2nd Workshop on Algorithm Engineering
WAE'98 – Proceedings

Kurt Mehlhorn, Ed.

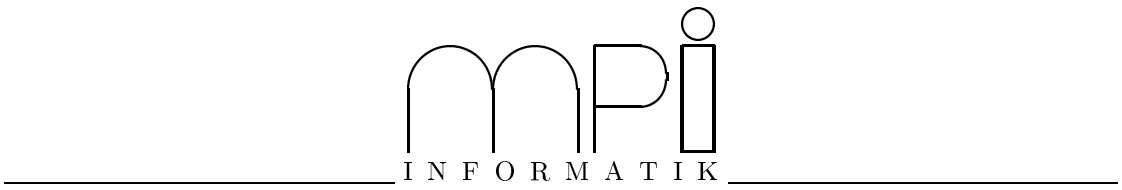
MPI–I–98–1-019 August 1998

Kurt Mehlhorn, Ed.

WAE'98

2nd Workshop on Algorithm Engineering

Max-Planck-Institut für Informatik
Saarbrücken, Germany, August 20–22, 1998
Proceedings



FOREWORD

The papers in this volume were presented at the *2nd Workshop on Algorithm Engineering* (WAE'98) held in Saarbrücken, Germany, on August 20–22, 1998. The Workshop was devoted to researchers and developers interested in the practical aspects of algorithms and their implementation issues. In particular, it brought together researchers, practitioners and developers in the field of algorithm engineering in order to foster cooperation and exchange of ideas. Relevant themes of the Workshop were the design, experimental testing and tuning of algorithms in order to bridge the gap between their theoretical design and practical applicability.

There were 32 submissions, all in electronic format. The Program Committee selected 18 papers in a deliberation conducted via an "electronic meeting" that ran from June 1 to June 14, 1998. The selection was based on perceived originality, quality, and relevance to the subject area of the workshop. Submissions were not refereed, and many of them represent preliminary reports of continuing research. A special issue of the *ACM Journal on Experimental Algorithmics* will be devoted to selected papers from WAE'98.

We would like to thank all those who submitted papers for consideration, the Program Committee members and their referees for their invaluable contribution, and the members of the Organizing Committee for all their time and effort. We are grateful to Pino Italiano, Salvatore Orlando, and Fabio Pittarelo for their generous help with the software of the submissions and program committee Web servers.

We gratefully acknowledge support from ALCOM-IT (a European Union ESPRIT LTR Project), and the Max-Planck-Institut für Informatik.

Saarbrücken, August 1998

Kurt Mehlhorn

Invited Lecturers

Thomas Lengauer	GMD, Bonn, Germany
Bernard Moret	University of New Mexico, Albuquerque, USA
Petra Mutzel	Max-Planck-Institut für Informatik, Saarbrücken, Germany

Program Committee

Jean Daniel Boissonat	INRIA, Sophia Antipolis, France
Andrew V. Goldberg	NEC Research Institute, Princeton, USA
David S. Johnson	AT & T Labs Research, Florham Park, USA
Kurt Mehlhorn, Chair	Max-Planck-Institut für Informatik, Saarbrücken, Germany
Friedhelm Meyer auf der Heide	University of Paderborn, Germany
Bernard Moret	University of New Mexico, Albuquerque, USA

Organizing Committee

Gerth Brodal	Max-Planck-Institut für Informatik, Saarbrücken, Germany
Ingrid Finkler-Paul	Max-Planck-Institut für Informatik, Saarbrücken, Germany
Christoph Storb	Max-Planck-Institut für Informatik, Saarbrücken, Germany
Roxane Wetzel	Max-Planck-Institut für Informatik, Saarbrücken, Germany
Christos D. Zaroliagis, Chair	Max-Planck-Institut für Informatik, Saarbrücken, Germany

CONTENTS

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 1–12

Why CAD Data Repair Requires Discrete Algorithmic Techniques

Karsten Weihe
Fakultät für Mathematik und Informatik, Universität Konstanz, Fach D188
78457 Konstanz, Germany
e-mail: karsten.weihe@uni-konstanz.de

and

Thomas Willhalm
Fakultät für Mathematik und Informatik, Universität Konstanz, Fach D188
78457 Konstanz, Germany
e-mail: thomas.willhalm@uni-konstanz.de

ABSTRACT

We consider a problem of reconstructing a discrete structure from unstructured numerical data. The problem arises in the computer-aided design of machines, motor vehicles, and other technical devices. A CAD model consists of a set of surface pieces in the three-dimensional space (the so-called *mesh elements*). The neighbourhoods of these mesh elements, the *topology* of the model, must be reconstructed. The reconstruction is non-trivial because of erroneous gaps between neighboured mesh elements.

However, a look at the real-world data from various applications strongly suggests that the pairs of neighboured mesh elements may be (nearly) correctly identified by some distance measure and some threshold. In fact, to our knowledge, this is the main strategy pursued in practice. In this paper, we make a first attempt to design systematic studies to support a claim of failure: we demonstrate empirically that human intuition is misleading here, and that this approach fails even for "innocent-looking" real-world data. In other words, it does not suffice to look at individual pairs of surface pieces separately; incorporating discrete relations between mesh elements is necessary.

1. Introduction

Computer-aided design plays an important role in today's engineering. In this paper, we deal with CAD data models such as the one shown in Figure 1. Such a model consists of mesh elements and approximates the surface of the workpiece. In general, the mesh elements are not parts of a plane, and their edges are not straight lines. To give a concrete example: in the data available to us, *trimmed parametric surface patches* were used (see [8]).

One of the tasks which is to be done automatically is the reconstruction of the so-called *topology* of the CAD data model, i.e. the information whether and where two mesh elements are to be regarded as immediately neighboured (Figure 7). Many wide-spread data formats for CAD models do not provide the neighbourhoods. The topology of a CAD model is important, since almost every further step of the CAD process relies on this information. Section 2 gives examples of such steps.

Pictures such as Figures 1 and 4 suggest that the edges of neighboured mesh elements fall together geometrically. This is generally not the case. There are normally gaps between the mesh elements – gaps that can be as large as in Figures 2 and 3. These gaps are sometimes even larger than mesh

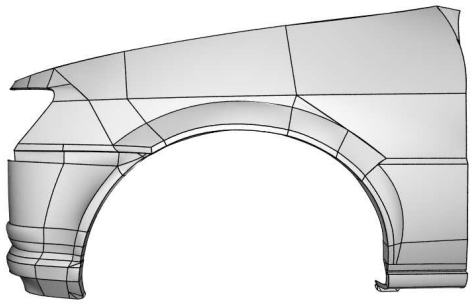


Figure 1: This mudguard (workpiece 8) is a typical example of our real world instances. It consists of surfaces patches (the so-called mesh elements). The black curves indicate the edges of the mesh elements.

elements in the same workpiece. They are the reason why the problem is non-trivial, since we have no knowledge whether an edge is neighboured to one, several or no other edges (Figure 8). Consider two edges that are situated next to each other. The gap between them may have been intended by the engineer who designed of the workpiece to separate them, but the two surface patches might also be regarded as parts of one closed surface. The automatic tools have to guess the intention of the engineer. The problem is thus inherently subjective and not mathematical in nature. In particular, the quality of the output is the point of interest. In fact, time and space consumption is negligible compared to certain other steps of the CAD process.

In this paper, we focus on a common approach, which is highly suggested by Figures 1 and 4: to define a distance measure of two edges and choose a threshold value[1]. An example of a distance measure is the maximal distance between two points on the respective edges, or the sum of the distances between the end points. Only pairs of edges whose distance is smaller than the threshold value are considered as neighboured. This may be viewed as estimating the *absolute error* of the placement of the edges.

To refine this approach, one can replace the absolute error with the *relative error*. Properties such as the lengths of the edges, or the area or perimeter of the mesh elements seem reasonable scalings for such a scaled distance measure. To our knowledge, these approaches with scaled or unscaled distance measures are the main strategy in today's industrial practice [7, 8]. Even Figures 2 and 3 suggest that this approach yields a very good approximation of the right neighbourhoods. Thus the problem does not seem to deserve further research.

The main contribution of this paper is to demonstrate empirically that in general this approach fails against all expectations – even if we assume that the algorithm automatically finds the best threshold value for each edge (of course, this assumption is far too optimistic). We have examined a variety of unscaled distance measures and possible extensions to scaled distance measures. Naturally our investigations only cover a subset of all possible distance measures. However, we believe that the most logical ones were treated and that they are good representatives for all useful distance measures, because our results are not too sensitive to the choice of the distance measure. This will be discussed further in Section 4.

Our results imply that an algorithm that is based solely on some measure for the (relative or absolute) error cannot determine a satisfying approximation of the topology. In other words,

[1] We produced many more pictures than are shown in this paper. All of them give the same impression.

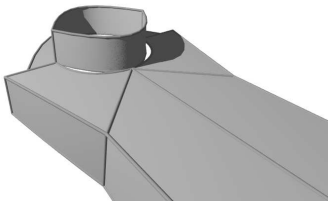


Figure 2: An example of a sloppy design (workpiece 15).

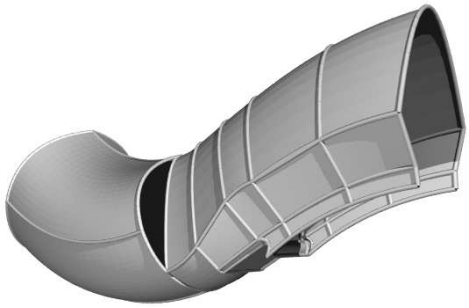


Figure 3: An extreme gap from the interior of Figure 4. The black, semicircular hole does not make any sense from an engineering point of view and might be a serious case of sloppy design.

unstructured information such as the position, size, and shape of mesh elements does not suffice to decide whether or not two mesh elements are neighboured. One seems forced to look at the problem from a more abstract point of view and use *discrete techniques*. By that we mean that rules are applied in which geometrical and numerical details are abstracted away: some kind of *logical inference rules*, which incorporate more than two mesh elements in their reasoning. A simple example is the rule that the relation of neighbourhoods has to be transitive: if patches A and B as well as patches B and C are neighboured, then A and C must also be neighboured. Such a rule might add missing neighbourhoods along branchings like in Figure 8. Furthermore, an algorithm may use meta-information about the structure, the topology, of the workpiece. The special case where the CAD model is the surface of a (connected) solid object or a plane deformed by a press is an example of strong meta-information: the graph induced by the mesh elements and neighbourhoods is planar.

In [10] we presented an algorithm for the reconstruction of the topology, which relies on structural techniques and which produced acceptable results for our benchmarks. As a by-product, the results presented here in retrospect justify the choice of a discrete approach in [10].

In summary, we were faced with a practical problem that does not seem to be mathematical in nature: the dirtiness of the problem cannot be abstracted away, since it is at the heart of the problem. This paper is a first attempt to design systematic studies on real-world data to support negative claims. It demonstrates that the dirtiness is not limited to artificial and pathological examples. Errors that caused the common approach to fail were found all over the test set. Cleaning the problem up to a (mathematically) nice one would make a completely different problem of it – actually, the real problem would vanish.

2. Background

Need for topology. The topology of a CAD model is essential for most automated tasks. For example, applying finite element methods only makes sense when forces or heat flow are transmitted correctly from element to element. The topology is also necessary to refine or coarsen the mesh of a workpiece. Figure 4 shows an example of a CAD data model that was constructed manually, and a refined mesh of the same workpiece that was automatically produced by the algorithm in [5]. Furthermore, the topology is needed for smoothing surfaces or replacing several mesh elements by others of a simpler type. See [7] for a list of further tasks.

Reasons for gaps. As already stated in Section 3, the unintented gaps are the main reason why the reconstruction of the topology is difficult. The gaps in Figures 2 and 3 are obviously not due to mere numerical rounding errors. We have found several explanations why these and other gaps exist.

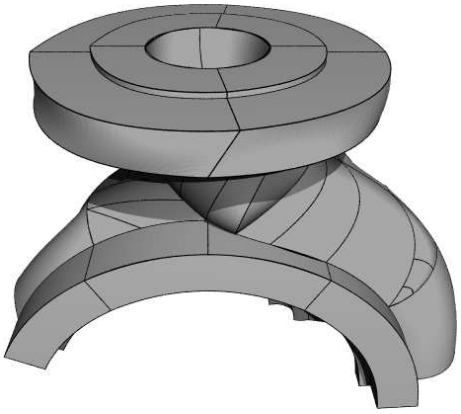


Figure 4: A CAD model of a pump (workpiece 13).

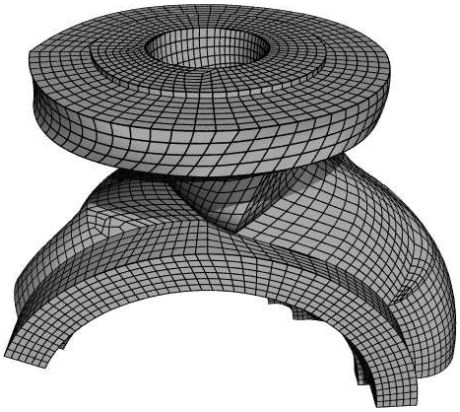


Figure 5: The same workpiece refined by the algorithm in [5].

- While creating a workpiece, the designer can position the edges freely in the three dimensional space. In particular, two edges can be neighboured on parts of their full length. The low order of the polygons that model the edge makes gaps unavoidable then.
- Conversions between non-isomorphic data formats sometimes make the approximation of a free form surface by another type necessary.
- Automatic tools that create CAD models from input devices like computer tomographs or 3D scanners do not have any knowledge about the topology of the workpiece. Errors and an inaccurate precision of measurement may then lead to gaps.

See [3] for a detailed discussion of these issues.

Existing approaches. Some restrictions on the input format as well as much cleaner CAD models sometimes allow the use of a distance measure in combination with a threshold value. Sheng and Meier examined in [8] the case when the surface is the boundary of a solid object. They also faced the problem of gaps that are as large as mesh elements. Their solution was to interact with the user in cases of doubt.

