921	227,559	90%	210,056	233,361	90%

Table 3. Summary of the I/O costs of all update experiments.

and that our algorithm (BR) additionally improves over SN—see Figure 6 (a). As far as query performance is concerned, our experiments showed that SN results in worse indexes than the repeated insertion algorithm (UI). On the other hand, the indexes generated by our algorithm (BR) are up to 10% better than the indexes produced by SN, and they match or even improve the query performance obtained by indexes updated using UI—refer to Figure 6 (b). The major problem with SN with respect to query performance is that it does not take into consideration the objects already present in the R-tree; all new leaves are built without looking at the index to be updated.

The detailed I/O costs of all update experiments are given in Table 3. Our experiments show that our bulk update algorithm outperforms the previously known algorithms in terms of update time, while producing an index of at least the same quality. The overall conclusion of our experiments is that our buffer technique is not only of theoretically interest, but also a practically efficient method for performing bulk updates on dynamic R-trees.

4 Conclusions

In this paper we have presented a new buffer algorithm for performing bulk operations on dynamic R-trees that is efficient both in theory and in practice. Our algorithm matches the performance of the previously best known update algorithm and at the same time it improves the quality of the produced index. It allows for simultaneous updates and queries which is essential in many on-line environments. One key feature of our algorithm is that from a high level point of view the bulk operations are performed precisely as if buffers were not used. For example, our bulk insertion algorithm is conceptually identical to the repeated insertion algorithm. From an implementation point of view another key feature of our algorithm is that it admits a nice modular design because it only accesses the underlying index through the standard routing and restructuring routines. Having implemented our buffering algorithms we can thus combine them with the most efficient existing index implementation for the problem class at hand.

References

1. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.
3. L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, February/August 1996.
4. L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS.* Springer-Verlag, LNCS 1340, 1997.
5. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. 24th Intl. Conf. on Very Large Databases*, pages 570–581, 1998.
6. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 685–694, 1998.
7. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
8. B. Becker and S. Gschwind and T. Ohler and B. Seeger and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.
9. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.
10. S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Intl. Conf. on Extending Database Technology*, pages 216–230, 1998.
11. Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.
12. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
13. R. F. Cromp. An intellegent information fusion system for handling the archiving and querying of terabyte-sized spatial databases. In *S. R. Tate ed., Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community, CESDIS Technical Report Series, TR–93–99*, pages 75–84, 1993.
14. D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *Proc. 20th Intl. Conf. on Very Large Databases*, pages 558–569, 1994.
15. D. Greene. An implementation and performance analysis of spatial data access methods. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 606–615, 1989.
16. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1985.
17. I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. 2nd Int. Conf. on Information and Knowledge Management (CIKM)*, pages 490–499, 1993.
18. I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. 20th Intl. Conf. on Very Large Databases*, pages 500–509, 1994.

19. I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proc. 4th Intl. Symp. on Spatial Data Handling*, pages 3B.31–3B.42, 1996.
20. S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. IEEE International Conferance on Data Engineering*, pages 497–506, 1997.
21. J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, LNCS 1340, 1997.
22. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 259–270, 1996.
23. N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 89–111, 1997.
24. N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 17–31, 1985.
25. C. Ruemmler and J. Wilkes An introduction to disk drive modeling. *IEEE Computers*, 27(3):17–28, 1994.
26. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. 13th Int. Conf. on Very Large Databases*, pages 507–518, 1987.
27. TIGER/Line (tm). 1992 technical documentation. Technical report, U. S. Bureau of the Census, 1992.
28. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. 23rd Intl. Conf. on Very Large Databases*, pages 406–415, 1997.
29. D. E. Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symposium on Parallel Computation*, 1994.
30. D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1997. Available via WWW at `http://www.cs.duke.edu/TPIE/`.
31. D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, 1996.
32. J. S. Vitter. External Memory Algorithms. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, to appear. Earlier shorter versions appear as an invited tutorial in *Proc. 17th ACM Symp. on Principles of Database Systems*, pages 119–128, 1998, and as an invited paper in *Proc. 6th Annual European Symposium on Algorithms*, pages 1–25, 1998.

Author Index