Knowledge about the discrete structure is used in very few existing approaches. In [2] Bøhn and Wozny restricted the problem to workpieces where the topology is known except for a few holes (which may be viewed as another kind of discrete pre-knowledge). This is of course less difficult than a total reconstruction of the topology. Apart from this, their approach is heavily based on the assumption that the mesh elements form the surface of a solid object. It incorporates the fact that the closed surface is orientable and uses the knowledge on which side of the surface the solid object is situated. However, their algorithm can only find a special kind of gap. The same problem was also treated by Barequet and Sharir in [1], but heir approach is not limited to a special kind of gap. We presented an algorithm in [10] for the problem in the generality we describe in Section 3. In contrast, in our problem variant, the topology – or any other kind of meta-information – is not even partially available.

3. Problem Description and Discussion

Input. To give a concrete, illustrative example, we will describe the format of the data that is available to us. A CAD model consists of a set of parametric surface patches. In our case, a patch

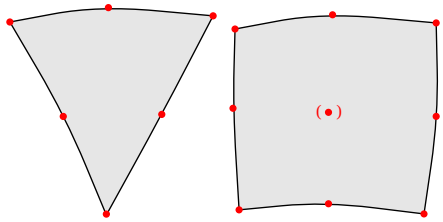


Figure 6: A triangular mesh element with six supporting points and a quadrilateral mesh element with eight supporting points on the boundary and one (optional) point in the interior.

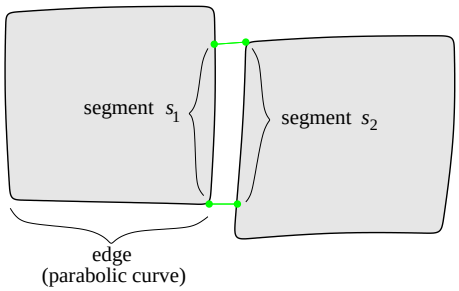


Figure 7: Two neighboured mesh elements and the segments s_1 and s_2 that constitute this neighbourhood.

can be three or four sided (Figure 6). The edges are parabolic curves, and the whole element is a special case of a *bicubic patch* as it is defined in chapter 5 of [9]. Basically this means that the restrictions imply that the three sided patch is uniquely determined by six points. The four sided mesh elements exist in two versions, which are defined with eight or nine points.

Figure 7 demonstrates that edges need not be neighboured on a whole edge. On the other hand, not only two, but even three or more mesh elements may have an edge in common (Figure 8). Figure 8 also shows a rim and a hole as two exemplary configurations where a mesh element is not neighboured to any other mesh element on an edge.

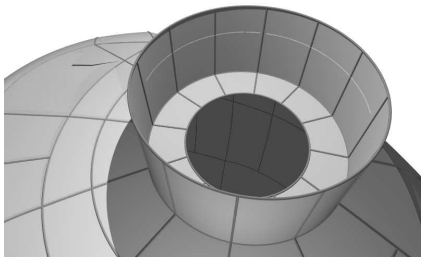


Figure 8: (workpiece 10) A mesh element can have edges that are not neighboured to any other edge. They may be around a *hole* or along a *rim*. Edges can also have more than one neighbour along so-called *branchings*.

Gaps. The main problem when finding neighbourhoods is that usually the CAD models contain significant gaps. These gaps have to be distinguished from holes that are intended by the designer of the CAD model. Consider two edges that are situated next to each other. The algorithm has to decide whether they are intended to be neighboured or just two separate edges with a hole in between. Since the gaps can be as large as in Figure 2 or 3, this decision becomes a serious problem. (A few possible reasons why gaps of this size may occur were discussed in Section 2.) The problem is made even more difficult by the occurrence of branchings: if for one of a pair of edges a neighbour has already been found, this does not exclude the possibility that the edges are nevertheless neighboured.

Discrete information. We have found gaps that are larger than mesh elements of the same workpiece. This has some unpleasant consequences, which are depicted in Figure 9. On the left side the upper and lower mesh elements are neighboured, whereas this is not the case on the right side, owing to the triangle between them.

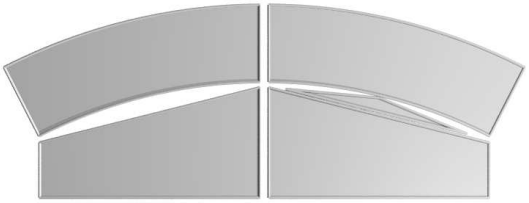


Figure 9: On the left side, the mesh elements are neighboured, but their mirror-symmetric counterparts on the right side are not, due to the triangle between them.

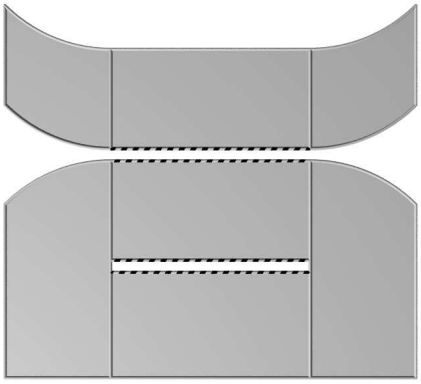


Figure 10: Two gaps of the same width and length. The surrounding mesh elements are a hint that the upper one is probably intended whereas the lower one is an error.

REMARK: The detection of this kind of configuration has been applied in [10]. It remarkably reduced the number of errors in the results of our algorithm.

Not only a mesh element between an examined pair of edges, but also the surrounding mesh elements must be incorporated into the interpretation of a local configuration. For example, in Figure 10 two gaps of the same length and width are shown. But in the upper case the gap continues on both sides and in the lower case the gap closes. This is a good indication that the first gap is intended as opposed to the second one. Roughly speaking, all of these considerations are examples of what we mean by incorporating discrete information.

Claim. We have seen that in certain situations discrete information must be used. Based on a systematic computational study, we will show that such situations occur too often to be ignored. They are typical, not pathological.

4. Methodology

Test cases. The workpieces that we have examined stem from industrial applications. Unfortunately it is quite difficult to get such instances because industrial companies keep them confidential. We do not even know which CAD applications they have used, and we are not allowed to distribute the CAD data sets to other researchers. However, the number of test cases is still high enough to demonstrate the trend to a negative result. We also know that the test cases were created with diverse applications in various companies.

We believe that it is impossible to systematically generate artificial instances that model realistic workpieces: a random set of parametric surface patches might never resemble a workpiece like a pump or a console. If we had introduced artificial gaps into closed surfaces to imitate the dirtiness of the data, we would not have examined the nature of common errors, but our interpretation of them.

Examined class of algorithms. In this paper, we consider the class of algorithms that work according to the following pattern: an algorithm decides whether or not a pair of edges is neighboured by comparing the distance of the edges with a threshold value. The algorithms differ only in the used distance measure. Furthermore, we assume that the algorithms always find the best threshold value for each workpiece. In the case of scaled distance functions, we even assume that the algorithms choose the best threshold value for each single edge. Both assumptions are by far more than one can expect from an algorithm without human interaction.

Reference neighbours. We use a set of *reference neighbours* for making statistics and assessing the output of different algorithms. Since the problem is so ill-posed, it is impossible to automatically check the output of an algorithm otherwise. The reference neighbourhoods were produced by a commercial CAD tool [4] and our algorithm in [10]. The result was checked manually using the method described in [10]. Where necessary the neighbourhoods were corrected in this project and the project of [5].

Potential neighbours. It does not suffice to count the number of reference neighbourhoods missed by an algorithm; we also have to test for pairs of edges that an algorithm erroneously considers as neighboured. For a fair test, we do not examine all pairs of edges, but only those that are not obviously wrong. For a more precise explanation, recall that each parametric surface patch is bordered by its trimming curves. Each edge is a segment of such a parameterized curve. The segment is defined by a parameter interval of the parameter of the curve. If the projection of another edge onto the curve does not intersect with this interval, we exclude this pair of edges from our investigations.

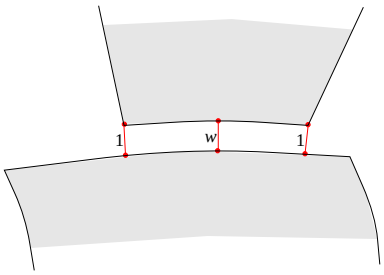


Figure 11: The distances between the end points are added to the distance between the middle points of the segments weighted by w to calculate the weighted distance.

Distance measures. We have examined two kinds of distance measures. The first kind of distance measures is computed by finding the mean value of the distances at ten pairs of mutually opposite points. These points are equidistant according to the parametrisation of the edge. We call this distance measure *uniform*.

The distance measures of the second kind are *weighted*. They result from the insight that an engineer sets the end points of an edge manually, whereas the interior of the edge is interpolated automatically. Therefore, the end points might be more precisely positioned than the interior of the edge. We measure the distance between two edges at three pairs of points. As in Figure 11, the distances at the end points are incorporated with weight 1 for the sake of symmetry. We refer to the individual distance measures by the weight w of the third distance, which is the only difference between them. We will discuss at the end of this section why we think these distance measures are good representatives.

Examined minima. For each distance measure and workpiece we produced a diagram like Figure 12. The threshold value varies on the abscissa. The decreasing curve shows the number of reference neighbours that are missed by using the specified threshold value, whereas the increasing line documents the number of pairs of edges that are incorrectly found to be neighboured.

There are two points of special interest in these diagrams. The first one is the total minimum number of faults – reference neighbours that are missed plus false neighbourhoods. The second interesting point is the minimum number is of wrong neighbours, subject to the condition that all reference neighbourhoods are found. In [10] we discussed why this minimum can be more important than the first one. For each distance measure and workpiece, we determined both values. This is what we meant above by claiming to assume that an algorithm always implicitly finds the best threshold value for every workpiece.

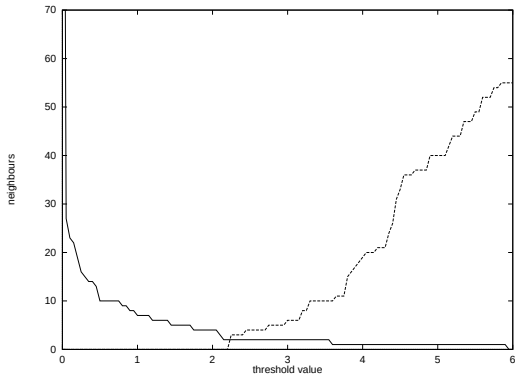


Figure 12: The number of reference neighbours that fail (decreasing line) and the number of pairs of edges that would be considered erroneously as neighboured (increasing line) for workpiece 8 and the uniform distance measure.

Relative error. Our experiments have shown that estimating the relative error with respect to geometric attributes does not produce better results than using the absolute error. For a meaningful visualization of the results, the estimation

$$\frac{\text{distance}}{\text{scaling}} \leq \text{threshold}$$

was replaced by

$$\text{distance} \leq \text{threshold} \cdot \text{scaling}$$

in the pictures and tables. We then determined all feasible threshold values for the (unscaled) uniform distance measure for every edge:

- The minimal feasible threshold value is the maximum of the distances to all reference neighbours.
- The maximal feasible threshold value is the minimum of the distances to all other edges.

A scaled threshold value for an edge between these bounds separates the correct neighbours of this edge from its false neighbours. We have examined the following four scalings:

1. the length of the edge,
2. the perimeter of the mesh element,
3. the area of the mesh element, and
4. the extent of the edge above a straight line through the end points of the edge, as shown in Figure 13. We call this distance the *curvature* of the edge.

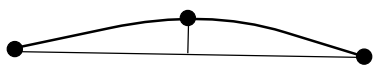


Figure 13: The *curvature* of an edge is measured by the distance from the supporting point in the middle of the edge to a line through the points at the ends of the edge.

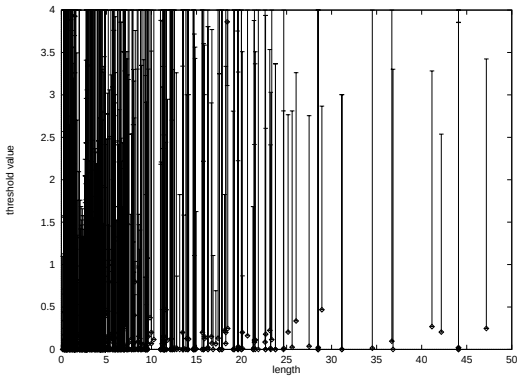


Figure 14: The possible threshold values for workpiece 12. Each interval shows the upper and the lower bound for one edge. Their position on the abscissa is the length of the edge.

We have generated a diagram for every scaling and workpiece. For every edge a vertical interval occurs in such a diagram. The intervals are positioned on the abscissa according to the value of the scaling for the particular edge. The lower and upper bound of the threshold determine the end points of the interval. An example is shown in Figure 14.

Our goal is to decide whether there is a reasonably simple relationship between geometric characteristics and feasible threshold values. In our opinion, a suitable visualization produces the most convincing arguments about the complexity of a possible interrelation. By that we mean a function that passes through all intervals. In particular, we determined the curve with minimal length in the set of continuous functions that pass between the permitted bounds.

This function with minimal length can be constructed automatically: the upper and lower bounds for each appearing value generate a polygon in the two dimensional plane. We seek the shortest path from the leftmost segment to the rightmost one. According to [6] Section 8.2.1, a shortest path in a polygon is a subpath of the visibility graph of the vertices of the polygon. If we replace the start and end points by segments, this solves our problem.

Representative distance measures. For performing our tests we had to choose different distance measures. We think that our distance measures are good representatives and include the most logical ones. It is obvious that the distance measure should rely on a metric of the three dimensional space. The metric has to be invariant under translation and rotation of the workpiece. Since the algorithm should produce the same result for a workpiece after scaling, the distance must be scalable, and thereby only those metrices that are derived from a norm are reasonable. Since all of those metrics are equivalent in a sense, we selected the Euclidean distance as a representative. A natural extension of the Euclidean distance between two points to a distance measure of edges is the (discrete approximation of the) area between the edges.

In this paper, we consider some related distance measures. Scalings other than the length are treated, and different weights are used. Nevertheless, our results provide a strong argument that this does not improve the situation.

Of course, one might argue that a significantly more complicated and less natural distance measure could solve the problem. However, the conclusions from the empirical results turned out not to be very sensitive to variations of the distance measure. Switching to another distance measure can improve or impair the quality of the output for a single workpiece, but on average the result does not change dramatically.

5. Computational Results

Absolute errors. We first discuss the quality of the output for distance measures that represent the absolute error estimation. All of these distance measures failed for some workpieces. The errors are out of magnitude for the subsequent tasks in the CAD process that use the neighbourhoods. Furthermore, none of the distance measures turned out to be generally preferable over the others.

In Table 1, the minima according to Section 4 are listed for individual distance measures and workpieces. The occasional great differences between good and bad results can only be explained by the size of the gaps in the workpieces: if gaps that are larger than mesh elements occur in a workpiece, the results are considerably worse. There are no other structural differences between the workpieces that could explain this effect. For model no. 6, the number of reference neighbours that were considered as wrong for a given threshold value did not drop to zero in the observed interval $[0, 10]$ for the uniform distance measure.

No.	# elements	neighbours		uniform		weighted 0.0		weighted 0.5		weighted 1.0		weighted 1.5	
		ref.	pot.										
1	34	55	1025	4	7	4	4	4	4	4	4	4	4
2	103	210	7860	0	0	0	0	0	0	0	0	0	0
3	295	676	35134	13	56	12	33	13	34	13	40	12	41
4	68	128	2304	0	0	0	0	0	0	0	0	0	0
5	251	573	23338	0	0	0	0	0	0	0	0	0	0
6	156	321	11525	26		18	72	18	72	20	72	22	72
7	342	823	48689	7	447	7	27	5	35	5	33	5	37
8	131	222	8669	2	55	0	0	0	0	1	5	2	11
9	530	1205	129596	0	0	0	0	0	0	0	0	0	0
10	608	1280	198055	0	0	0	0	0	0	0	0	0	0
11	24	40	567	0	0	0	0	0	0	0	0	0	0
12	179	409	20537	12	968	9	749	8	831	9	853	11	889
13	154	321	28860	3	3	3	3	2	2	2	2	2	3
14	237	446	29072	4	10	2	13	2	13	2	13	4	13
15	158	341	19085	0	0	3	3	0	0	0	0	0	0
16	156	261	17695	4	101	35	293	27	193	7	76	4	268

Table 1: Minimal number of wrong neighbourhoods for different distance measures. The weighted distance measure is determined according to Figure 11. For each distance measure, the first column lists the minimal number of false neighbourhoods – missing reference neighbours plus additional false neighbours. The second column contains the minimal number of errors, when all reference neighbours are found.

Number of supporting points. We first compare the uniform distance measure to a distance measure with only three measured distances, two at the end points and one in the middle of the edge. The results for the latter are listed in the column labelled "weighted 1.0". By comparing the two mentioned columns, the reader sees that the more fine-grained uniform distance measures does not lead to a better result. This realization explains furthermore why we did not consider more than three supporting points for weighted distance measures.

Weighted distances. We chose four different values for the weight in the middle of the edge. The search for a favorable weight appears fruitless: for every distance measure there is at least one data file where this weight does not seem appropriate. In most cases however, the choice of the weight only marginally influences the quality of the output: the results are similar for all four weights.

Relative errors. We now present the upper and lower bounds for the threshold value and discuss a possible relationship with scaling values of the mesh elements. In the data that is available to us, we even found some cases where the upper bound was smaller than the lower bound. To be able to proceed with our investigations, we increased the upper bound to match the lower one in those cases in which no feasible threshold value exists at all.

The appendix contains diagrams for all four scalings. They are typical in the sense that all of these functions "zigzag" irregularly, if there is no feasible unscaled threshold value for the workpiece. There are not even two functions that reveal any resemblance. These and all other diagrams show that there is probably not any interrelation between a suitable threshold value and the examined scalings.

6. Conclusion and Outlook

Summary. The common approach of using a distance measure and a global threshold value seems to fail for a variety of unscaled distance measures. Beyond this, we have seen that a relative error estimation does not improve the situation, even if the threshold value is individually chosen for each edge. The empirical results suggest that, without additional discrete information, the problem cannot be solved completely.

Source of discrete structure. Unfortunately, the topology is not available to an algorithm, since it is the output of the algorithm itself. As presented in [10], this conflict can be resolved. We first generated a rough approximation of the topology with a conventional approach. This source of a discrete structure was still good enough to be used by a discrete algorithm that applied logical inference rules as described above to correct errors in the topology. This dramatically improved our result.

Global properties. In some special cases, there exist global characteristics that can help reconstruct the topology. For example, in some applications we know that each edge is neighboured to at most or maybe even exactly one other edge. For instance, the surface of a solid object cannot contain any intended holes. A program processing such input can therefore safely close each gap since it knows that all gaps are errors.

Final remark. In analysing this problem, a difficulty becomes apparent that seems to be inherent to some practical problems. The standard stategy of analysing a problem, that is raising it to an abstract, mathematical model and solving the abstract problem efficiently, does not work here. One has to "stay in the mud."

References

[1] G. Barequet and M. Sharir. *Filling gaps in the boundary of a polyhedron*, in *Computer Aided Geometric Design* **12** (1995), 207-229.

[2] J. H. Bøhn and M. J. Wozny. *A Topology-Based Approach for Shell-Closure*, in *Geometric Modeling for Product Realization*, IFIP Transactions **B-8** (1993), 297-319.

[3] G. M. Fadel and C. Kirschman. *Accuracy issues in CAD to RP translations*, in *Internet Conference on Rapid Product Development*. MCB University Press, 1995. http://www.mcb.co.uk/services/conferen/dec95/rapidpd/fadel/fadel.htm.

[4] G. Krause. *Interactive finite element preprocessing with ISAGEN*, in *Finite Element News* **15**, 1991.

[5] R. H. Möhring, M. Müller-Hannemann, and K. Weihe. *Mesh refinement via bidirected flows; modelling, complexity, and computational results.* Journal of the ACM 44(3), 1997, 395-426.

[6] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[7] S. J. Rock and M. J. Wozny. *Generating topological information from a 'bucket of facets'*, in H.L. Marcus et al., editors, *Solid Freeform Fabrication Symposium Proceedings*, 1992, 251-259.

[8] X. Sheng and I. R. Meier. *Generating Topological Structures for Surface Models.* IEEE Computer Graphics and Applications Vol. 15, No. 6. 1995.

[9] B. Su and D. Liu. *Computational Geometry – Curve and Surface Modeling.* Academic Press, Inc., 1989.

[10] K. Weihe and T. Willhalm. *Reconstructing the Topology of a CAD model - a discrete approach*, in *Proceedings of the 5th European Symposium on Algorithms, ESA 1997*, Springer Lecture Notes in Computer Science 1284, 500-513.

Appendix: Diagrams about the Examined Scalings

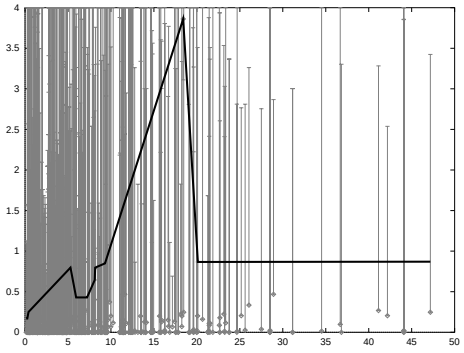


Figure 15: Approximating function for a possible threshold value depending on the length of edges for workpiece 12,

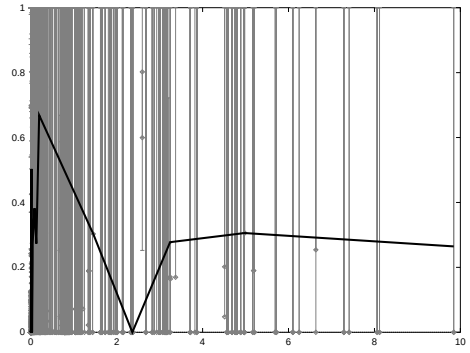


Figure 16: depending on the curvature of edges for workpiece 3,

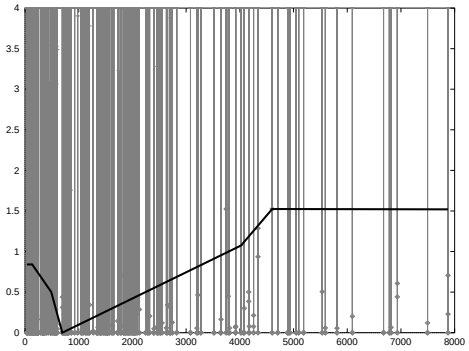


Figure 17: depending on the area of mesh elements for workpiece 14, and

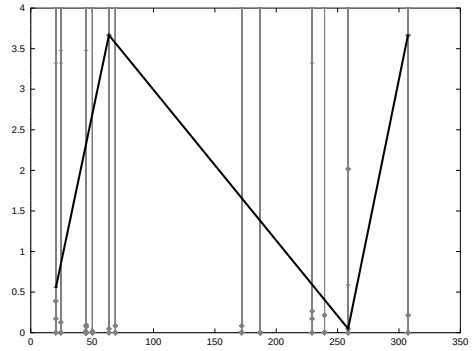


Figure 18: depending on the perimeter of mesh elements for workpiece 1.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 13–24

Efficient Implementation of an Optimal Greedy Algorithm for Wavelength Assignment in Directed Tree Networks

Thomas Erlebach[1]
Institut für Informatik, TU München, 80290 München, Germany
e-mail: erlebach@in.tum.de

and

Klaus Jansen[2]
IDSIA Lugano, Corso Elvezia 36, 6900 Lugano, Switzerland
e-mail: klaus@idsia.ch

ABSTRACT

In all-optical networks with wavelength division multiplexing several connections can share a physical link if the signals are transmitted on different wavelengths. As the number of available wavelengths is limited in practice, it is important to find wavelength assignments minimizing the number of different wavelengths used. This path coloring problem is $\mathcal{NP}$-hard, and the best known polynomial-time approximation algorithm for directed tree networks achieves approximation ratio 5/3, which is optimal in the class of greedy algorithms for this problem. In this paper the algorithm is modified in order to improve its running-time to $O(T_{ec}(N, L))$ for sets of paths with maximum load L in trees with N nodes, where $T_{ec}(n, k)$ is the time for edge-coloring a k-regular bipartite graph with n nodes. The implementation of this efficient version of the algorithm in C++ using the LEDA class library is described, and performance results are reported.

1. Introduction

Data can be transmitted through optical fiber networks at speeds that are considerably higher than in conventional networks. One wavelength can transmit signals at a rate of several gigabits per second. Furthermore, an optical fiber link can carry signals on different wavelengths simultaneously (wavelength division multiplexing). In order to fully exploit the bandwidth offered by optical networks, efficient wavelength assignment algorithms are required. For an optical network and a set of connection requests, each connection request must be assigned a transmitter-receiver path and a wavelength, such that connections using the same directed fiber link are assigned different wavelengths. Such a wavelength assignment should be computed efficiently, and it should use as few distinct wavelengths as possible.

For directed tree networks, the best known polynomial-time wavelength assignment algorithm uses at most $(5/3)L$ wavelengths [4, 8], where L is the maximum load of a directed fiber link and, thus, a lower bound on the optimal number of wavelengths. The exact optimization problem is known to be $\mathcal{NP}$-hard even for binary trees [2].

In this paper, we present modifications of the algorithm from [4] that allow a more efficient implementation. Furthermore, we describe an implementation of the modified algorithm in C++ using the LEDA class library [10]. Finally, we present performance results obtained with the implementation.

[1]supported by DFG contract SFB 342 TP A7
[2]partially supported by the EU ESPRIT LTR Project NO. 20244 (ALCOM-IT)

2. Preliminaries

The optical network with directed tree topology is represented by a directed tree $T = (V, E)$ with $|V| = N$. Here, a directed tree is the graph obtained from an undirected tree by replacing each undirected edge by two directed edges with opposite directions. Each connection request in T is given by a transmitter-receiver pair (u, v) with $u, v \in V$ and $u \neq v$, and it corresponds to the directed path from u to v in T. We view the process of assigning wavelengths to connection requests as coloring the corresponding paths. Two paths *intersect* if they share a directed edge. A feasible coloring of a set R of directed paths is an assignment of colors to paths in R such that intersecting paths receive different colors. A path p *touches* a node $v \in V$ if it starts at v, ends at v, or contains v as an internal node.

Given a directed tree $T = (V, E)$ and a set R of directed paths in T, the *path coloring problem* is to find a feasible coloring of R minimizing the number of distinct colors used.

For a set R of directed paths in T, the *load* $L(e)$ of edge $e \in E$ is the number of paths in R using edge e. The maximum load $\max_{e \in E} L(e)$ is denoted by L. Obviously, L is a lower bound on the optimal number of colors in a feasible coloring for R.

We express the running-time of path coloring algorithms in terms of parameters N (number of nodes of the given tree network) and L (maximum load of a directed edge). Given these two parameters, the size of the input (which consists of the tree network and the set of connection requests) can still vary between $\Theta(N + L)$ and $\Theta(NL)$. Our implementation does not try to make use of optimization opportunities for inputs whose size is substantially less than $\Theta(NL)$. In fact, our algorithm assumes that the load on all edges is exactly L. If this is not the case, it adds dummy paths between neighboring nodes until this requirement is satisfied.

The undirected version of the path coloring problem, where colors must be assigned to undirected paths in an undirected tree, has been studied as well [13, 3]. In the directed version, coloring the paths touching one node is equivalent to edge-coloring a bipartite graph, which is easy, and the difficulty lies in combining the colorings obtained for different nodes. In the undirected version, coloring the paths touching one node is equivalent to edge-coloring a general graph, which is $\mathcal{NP}$-hard, and it is easy to combine the colorings obtained for paths touching different nodes to obtain a coloring of all paths without using additional colors. Therefore, any approximation algorithm for edge-coloring with approximation ratio ρ directly translates into a ρ-approximation algorithm for path coloring in undirected graphs. The best known edge-coloring algorithm has asymptotic approximation ratio 1.1 [12]. In the present paper, we deal only with the directed path coloring problem.

2.1. Greedy algorithms

It is known that the path coloring problem is $\mathcal{NP}$-hard even for binary trees [2]. Therefore, one is interested in polynomial-time approximation algorithms. All such algorithms studied so far are in the class of *greedy algorithms* [11, 7, 9, 8, 4]. Here, the term "greedy algorithm" refers to a path coloring algorithm with the following properties:

- Initially, the algorithm picks a start node $s \in V$ and assigns colors to all paths touching s. (**Initial Coloring**)
- Then the algorithm visits the remaining nodes of the tree T in dfs-order; when it visits node v, it assigns colors to all paths touching v that have not been assigned a color at a previous node. (**Coloring Extension**)
- Once a path has been assigned a color, its color is not changed at a later node.

Individual greedy algorithms known so far differ only in the implementation of the coloring extension substep. It is known that every greedy algorithm has approximation ratio at least 5/3

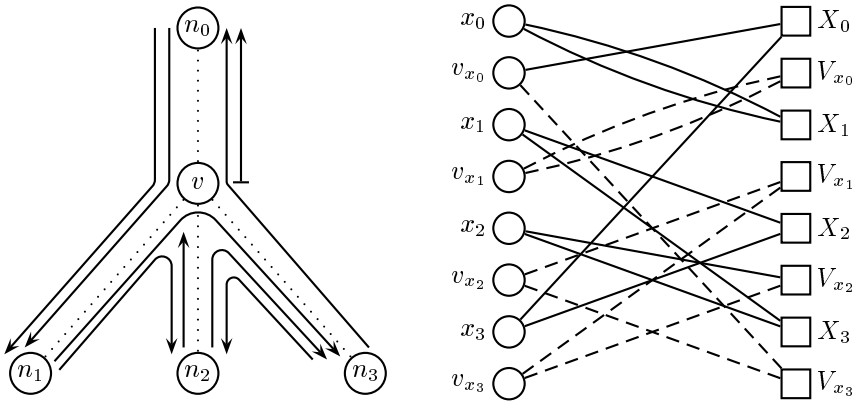


Figure 1: Construction of the bipartite graph G_v

in the worst case [6]. The algorithm from [4] achieves approximation ratio 5/3; therefore, we refer to the algorithm as an *optimal greedy algorithm*. Greedy algorithms can be implemented efficiently in a distributed setting: once the start node in the network has assigned wavelengths to all paths touching that node, it can transfer control to its neighbors, who can proceed independently and in parallel.

2.2. Constrained bipartite edge-coloring

The problem of coloring paths touching a node v can be reduced to the problem of coloring edges in a bipartite graph G_v. Denote by n_0 the parent of v and by $n_1, n_2, \ldots, n_k$ the children of v. The bipartite graph G_v has left and right vertex set $\bigcup_{i=0}^{k}\{x_i, v_{x_i}\}$ and $\bigcup_{i=0}^{k}\{X_i, V_{x_i}\}$, respectively. Each path touching v contributes one edge to G_v as follows:

- A path coming from a node n_i and heading for a node n_j contributes an edge (x_i, X_j).
- A path coming from a node n_i and terminating at node v contributes an edge (x_i, V_{x_i}).
- A path starting at node v and heading for a node n_i contributes an edge (v_{x_i}, X_i).

Note that all vertices x_i and X_i have degree L (recall that we have assumed load L on every edge of T), whereas the vertices v_{x_i} and V_{x_i} may have smaller degree. In order to make G_v L-regular, dummy edges are added between v_{x_i} vertices on the left side and V_{x_j} vertices on the right side, if necessary. (More precisely, for every edge (x_i, X_j) a dummy edge (v_{x_j}, V_{x_i}) is added.) Figure 1 illustrates this construction, with dashed lines indicating dummy edges.

It is easy to see that two paths touching v must be assigned different colors if and only if the corresponding edges in G_v share a vertex. Hence, any valid edge-coloring of G_v constitutes a valid coloring for the paths touching v.

If v is the start node, all paths touching v are uncolored, and any valid edge-coloring of G_v can be used to find the initial coloring of paths touching v. Otherwise, the paths touching v and n_0 have been colored at a previous node, and the edges incident to x_0 and X_0 in G_v have already received a color. We refer to these edges as *pre-colored* edges. In this case, the algorithm must find an edge-coloring of G_v that is consistent with the pre-colored edges. Hence, it must solve a *constrained bipartite edge-coloring problem* in order to carry out the coloring extension substep.

Each pair (x_i, X_i) or (v_{x_i}, V_{x_i}) constitutes a *row* of G_v. The row (x_0, X_0) is called the *top row* of G_v. The vertices in a row are said to be *opposite* each other. Note that vertices in the same row

of G_v (i.e., vertices that are opposite each other) can not be adjacent. A row *sees* a color c if an edge incident to a vertex of that row is colored with c. Two opposite vertices *share* a color c if each vertex in the row has an incident edge colored with c.

The colors that appear on pre-colored edges of G_v are called *single* colors if they appear only once, and *double* colors if they appear twice (i.e., they are shared by x_0 and X_0). Denote by S the number of single colors, and by D the number of double colors. Recall that only the $2L$ edges incident to x_0 and X_0 are pre-colored. If there are f different colors on these $2L$ edges, we have $S + D = f$ and $S + 2D = 2L$.

2.3. Edge coloring

A bipartite graph with maximum degree Δ can be edge-colored using Δ colors. Such an edge-coloring can be computed in time $O(m \log m)$ [1] or in time $O(\Delta m)$ [14], where m is the number of edges in the graph. The edges of a Δ-regular bipartite graph can be partitioned into Δ perfect matchings. One perfect matching or the whole partitioning can be computed in time $O(\Delta m)$ using the algorithm from [14]. In particular, a 3-regular bipartite graph with n vertices can be partitioned into three perfect matchings in time $O(n)$. This is necessary for coloring triplets (see Section 3) in linear time.

In the remainder of this paper, let $T_{ec}(n, k)$ denote the time for edge-coloring a k-regular bipartite graph with n nodes. We have $T_{ec}(n, k) \geq nk/2$ (as the graph contains $nk/2$ edges) and $T_{ec}(n, k) = O(\min\{k^2 n, kn \log(kn)\})$.

3. Algorithm Description

The greedy algorithm for path coloring in directed trees from [4] maintains the following two invariants:

Invariant 1: For any pair (u, v) of neighboring nodes in T, the number of different colors assigned to paths using the edge (u, v) or the edge (v, u) is at most $(4/3)L$.

Invariant 2: The number of different colors assigned to paths is at most $(5/3)L$.

The initial coloring at start node s is obtained by computing an edge-coloring with L colors for the L-regular bipartite graph G_s. Obviously, Invariants 1 and 2 are satisfied at this point. Now, each coloring extension substep must maintain the invariants. Before the coloring extension substep at node v, the paths touching v and its parent are colored using at most $(4/3)L$ colors. Hence, the pre-colored edges of G_v are colored using at most $(4/3)L$ colors.

Recall that S is the number of single colors and D is the number of double colors on pre-colored edges of G_v. Invariant 1 ensures that $S + D \leq (4/3)L$. Since we assume that every edge has load L, we have $S + 2D = 2L$ and, consequently, $D \geq (2/3)L$. Furthermore, one can assume that D is exactly $(2/3)L$. The reason is that, if D is greater than $(2/3)L$, one can simply "split" an appropriate number of double colors by assigning one of the two pre-colored edges colored with the same double color a new color for the duration of this coloring extension substep. A series of color exchanges can then re-establish the original colors on the pre-colored edges without violating the invariants. With $D = (2/3)L$, $S + 2D = 2L$ implies $S = (2/3)L$. Hence, among the $2L$ pre-colored edges of G_v, $S = (2/3)L$ are colored with single colors and $2D = (4/3)L$ are colored with double colors (each double color appears on exactly two pre-colored edges).

The goal of the coloring extension substep at node v is to find an edge-coloring of G_v that is consistent with the colors on pre-colored edges, that uses at most $(1/3)L$ colors not appearing on pre-colored edges (so-called *new* colors), and that uses at most $(4/3)L$ colors in each row of G_v (implying that Invariant 1 is maintained). This is achieved as follows:

(1) partition G_v into L perfect matchings

(2) group the matchings into chains and cycles

(3) select $L/3$ *triplets*, i.e., subgraphs consisting of three matchings each

(4) color the uncolored edges of each triplet using at most one new color and such that every row except the top row sees at most four colors

Every perfect matching obtained in step (1) contains two pre-colored edges: one pre-colored edge incident to x_0, and one incident to X_0. (Recall again that only the edges incident to x_0 and X_0 are pre-colored.) The matchings are classified according to the colors on their pre-colored edges: a matching is an SS-matching, if these colors are two single colors; an ST-matching, if one color is a single color and the other is a double color; a TT-matching, if the colors are two distinct double colors; and a PP-matching, if both pre-colored edges have been assigned the same double color.

The original version of the algorithms in [4] required that the matchings obtained in step (1) contain a maximal number of PP-matchings, in the sense that the union of all SS-, ST-, and TT-matchings does not contain an additional PP-matching. However, whereas an arbitrary partitioning into L perfect matchings can be computed very efficiently using one of the algorithms from [1] or [14], it is not clear how this additional property can be achieved without substantial increase in running-time. In particular, a straightforward implementation (checking for each double color whether the union of all SS-, ST-, and TT-matchings contains a PP-matching with that double color) would require $O(L)$ calls to a maximum matching algorithm, and the maximum matching algorithm from [5] has running-time $O(\sqrt{n}m)$ for bipartite graphs with n vertices and m edges. The main improvement of the algorithm presented in this paper is a modification that allows step (1) to compute an arbitrary partitioning into L matchings.

At a node v with degree $\delta(v)$, the bipartite graph G_v has $n = 4\delta(v)$ vertices and $nL/2$ edges. Using an arbitrary edge-coloring algorithm for bipartite graphs, step (1) can be performed in time $T_{ec}(n, L)$. Step (2) can be performed in time $O(n + L)$, and the triplets can be selected and colored in time $O(nL)$ in steps (3) and (4). Hence, the running-time for the coloring extension substep is dominated by step (1) and amounts to $O(T_{ec}(n, L))$. Note that, by the reasoning above, the time required for step (1) in the original version [4] of the algorithm is $O(n^{1.5}L^2)$. Hence, the modified algorithm with running-time $T_{ec}(n, L) = O(\min\{nL^2, nL\log(nL)\})$ yields a significant improvement.

3.1. Grouping into chains and cycles

We say that two matchings are *adjacent* if the same double color appears on a pre-colored edge in each of the two matchings. Every matching is adjacent to at most two other matchings, because it contains only two pre-colored edges and each color appears on at most two pre-colored edges of G_v. Therefore, this adjacency relation can be used to group the matchings into *chains* (a chain is a sequence of at least two adjacent matchings, starting and ending with an ST-matching and consisting of TT-matchings in between) and *cycles* (a cycle is a sequence of at least two adjacent TT-matchings such that the first and the last matching are also adjacent; in a cycle consisting of two matchings, both matchings have the same two double colors on pre-colored edges). A cycle or chain consisting of ℓ matchings is called ℓ-cycle or ℓ-chain, respectively. Hence, the L matchings are grouped into chains, cycles, SS-matchings, and PP-matchings. This grouping can be implemented to run in time $O(L)$.

Next, the chains and cycles are preprocessed so that the resulting chains and cycles do not contain parallel pre-colored edges. If the i-th and j-th matching in a cycle or chain contain parallel pre-colored edges incident to, say, x_0, then the parallel pre-colored edges can be exchanged, and this results in a shorter cycle or chain (or SS-matching) and an additional cycle. This preprocessing can be implemented to run in time $O(n + L)$.

3.2. Coloring the triplets

Before we describe how the triplets are selected, we present the coloring method used for each triplet, because this coloring method determines the requirements for the selection of the triplets.

As introduced by Kumar and Schwabe in [9], we consider 3-regular subgraphs H of G_v with the following property: among the six pre-colored edges of H, two are colored with the same double color d, two are colored with single colors s and s', and two are colored with double colors a and b, possibly $a = b$; more precisely, colors s, a and d appear on edges incident to x_0, and colors s', b and d appear on edges incident to X_0. We refer to such subgraphs as KS-subgraphs. Generalizing the coloring methods from [9] and [4], we obtain the following new result:

Lemma 3.1. *Let H be a KS-subgraph with n vertices. If no row except the top row sees more than four colors on pre-colored edges, then the uncolored edges of H can be colored in time $O(n)$ using at most one new color and colors d, s and s' such that no row except the top row sees more than four colors.*

Sketch of Proof. Using the algorithm from [14], we can in time $O(n)$ either partition H into a gadget (a subgraph in which x_0 and X_0 have degree 3 and all other vertices have degree 2) and a matching on all vertices except x_0 and X_0, or into a PP-matching and a cycle cover. In the former case, the coloring method from [9] is applicable; in the latter case, the uncolored edges of the PP-matching can be colored with its double color, and the cycle cover can be colored using s, s' and a new color such that opposite vertices share at least one color. ∎

Hence, the algorithm tries to select triplets satisfying the condition of Lemma 3.1 whenever possible. As cycles and chains have been preprocessed to contain no parallel pre-colored edges, any KS-subgraph containing two matchings from the same cycle or chain satisfies the condition of Lemma 3.1. In other cases, the coloring methods from the following lemmas in [4] are used.

Lemma 3.2. *Let H_1 be a subgraph of G_v that is a gadget with single color s and double colors a and a' incident to x_0, and with single color s' and double colors b and b' incident to X_0. Assume that H_1 does not contain parallel pre-colored edges. Furthermore, assume that the set of pre-colored edges of G_v with colors in $\{a, a', b, b'\}$ does not contain parallel edges. Then H_1 can be colored in time $O(n)$ without using any new color such that opposite vertices (except x_0 and X_0) share at least one color.*

Lemma 3.3. *Let C be a cycle cover obtained as the union of an SS-matching and a TT-matching without parallel pre-colored edges. Denote its single colors by s and s', and its double colors by a and b. C can be colored in time $O(n)$ using only s and a new color such that every pair of opposite vertices (except x_0 and X_0) shares at least one color.*

3.3. Selecting the triplets

Recall that G_v contains $(2/3)L$ pre-colored edges with single colors and $(4/3)L$ pre-colored edges with double colors. Every triplet selected contains 2 pre-colored edges with single colors and 4 pre-colored edges with double colors, ensuring that the ratio between pre-colored edges with double colors and with single colors remains $2 : 1$. This implies that there are always sufficiently many SS-matchings and 2-chains to form such triplets.

The rules for choosing triplets are as follows. From a cycle of even length, combine two consecutive TT-matchings with an SS-matching, as long as SS-matchings are available, and group each remaining TT-matching with a 2-chain. If at least one 2-chain is available, a cycle of odd length can also be handled like this. From a chain of odd length, combine the first two matchings and the last one, and treat the remainder like a cycle of even length. If there are two chains of even length, the first of which has length > 2, combine the first two matchings of the first chain and the last matching of the second chain, combine the last two matchings of the first chain and the first matching of the second chain, and treat the remaining sequences of even length like cycles of even length. A 2-chain and a PP-matching, or an SS-matching and two PP-matchings, can be combined and colored as required without using any new color.

Having dealt with these easy cases, we are left with SS-matchings, at most one PP-matching, at most one chain of even length > 2, and a number of cycles of odd length. Pairs (C_1, C_2) of cycles of odd length are handled as follows. First, pick an arbitrary SS-matching M. Then find a matching M_1 from C_1 and a matching M_2 from C_2 such that neither C_1 nor C_2 contains a pre-colored edge that is parallel to a pre-colored edge of M. (Such matchings exist because C_1 and C_2 do not contain parallel pre-colored edges after preprocessing and because both have length at least three.) Let the colors on pre-colored edges of M_1 be a_1 and b_1, and those on pre-colored edges of M_2 be a_2 and b_2. The algorithm checks whether the set of pre-colored edges in G_v with colors in $\{a_1, b_1, a_2, b_2\}$ contains parallel edges. If this is the case, exchanging two edges turns C_1 and C_2 into one cycle C of even length, and C can be handled by combining pairs of consecutive TT-matchings with an SS-matching, taking care that all resulting triplets satisfy the condition of Lemma 3.1. Otherwise, the triplet (M, M_1, M_2) can either be partitioned into a gadget (for which Lemma 3.2 is applicable) and a matching (that can be colored with a new color), and the remaining parts of C_1 and C_2 can be handled like cycles of even length, or M_1 and M_2 can be rearranged into two new matchings M_1' and M_2' with double colors a_1 and b_2 in M_1', and a_2 and b_1 in M_2', so that C_1 and C_2 are again turned into a single cycle of even length that can be handled as above.

Handling an odd cycle C and an even chain A is done as follows. Pick an arbitrary SS-matching M. If C and A contain TT-matchings M_C and M_A, respectively, that do not have pre-colored edges parallel to pre-colored edges in M, the algorithm proceeds similar as in the case of pairs of odd cycles described above. If A does not contain a TT-matching without pre-colored edges parallel to a pre-colored edge in M, A must have length four and each of its two TT-matchings must have exactly one pre-colored edge parallel to a pre-colored edge in M. In that case, exchanging parallel pre-colored edges turns $A \cup M$ into a 3-chain (for which Lemma 3.1 is applicable) and either a 2-chain (which can be combined with the odd cycle C as described above) or an SS-matching and a PP-matching.

After combining pairs of odd cycles and (possibly) one odd cycle and one even chain, what remains (except SS-matchings) is one of the following: a PP-matching and an odd cycle, or a PP-matching and an even chain. If there is a PP-matching M_P and an odd cycle C, pick an arbitrary SS-matching M and a matching M_C from C such that M and M_C do not have parallel pre-colored edges. Color the triplet (M_P, M_C, M) by coloring M_P with its double color and coloring (M, M_C) using Lemma 3.3. Handle the remainder of C like a cycle of even length.

If there is a PP-matching M_P and an even chain A, again pick an arbitrary SS-matching M. If A contains a TT-matching M_A such that M and M_A do not contain parallel pre-colored edges, proceed as above. Otherwise, the chain A must have length four, and each of its two TT-matchings M_1 and M_2 has exactly one of its two pre-colored edges parallel to a pre-colored edge of M. In this case, (M_P, M_1, M) and the three remaining matchings of A each constitute a KS-subgraph for which Lemma 3.1 is applicable.

Implementing the algorithm as outlined above, it is not difficult to achieve running-time $O(nL)$ for selecting and coloring all triplets.

3.4. Total running-time

The initial coloring at start node s takes time $O(T_{ec}(\delta(s), L))$. Each coloring extension step at a node v takes time $O(T_{ec}(\delta(v), L))$. Hence, the total running-time T is

$$T = O(\sum_{v \in V} T_{ec}(\delta(v), L)) = O(T_{ec}(N, L)).$$

Using the edge-coloring algorithm from [1], a running-time of $O(NL(\log N + \log L))$ is achieved. Using the algorithm from [14], a running-time of $O(NL^2)$ is obtained.

4. Implementation

We have implemented the approximation algorithm for path coloring in directed trees as outlined in the previous section. The implementation was carried out in C++ using the LEDA class library [10]. The performance results in Section 5 were obtained using LEDA 3.5.2, but the implementation has also been tested with LEDA 3.6.1. In addition to the algorithm itself, a graphical user interface allowing demonstration of the algorithm has been implemented (see Section 4.2). Schrijver's algorithm [14] has been used as the edge-coloring subroutine, implying that the implemented path coloring algorithm has a worst-case running-time of $O(NL^2)$ for inputs consisting of a tree with N nodes and paths with maximum load L. No efforts have been made to tune the implementation and gain a constant factor in running-time by standard techniques. In particular, all validity tests (see Section 4.1) and debug routines have been left in the code.

The various parts of the implementation of the algorithm are as follows:

- Coloring triplets according to Lemma 3.1: 3,500 lines of code (83,000 bytes)
- Coloring gadgets according to Lemma 3.2: 1,000 lines of code (25,000 bytes)
- Schrijver's edge-coloring algorithm [14]: 600 lines of code (11,500 bytes)
- Constrained bipartite edge-coloring algorithm (using the three subroutines above; including preprocessing of chains and cycles, and selection of triplets): 3,600 lines of code (97,000 bytes)
- Main control structure of algorithm (visiting the nodes of the tree in DFS order and constructing the bipartite graphs): 500 lines of code (11,500 bytes)

This adds up to 9,200 lines of code (228,000 bytes). The comparatively big code size is mainly caused by the huge number of similar cases that must be distinguished and treated in a slightly different way in the coloring routines. The user interface of the algorithm is provided by a function `assign_wavelengths`, which takes as arguments a tree T and a list L of connection requests (pairs of nodes) in T. It returns the list of wavelengths (colors) assigned to the requests by the algorithm.

4.1. Testing and debugging

The components of the algorithm (in particular, subroutines for coloring different cases of triplets or gadgets) were implemented and tested independently before they were put together. Right from the start, assertions (using the C++ `assert`-macro) and validity tests were inserted into the code at numerous points. Almost every subroutine checks the consistency of its input and of its output, and the validity of certain assumptions about the data at various points during its execution. All these tests and assertions could be implemented without increasing the asymptotic running-time of the algorithm. We found this technique (adding `assert` statements and validity tests) extremely helpful, because it allowed us to recognize, pin down, and fix bugs very efficiently.

When the path coloring algorithm appeared correct and gave reasonable output on sample instances, we employed an automated test routine that repeatedly generates random requests in a tree and checks whether the output of the path coloring algorithm is consistent (number of colors used is below $(5/3)L$, intersecting paths are assigned different colors) on these inputs. This automated test revealed several more bugs, but again these bugs could be pinned down and fixed very quickly mainly because of the assertions and tests inserted in the code. The automated test routine did not reveal any further bugs in the present version of the implementation.

4.2. Graphical User Interface

The graphical front-end used for demonstration of the path coloring algorithm contains the following features:

1. edit the tree network using the graph editor supplied by the LEDA class `GraphWin`

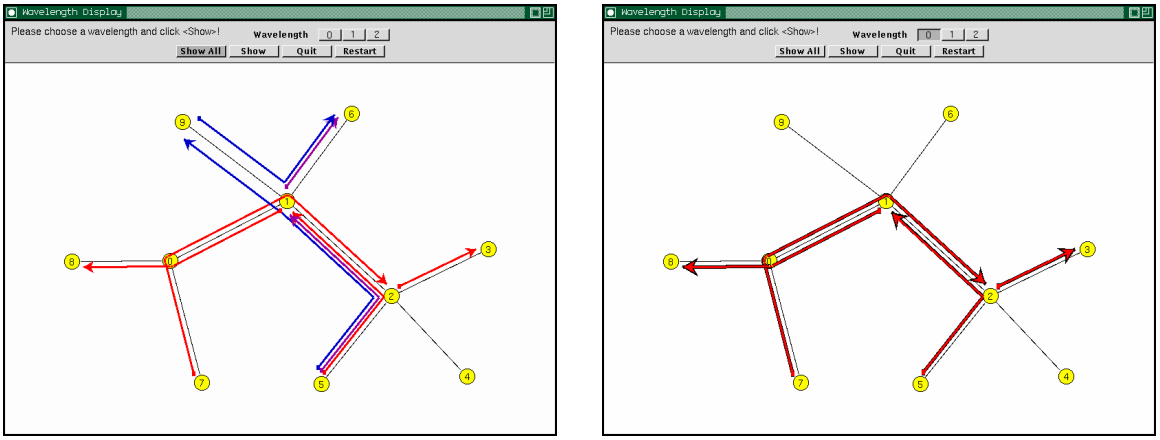


Figure 2: Screenshots displaying an instance with maximum load 3 for which the algorithm found an optimal assignment with 3 wavelengths (right side shows requests assigned wavelength 0)

2. enter connection requests manually by clicking transmitter and receiver node, or generate a number of requests randomly (transmitter and receiver are picked randomly according to a uniform distribution)

3. let the algorithm assign wavelengths to the requests

4. display the requests and their wavelengths as colored paths in the tree, either all requests simultaneously (for request sets with small maximum load) or only all requests that were assigned a certain wavelength

In addition, a "continuous" mode has been implemented that repeatedly generates random requests, assigns wavelengths, and displays the resulting assignments without any user interaction. Figure 2 shows screenshots of a request set that could be colored optimally with three wavelengths. The screenshot on the right side shows those requests that were assigned wavelength 0.

5. Performance Results

The performance of the implementation has been evaluated empirically with respect to two criteria: running-time, and number of wavelengths used. The left plot in Figure 3 shows how the running-time (CPU time) depends on the number N of nodes in the tree for fixed maximum load L; the right plot shows how the running-time depends on L if the number of nodes is kept fixed (here, $N = 100$ was used). The experiments were conducted on a Pentium PC (MMX, 166 MHz) running the Linux operating system. The variation of running-time between several runs with the same parameters was negligible. As expected, the plots confirm the linear dependency of the running-time on N and the super-linear (although not obviously quadratic) dependency on L. The possibly surprising ups and downs in the right plot are caused by the effect of the number of ones in the binary representation of L on the performance of edge-coloring algorithms for L-regular bipartite graphs (including the implemented algorithm from [14]) that use Euler partition in the case of even degree.

The trees used for the experiments were nearly balanced 5-ary trees, so that the number of internal nodes is approximately 1/5 of the total number of nodes. Each internal node has 5 children (except at most one internal node, which may have fewer children, so that the total number of nodes is N), and in the case of $L = 100$, for example, the resulting bipartite graphs G_v have

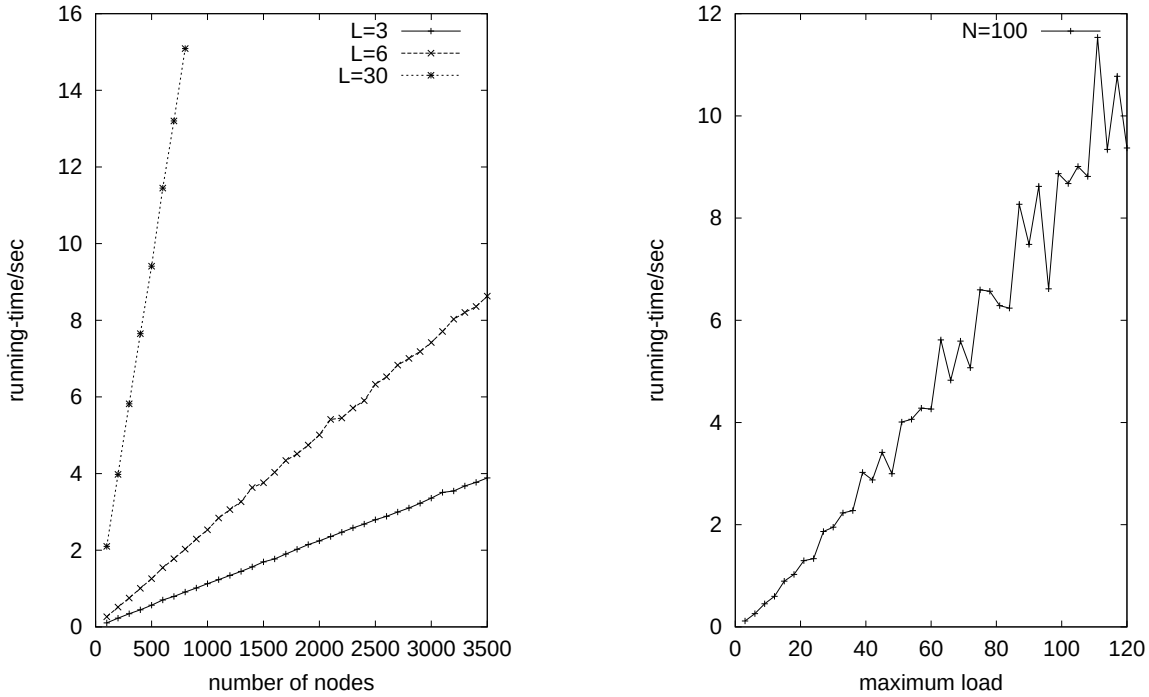


Figure 3: CPU times measured on a Pentium PC (MMX, 166 MHz) running Linux

$4 \cdot 6 = 24$ vertices and 1200 edges. (Changing the degree of the trees did not affect the running-time significantly; this observation is in agreement with the theoretical time bound $O(NL^2)$, which does not depend on the degree of the nodes in the tree. Our algorithm is even faster for 100-ary trees than for 5-ary trees with the same number of nodes, because only 1/100 of the nodes are internal nodes and the bipartite graphs G_v must be constructed only for internal nodes.) The sets of connection requests were generated randomly according to two different methods: the *random load* method creates a set R of requests by generating a given number of random requests (a random request is a request where sender and receiver are chosen randomly among all nodes of the tree according to a uniform distribution) and inserting all of them into the set R; the *full load* method generates a given number of random requests and inserts each request into R only if it doesn't make the maximum load greater than a given bound L, and then adds requests between neighboring nodes until the load on every directed edge is exactly L. The plots in Figure 3 were obtained using inputs generated by the *full load* method.

The number of wavelengths used by the algorithm was close to $(5/3)L$ in all experiments with inputs generated by the *full load* method, and close to $(4/3)L$ in all experiments with inputs generated by the *random load* method. The former observation can be explained by the "worst case affinity" of the algorithm: if the number D of double colors is more than $(2/3)L$ for a coloring extension substep, the algorithm gives away this possible advantage by temporarily splitting the double colors into single colors. The latter observation is caused by the special characteristics of load generated by the *random load* method: it is likely that the maximum load appears only on a few edges incident to the root of the tree, whereas the remainder of the tree is loaded only very lightly. For the same reason, the running-times for inputs generated by the *random load* method were also slightly smaller than for the *full load* method for the same values of N and L. Furthermore, we observed that the ratio between the number of colors used by the algorithm and the maximum load L became slightly

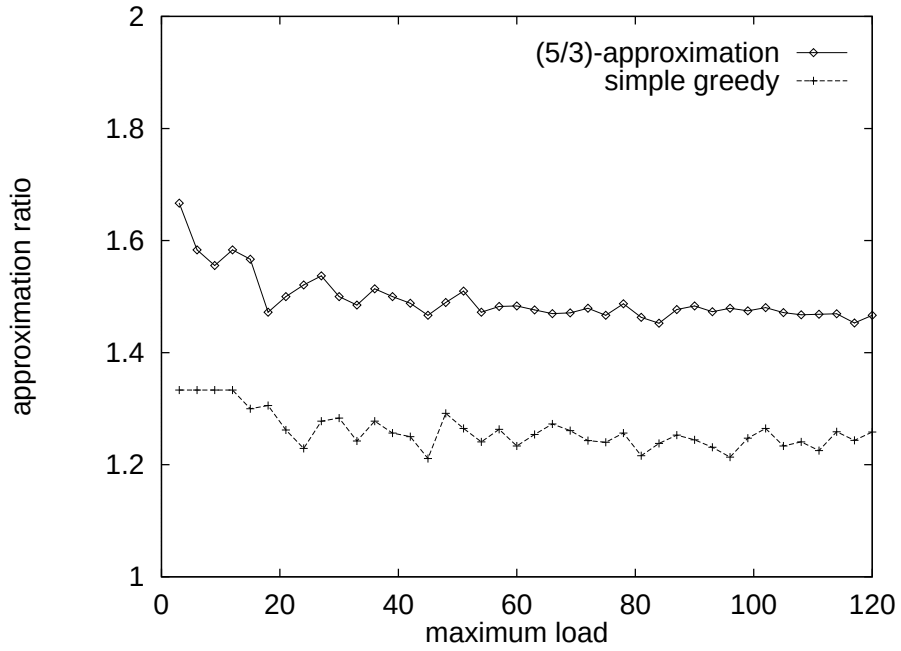


Figure 4: Approximation ratio achieved by the simple greedy algorithm and by the (5/3)-approximation algorithm for random requests in a balanced binary tree with 100 nodes

smaller when L was increased.

For comparison, we have also implemented the most simple greedy method for path coloring in directed trees: the nodes of the tree are visited in dfs-order, and at each node the uncolored paths touching that node are considered one by one and assigned the smallest available color. It is known that this simple greedy algorithm requires $2L-1$ colors in the worst case. However, our experiments show that it performs substantially better on randomly generated inputs. Figure 4 shows the ratio between the number of colors used by each of the two algorithms (the simple greedy algorithm and our (5/3)-approximation algorithm) and the maximum load L in experiments that were done with a (nearly) balanced binary tree with 100 nodes. The requests were generated according to the *full load* method. While the (5/3)-approximation algorithm used close to $(5/3)L$ colors (at least for small values of L), the simple greedy algorithm required only approximately $(4/3)L$ colors. Hence, for achieving good results in practice one should run both algorithms and use the better of the two colorings obtained; this way, the $(5/3)L$ worst-case guarantee can be combined with better average-case behavior.

6. Conclusion

In this paper, we have shown how the approximation algorithm for path coloring in directed trees from [4] can be modified so as to allow an efficient implementation whose running-time is dominated by the time for edge-coloring bipartite graphs that are constructed at each node of the network. Furthermore, we have presented an implementation of the algorithm in C++ using the LEDA class library. The observed CPU times match the theoretical bounds on the running-time. The number of colors required by the algorithm on fully loaded networks with randomly generated load is very close to the worst-case number of $(5/3)L$. Suspecting that the optimal number of colors for such instances is much closer to L (supported by the observation that the simple greedy method uses

only approximately $(4/3)L$ colors on these inputs), we think that additional heuristics are required to improve the number of colors used by the algorithm in the average case. (It is known that every greedy algorithm requires $(5/3)L$ colors in the worst case [6].) In particular, a promising approach might be to make better use of double colors if the number of colors appearing on pre-colored edges before a coloring extension step is smaller than $(4/3)L$.

References

[1] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM J. Comput.*, 11(3):540–546, August 1982.

[2] T. Erlebach and K. Jansen. Call scheduling in trees, rings and meshes. In *Proceedings of the 30th Hawaii International Conference on System Sciences HICSS-30*, volume 1, pages 221–222. IEEE Computer Society Press, 1997.

[3] T. Erlebach and K. Jansen. Scheduling of virtual connections in fast networks. In *Proceedings of the 4th Parallel Systems and Algorithms Workshop PASA '96*, pages 13–32. World Scientific Publishing, 1997.

[4] T. Erlebach, K. Jansen, C. Kaklamanis, and P. Persiano. An optimal greedy algorithm for wavelength allocation in directed tree networks. In *Proceedings of the DIMACS Workshop on Network Design: Connectivity and Facilities Location*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 40, pages 117–129. AMS, 1998.

[5] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, December 1973.

[6] K. Jansen. Approximation Results for Wavelength Routing in Directed Trees. In *Proceedings of IPPS '97, Second Workshop on Optics and Computer Science (WOCS)*, 1997.

[7] C. Kaklamanis and P. Persiano. Efficient wavelength routing on directed fiber trees. In *Proceedings of the 4th Annual European Symposium on Algorithms ESA '96*, pages 460–470, 1996.

[8] C. Kaklamanis, P. Persiano, T. Erlebach, and K. Jansen. Constrained bipartite edge coloring with applications to wavelength routing. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming ICALP '97*, LNCS 1256, pages 493–504. Springer-Verlag, 1997.

[9] V. Kumar and E. J. Schwabe. Improved access to optical bandwidth in trees. In *Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms SODA '97*, pages 437–444, 1997.

[10] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38, 1995.

[11] M. Mihail, C. Kaklamanis, and S. Rao. Efficient access to optical bandwidth. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 548–557, 1995.

[12] T. Nishizeki and K. Kashiwagi. On the 1.1 edge-coloring of multigraphs. *SIAM J. Disc. Math.*, 3(3):391–410, August 1990.

[13] P. Raghavan and E. Upfal. Efficient routing in all-optical networks. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing STOC '94*, pages 134–143, New York, 1994. ACM SIGACT, ACM Press.

[14] A. Schrijver. Bipartite edge-coloring in $O(\Delta m)$ time. *SIAM J. Comput.*, 1997. To appear.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 25–36

Implementing a dynamic compressed trie

Stefan Nilsson
Helsinki University of Technology
e-mail: Stefan.Nilsson@hut.fi

and

Matti Tikkanen
Nokia Telecommunications
e-mail: Matti.Tikkanen@ntc.nokia.com

ABSTRACT

We present an order-preserving general purpose data structure for binary data, the LPC-trie. The structure is a highly compressed trie, using both level and path compression. The memory usage is similar to that of a balanced binary search tree, but the expected average depth is smaller. The LPC-trie is well suited to modern language environments with efficient memory allocation and garbage collection. We present an implementation in the Java programming language and show that the structure compares favorably to a balanced binary search tree.

1. Introduction

We describe a dynamic main memory data structure for binary data, the level and path compressed trie or LPC-trie. The structure is a dynamic variant of a static level compressed trie or LC-trie [2]. The trie is a simple order preserving data structure supporting fast retrieval of elements and efficient nearest neighbor and range searches. There are several implementations of dynamic trie structures in the literature [5, 6, 8, 17, 23]. One of the drawbacks of these methods is that they need considerably more memory than a balanced binary search tree. We avoid this problem by compressing the trie. In fact, the LPC-trie can be implemented using the same amount of memory as a balanced binary search tree.

In spatial applications trie-based data structures such as the quadtree and the octree are extensively used [30]. To reduce the number of disk accesses in a secondary storage environment, dynamic order-preserving data structures based on extendible hashing or linear hashing have been introduced. Lomet [22], Tamminen [33], Nievergelt et al. [24], Otoo [26, 27], Whang and Khrishnamurthy [35], Freeston [13], and Seeger and Kriegel [31] describe data structures based on extendible hashing. Kriegel and Seeger [19] and Hutflesz et al. [15] describe data structures based on linear hashing. All of the data structures mentioned above have been designed for a secondary storage environment, but similar structures have also been introduced for main memory. Analyti and Pramanik [1] describe an extendible hashing based main memory structure, and Larson [20] a linear hashing based one.

In its original form the trie [12, 14] is a data structure where a set of strings from an alphabet containing m characters is stored in a m-ary tree and each string corresponds to a unique path. In this article, we only consider binary trie structures, thereby avoiding the problem of representing large internal nodes of varying size. Using a binary alphabet tends to increase the depth of the trie when compared to character-based tries. To counter this potential problem we use two different compression techniques, path compression and level compression.

The average case behavior of trie structures has been the subject of thorough theoretic analysis [11, 18, 28, 29]; an extensive list of references can be found in Handbook of Theoretical Computer

Science [21]. The expected average depth of a trie containing n independent random strings from a distribution with density function $f \in L^2$ is $\Theta(\log n)$ [7]. This result holds also for data from a Bernoulli-type process [9, 10].

The best known compression technique for tries is path compression. The idea is simple: paths consisting of a sequence of single-child nodes are compressed, as shown in Figure 1b. A path compressed binary trie is often referred to as a Patricia trie. Path compression may reduce the size of the trie dramatically. In fact, the number of nodes in a path compressed binary trie storing n keys is $2n - 1$. The asymptotic expected average depth, however, is typically not reduced [16, 18].

Level compression [2] is a more recent technique. Once again, the idea is simple: subtries that are complete (all children are present) are compressed, and this compression is performed top down, see Figure 1c. Previously this technique has only been used in static data structures, where efficient insertion and deletion operations are not provided [4]. The level compressed trie, LC-trie, has proved to be of interest both in theory and practice. It is known that the average expected depth of an LC-trie is $O(\log \log n)$ for data from a large class of distributions [3]. This should be compared to the logarithmic depth of uncompressed and path compressed tries. These results also translate to good performance in practice, as shown by a recent software implementation of IP routing tables using a static LC-trie [25].

One of the difficulties when implementing a dynamic compressed trie structure is that a single update operation might cause a large and costly restructuring of the trie. Our solution to this problem is to relax the criterion for level compression and allow compression to take place even when a subtrie is only partly filled. This has several advantages. There is less restructuring, because it is possible to do a number of updates in a partly filled node without violating the constraints triggering its resizing. In addition, this relaxed level compression reduces the depth of the trie even further. In some cases this reduction can be quite dramatic. The price we have to pay is the potentially increasing storage requirements. However, it is possible to get the beneficial effects using only a very small amount of additional memory.

2. Compressing binary trie structures

In this section we give a brief overview of binary tries and compression techniques. We start with the definition of a binary trie. We say that a string w is the *i-suffix* of the string u, if there is a string v of length i such that $u = vw$.

Definition 2.1. *A* binary trie *containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is empty.*
- *If $n = 1$, the trie consists of a node that contains the element.*
- *If $n > 1$, the trie consists of a node with two children. The left child is a binary trie containing the 1-suffixes of all elements starting with 0 and the right child is a binary trie containing the 1-suffixes of all elements starting with 1.*

Figure 1a depicts a binary trie storing 15 elements. In the figure, the nodes storing the actual binary strings are numbered starting from 0. For example, node 14 stores a binary string whose prefix is 11101001.

We assume that all strings in a trie are prefix-free: no string can be a prefix of another. In particular, this implies that duplicate strings are not allowed. If all strings stored in the trie are unique, it is easy to insure that the strings are prefix-free by appending a special marker at the end of each string. For example, we can append the string 1000... to the end of each string. A finite string that has been extended in this way is often referred to as a semi-infinite string or sistring.

A path compressed binary trie is a trie where all subtries with an empty child have been removed.

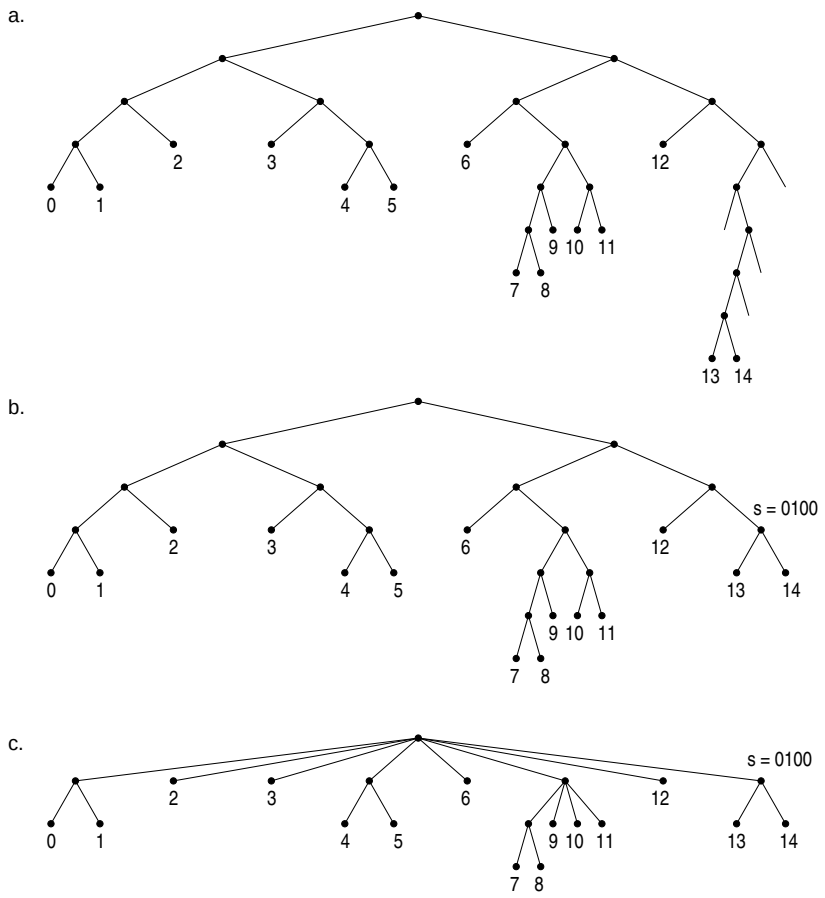


Figure 1: (a) A binary trie; (b) a path compressed trie; (c) a perfect LPC-trie.

Definition 2.2. *A* path compressed binary trie*, or Patricia trie, containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is empty.*
- *If $n = 1$, the trie consists of a node that contains the element.*
- *If $n > 1$, the trie consists of a node containing two children and a binary string s of length $|s|$. This string equals the longest prefix common to all elements stored in the trie. The left child is a path compressed binary trie containing the $(|s| + 1)$-suffixes of all elements starting with $s0$ and the right child is a path compressed binary trie containing the $(|s| + 1)$-suffixes of all elements starting with $s1$.*

Figure 1b depicts the path compressed binary trie corresponding to the binary trie of Figure 1a. A natural extension of the path compressed trie is to use more than one bit for branching. We refer to this structure as a level and path compressed trie.

Definition 2.3. *A* level and path compressed trie*, or an* LPC-trie*, containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is empty.*
- *If $n = 1$, the trie consists of a node that contains the element.*
- *If $n > 1$, the trie consists of a node containing 2^i children for some $i \geq 1$, and a binary string s of length $|s|$. This string equals the longest prefix common to all elements stored in the trie. For each binary string x of length $|x| = i$, there is a child containing the $(|s| + |x|)$-suffixes of all elements starting with sx.*

A perfect LPC-trie is an LPC-trie where no empty nodes are allowed.

Definition 2.4. *A* perfect LPC-trie *is an LPC-trie with the following properties:*

- *The root of the trie holds 2^i subtries, where $i \geq 1$ is the maximum number for which all of the subtries are non-empty.*
- *Each subtrie is an LPC-trie.*

Figure 1c provides an example of a perfect LPC-trie corresponding to the path compressed trie in Figure 1b. Its root is of degree 8 and it has four subtries storing more than one element: a child of degree 4 and three children of degree 2.

3. Implementation

We have implemented an LPC-trie in the Java programming language. Java is widely available, has well defined types and semantics, offers automatic memory management and supports object oriented program design. Currently, the speed of a Java program is typically slower than that of a carefully implemented C program. This is mostly due to the immaturity of currently available compilers and runtime environments. We see no reason why the performance of Java programs should not be competitive in the near future.

We have separated the details of the binary string manipulation from the trie implementation by introducing an interface `SiString` that represents a semi-infinite binary string. To adapt the data structure to a new data type, we only need to write a class that implements the `SiString` interface. In our code we give two implementations, one for ASCII character strings and one for short binary strings as found in Internet routers.

One of the most important design issues is how to represent the nodes of the trie. We use different classes for internal nodes and leaves. The memory layout of a leaf is straightforward. A leaf contains a reference to a key, which is a sistring, and a reference to the value connected with this key.

An internal node is represented by two integers, a reference to a `SiString` and an array of references to the children of the node. Instead of explicitly storing the longest common prefix string representing a compressed path, we use a reference to a leaf in one of the subtries. We need two additional integers, `pos` that indicates the position of the first bit used for branching and `bits` that gives the number of bits used. The size of the array equals $2^{\texttt{bits}}$. The number `bits` is not strictly necessary, since it can be computed as the binary logarithm of the size of the array.

The replacement of longest common prefix strings with leaf references saves us some memory while providing us access to the prefix strings from internal nodes. This is useful during insertions and when the size of a node is increased. An alternative would be to remove the references altogether. In this way we could save some additional memory. The drawback is that insertions might become slower, since we always need to traverse the trie all the way down to a leaf. On the other hand, a number of substring comparisons taking place in the path-compressed nodes of the trie would be replaced with a single operation finding the first conflicting bit in the leaf, which might well balance the extra cost of traversing longer paths. The doubling operation, however, would clearly be more expensive if the references were removed.

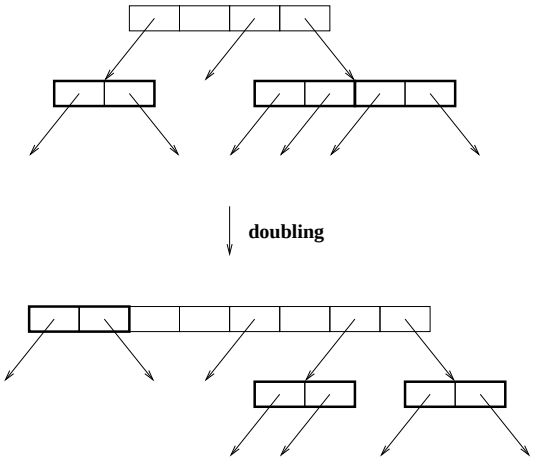


Figure 2: Node doubling.

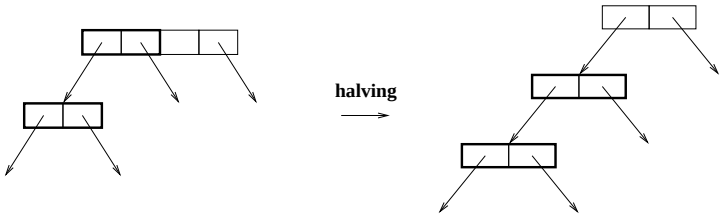


Figure 3: Node halving.

The search operation is very simple and efficient. At each internal node, we extract from the search key the number of bits indicated by `bits` starting at position `pos`. The extracted bits are interpreted as a number and this number is used as an index in the child array. Note that we do not inspect the longest common prefix strings during the search. It is typically more efficient to perform only one test for equality when reaching the leaf.

Insertions and deletions are also straightforward. They are performed in the same way as in a standard Patricia trie. When inserting a new element into the trie, we either find an empty leaf where the element can be inserted or there will be a mismatch when traversing the trie. This mismatch might happen when we compare the path compressed string in an internal node with the string to be inserted or it might occur in a leaf. In both cases we insert a new binary node with two children, one contains the new element and the other contains the previous subtrie. The only problem is that we may need to resize some of the nodes on the traversed path to retain proper level compression in the trie. We use two different node resizing operations to achieve this: halving and doubling. Figure 2 illustrates how the doubling operation is performed and Figure 3 shows the halving.

We first discuss how to maintain a perfect LPC-trie during insertions and deletions. If a subtrie of an internal node is deleted, we need to compress the node to remove the empty subtrie. If the node is binary, it can be deleted altogether, otherwise we halve the node. Note that it may also be necessary to resize some of the children of the halved node to retain proper compression.

On the other hand, it may be possible to double the size of a node without introducing any new empty subtries. This will happen if each child of the node is full. We say that a node is *full* if it

has at least two children and an empty path compression string. Note that it may be possible to perform the doubling operation more than once without introducing empty subtries.

When a node is doubled, we must split all of its full children of degree greater than two. A split of a child node of degree 2^i leads to the creation of two new child nodes of degree 2^{i-1}, one holding the 1-suffixes of all elements starting with 0 and one the 1-suffixes of all elements starting with 1. Once again, notice that it may also be necessary to resize the new child nodes to retain the perfect level compression.

In order to efficiently check if resizing is needed, we use two additional numbers in each internal node. One of the numbers indicates the number of null references in the array and the other the number of full children.

If we require that the trie is perfectly level compressed, we might get very expensive update operations. Consider a trie with a root of degree 2^i. Further assume that all subtries except one contain two elements and the remaining subtrie only one. Inserting an element into the one-element subtrie would result in a complete restructuring of the trie. Now, when removing the same key, we once again have to completely rebuild the trie. A sequence of alternating insertions and deletions of this particular key is therefore very expensive.

To reduce the risk of a scenario like this we do not require our LPC-trie to be perfect. A node is doubled only if the resulting node has few empty children. Similarly, a node is halved only if it has a substantial number of empty children. We use two thresholds: `low` and `high`. A node is doubled if the ratio of non-empty children to all children in the *doubled* node is at least `high`. A node is halved if the ratio of non-empty children to all children in the *current* node is less than `low`. These values are determined experimentally. In our experiments, we found that the thresholds 25% for `low` and 50% for `high` give a good performance.

A relatively simple way to reduce the space requirements of the data structure is to use a different representation for internal nodes with only two children. For small nodes we need no additional data, since it is cheap to decide when to resize the node. This will give a noticeable space reduction, if there are many binary nodes in the trie. In order to keep the code clean, we have not currently implemented this optimization.

4. Experimental results

We have compared different compression strategies for binary tries: mere path compression, path and perfect level compression, and path and relaxed level compression. To give an indication of the performance relative to comparison-based data structures, we also implemented a randomized binary search tree, or treap [32]. A study of different balanced binary search trees is beyond the scope of this article. We just note that the update operations of a treap are very simple and that the expected average path for a successful search is relatively short, approximately $1.4 \log_2 n - 1.8$.

A binary trie may of course hold any kind of binary data. In this study, we have chosen to inspect ASCII character strings and short binary strings from Internet routing tables. In addition, we evaluated the performance for uniformly distributed binary strings.

We refrained from low-level optimizations. Instead, we made an effort to make the code simple and easy to maintain and modify. Examples of possible optimizations that are likely to improve the performance on many current Java implementations include: avoiding synchronized method calls, avoiding the `instanceof` operator, performing function inlining and removing recursion, performing explicit memory management, for example by reusing objects, and hard coding string operations. All of these optimizations could be performed by a more sophisticated Java environment.

4.1. Method

The speed of the program is highly dependent on the runtime environment. In particular, the performance of the insert and delete operations depends heavily on the quality of the memory

management system. It is easier to predict the performance of a search, since this operation requires no memory allocation. The search time is proportional to the average depth of the structure. The timings reported in the experiments are actual clock times on a multi-user system.

When automatic memory management is used, it becomes harder to estimate the running time of an algorithm, since a large part of the running time is spent within a machine dependent memory manager. There is clearly a need for a standard measure. A simple measure would be to count the allocations of different size memory blocks. This measure could be estimated both analytically and experimentally. Accounting for memory blocks that are deallocated or, in the case of a garbage collected environment no longer referenced, is more difficult but clearly possible. To interpret these measures we need, of course, realistic models of automatic memory managers. We take the simple approach of counting the number of objects of different sizes allocated by the algorithm. That is, the number of leaves, internal nodes, and arrays of children pointers. Even this crude information turned out to be useful in evaluating the performance and tuning the code.

It is also a bit tricky to measure the size of the data structure, since the internal memory requirements of references and arrays in Java are not specified in the language definition. The given numbers pertain to an implementation, where a reference is represented by a 32-bit integer, and an array by a reference to memory and an integer specifying the size of the array.

We used the JDK (Java Development Kit) version 1.1.5 compiler from SUN to compile the program into byte code. The experiments were run on a SUN Ultra Sparc II with two 296-MHz processors and 512 MB of RAM. We used the JDK 1.1.5 runtime environment with default settings. The test data consisted of binary strings from Internet routing tables and ASCII strings from the Calgary Text Compression Corpus, a standardized text corpus frequently used in data compression research. The code and test data are available at URL `http://www.cs.hut.fi/~sni`

4.2. Discussion

Table 1 shows the average and maximum depths and the size of the data structures tested. We also give timings for inserting all of the elements (Put), retrieving them (Get), and deleting them one by one (Remove). The timings should be carefully interpreted, however, because the insertion and deletion times in particular depend very much on the implementation of the memory management system.

We use two variants of the relaxed LPC-trie. In the first variant, the `low` value 50% indicates the upper bound of the ratio of null pointers to all pointers in the *current* node and the `high` value 75% the lower bound of the ratio of non-null pointers to all pointers in the *doubled* node. In the second variant, the `low` and `high` values are 25% and 50%, respectively. Note that in all test cases the second variant outperforms the first one for our test data, even though the second variant has a poorer fill ratio. There is an interesting tradeoff between the number of null pointers and the number of internal nodes.

The trie behaves best for uniformly distributed data, but even for English text the performance is satisfactory. Interestingly, our experimental results agree with theoretical results on the expected number of levels for multi-level extendible hashing with uniformly distributed keys [34]. Level compression leads to a significant reduction of the average path length. The path compressed (Patricia) trie does not offer any improvement over the treap for our test data.

Figure 4 shows memory profiles for a sequence of insert and delete operations for English text. The amount of memory allocation needed to maintain the level compression is very small: The number of internal nodes allocated only slightly exceeds the number of leaves. However, we see that the algorithm frequently allocates arrays containing only two elements. This comes from the fact that to create a binary node, two memory allocations are needed: one to create the object itself and one to create the array of children. If we used a different memory layout for binary nodes we would reduce the number of allocated arrays considerably. For deletions very little memory management is needed. Comparing with Table 1 we may conclude that the level compression reduces the average depth of the trie structure from 20 to 9 using very little restructuring.

book1 (16622 lines, 16542 unique entries, 768770 characters)

	Depth		Put	Get	Remove	Size	
	Aver	Max	(sec)	(sec)	(sec)	(kB)	
Treap	16.2	30	1.5	1.3	1.3	323	
Patricia	20.2	41	3.9	1.3	2.6	388	
Perfect LPC	14.3	28	3.3	1.1	2.1	658	(382)
Relaxed LPC (50/75)	10.4	23	2.6	0.9	1.7	596	(403)
Relaxed LPC (25/50)	9.0	18	2.4	0.7	1.5	571	(391)

uniform random (50000 unique entries)

	Depth		Put	Get	Remove	Size	
	Aver	Max	(sec)	(sec)	(sec)	(kB)	
Treap	18.5	34	3.4	2.5	2.7	977	
Patricia	16.0	20	9.6	2.0	5.5	1171	
Perfect LPC	3.7	8	4.3	0.7	3.6	1635	(1076)
Relaxed LPC (50/75)	2.0	5	2.9	0.5	1.5	1246	(943)
Relaxed LPC (25/50)	1.6	4	2.1	0.5	1.2	1128	(908)

mae-east routing table (38470 entries, 38367 unique entries)

	Depth		Put	Get	Remove	Size	
	Aver	Max	(sec)	(sec)	(sec)	(kB)	
Treap	17.5	32	2.1	1.3	1.6	749	
Patricia	18.6	24	4.9	1.6	4.1	899	
Perfect LPC	5.8	13	4.0	0.7	3.3	1235	(825)
Relaxed LPC (50/75)	3.7	7	5.3	0.5	2.5	1006	(814)
Relaxed LPC (25/50)	2.9	5	4.5	0.4	2.1	955	(823)

Table 1: Some experimental results. The size figures in parentheses refer to a more compact trie representation.

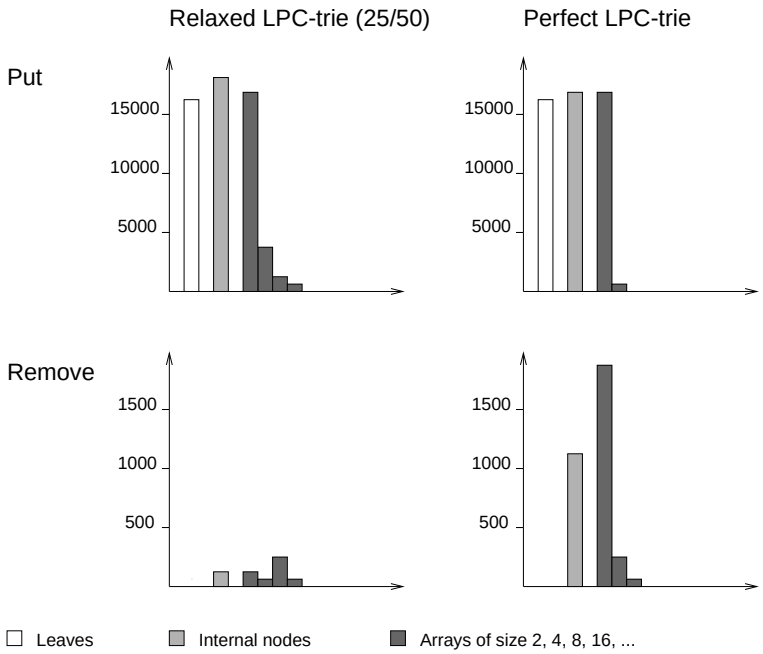


Figure 4: Memory profiles for book1 (16622 entries).

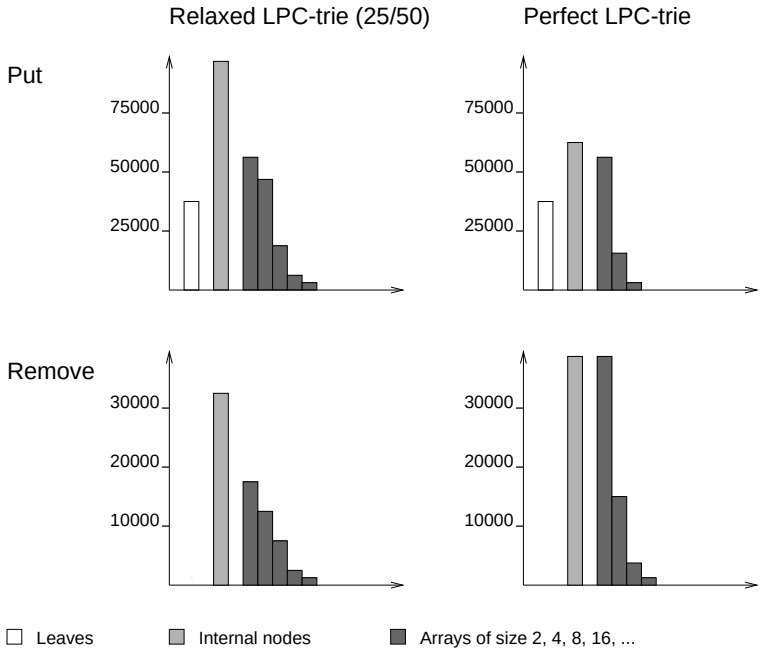


Figure 5: Memory profiles for mae-east (38470 entries).

For the routing table data the situation is different. In Figure 5 we see that the number of internal nodes and arrays allocated clearly exceeds the number of leaves. However, the extra work spent in restructuring the trie pays off. As can bee seen in Table 1, the average depth for of the Patricia tree is 18, the perfect LPC-trie has depth 6, and the relaxed LPC-trie depth 3. In this particular Java environment this reduction in depth is enough to compensate for the extra restructuring cost. The insertions times are, in fact, slightly faster for the level compressed tries as compared to a tree using only path compression. We also note that the naive implementation of the doubling and halving algorithms results in more memory allocation than would be strictly necessary and there seems to be room for improvement in running time by coding these operations more carefully.

In our implementation, the size of the trie structure is larger than the size of a corresponding binary search tree. However, using a more compact memory representation for binary nodes as discussed in Section 2 would give memory usage very similar to the treap. There are many other possible further optimizations. For data with a very skewed distribution such as the English text, one might introduce a preprocessing step, where the strings are compressed, resulting in a more even distribution [4]. For example, order preserving Huffman coding could be used.

5. Conclusions and Further Research

For both integers and text strings, the average depth of the LPC-trie is much less than that of the balanced binary search tree, resulting in better search times. In our experiments, the time to perform the update operations was similar to the binary search tree. Our LPC-trie implementation relies heavily on automatic memory management. Therefore, we expect the performance to improve when more mature Java runtime environments become available. The space requirements of the LPC-trie are also similar to the binary search tree. We believe that the LPC-trie is a good choice for an order preserving data structure when very fast search operations are required.

Further research is still needed to find an efficient node resizing strategy. Doubling and halving may introduce new child nodes that also need to be resized. In the current implementation, we recursively resize all of the new child nodes that fulfill the resizing condition. This may become too expensive for some distributions of data, because several subtries may have to be recursively resized. One way to avoid too expensive resizing operations is to delimit resizing to the nodes that lie along the search path of the update operation. This will make searches more expensive, but in an update intensive environment this might be the right thing to do. It is also possible to permit resizing to occur during searches in order to distribute the cost of resizing more evenly among operations.

Automatic memory management supported by modern programming language environments frees the application programmer from low level details in the design of algorithms that rely heavily upon dynamic memory. When the run time environment is responsible for memory management, it is possible to tailor and optimize the memory manager to take full advantage of the underlying machine architecture. This makes it possible to implement algorithms efficiently without explicit knowledge of the particular memory architecture. On the other hand, it is more difficult to take advantage of the fact that many algorithms need only a limited form of memory management that could be implemented more efficiently than by using a general purpose automatic memory manager. It also becomes more difficult to benchmark algorithms, when a large part of the run time is spent within the memory manager. There is clearly a need for a standardized measure to account for the cost of automatic memory management. A very interesting project would be to collect memory allocation and deallocation profiles for important algorithms and create performance models for different automatic memory management schemes. This should be of interest both to designers of general purpose data structures and automatic memory management schemes.

Acknowledgements

We thank Petri Mäenpää, Ken Rimey, Eljas Soisalon-Soininen, Peter Widmayer, and the anonymous referees for comments on the earlier draft of this paper.

Tikkanen's work has been carried out in the HiBase project that is a joint research project of Nokia Telecommunications and Helsinki University of Technology. The HiBase project has been financially supported by the Technology Development Centre of Finland (Tekes).

References

[1] A. Analyti, S. Pramanik. Fast search in main memory databases. *SIGMOD Record*, 21(2), 215–224, June 1992.

[2] A. Andersson, S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993.

[3] A. Andersson, S. Nilsson. Faster searching in tries and quadtrees – an analysis of level compression. In *Proceedings of the Second Annual European Symposium on Algorithms*, pages 82–93, 1994. LNCS 855.

[4] A. Andersson, S. Nilsson. Efficient implementation of suffix trees. *Software – Practice and Experience*, 25(2):129–141, 1995.

[5] J.-I. Aoe, K. Morimoto. An efficient implementation of trie structures. *Software – Practice and Experience*, 22(9):695–721, 1992.

[6] J.J. Darragh, J.G. Cleary, I.H. Witten. Bonsai: A compact representation of trees. *Software – Practice and Experience*, 23(3):277–291, 1993.

[7] L. Devroye. A note on the average depth of tries. *Computing*, 28(4):367–371, 1982.

[8] J.A. Dundas III. Implementing dynamic minimal-prefix tries. *Software – Practice and Experience*, 21(10):1027–1040, 1991.

[9] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20:345–369, 1983.

[10] P. Flajolet, M. Régnier, D. Sotteau. Algebraic methods for trie statistics. *Ann. Discrete Math.*, 25:145–188, 1985.

[11] P. Flajolet. Digital search trees revisited. *SIAM Journal on Computing*, 15(3):748–767, 1986.

[12] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[13] M. Freeston. The BANG file: a new kind of grid file. *ACM SIGMOD Int. Conf. on Management of Data*, 260–269, 1987.

[14] G.H. Gonnet, R.A. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, second edition, 1991.

[15] A. Hutflesz, H.-W. Six, P. Widmayer. Globally order preserving multidimensional linear hashing. *Proc of the 4th Int. Conf. on Data Engineering*, 572–587, 1988.

[16] P. Kirschenhofer, H. Prodinger. Some further results on digital search trees. In *Proc. 13th ICALP*, pages 177–185. Springer-Verlag, 1986. Lecture Notes in Computer Science vol. 26.

[17] D.E. Knuth. *TEX: The Program.* Addison-Wesley, 1986.

[18] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[19] H.-P. Kriegel, B. Seeger. Multidimensional Order Preserving Linear Hashing with Partial Expansions, *Proceedings of International Conference on Database Theory (Lecture Notes in Computer Science)*, Springer Verlag, Berlin, 203–220, 1986.

[20] P.-A. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4), 446 – 457, April 1988.

[21] J. van Leeuwen. *Algorithms and Complexity*, volume A of *Handbook of Computer Science*. Elsevier, 1990.

[22] D.B. Lomet. Digital B-trees. *Proc of the 7th Int. Conf. on Very Large Databases, IEEE*, 333–344, 1981.

[23] K. Morimoto, H. Iriguchi, J.-I. Aoe. A method of compressing trie structures. *Software – Practice and Experience*, 24(3):265–288, 1994.

[24] J. Nievergelt, H. Hinterberger, K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1), 38–71, 1984.

[25] S. Nilsson, G. Karlsson. Fast address lookup for internet routers. In Paul K editor, *Proceedings of the 4th IFIP International Conference on Broadband Communications (BC'98)*. Chapman & Hall, 1998.

[26] E.J. Otoo. A mapping function for the directory of a multidimensional extendible hashing. *Proc of the 10th Int. Conf. on Very Large Databases*, 491–506, 1984.

[27] E.J. Otoo. Balanced multidimensional extendible hash tree. *Proc 5th ACM SIGACT-SIGMOD Symposium on the Principles of Databases*, 491–506, 1985.

[28] B. Pittel. Asymptotical growth of a class of random trees. *The Annals of Probability*, 13(2):414–427, 1985.

[29] B. Pittel. Paths in a random digital tree: Limiting distributions. *Advances in Applied Probability*, 18:139–155, 1986.

[30] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1989.

[31] B. Seeger, H.P. Kriegel. The buddy-tree: an efficient and robust access method for spatial database systems. *Proc of the 16th Int. Conf. on Very Large Databases*, 590–601, 1990.

[32] R. Seidel and C.R. Aragon. Randomized binary search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[33] M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*. Mathematics and Computer Science Series No. 34, Helsinki, Finland, 1981.

[34] M. Tamminen. Two levels as good as any. *Journal of Algorithms*. 6(1):138–144, 1985.

[35] K.-Y. Whang, R. Krishnamurthy. *Multilevel grid files*. Research Report RC 11516 (#51719), IBM Thomas J. Watson Research Center, 43, 1985.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 37–48

Graph and Hashing Algorithms for Modern Architectures: Design and Performance

John R. Black, Jr. blackj@cs.ucdavis.edu
Charles U. Martel martel@cs.ucdavis.edu
Hongbin Qi qi@cs.ucdavis.edu

Department of Computer Science, University of California, Davis [1]
Davis, CA 95616, USA

ABSTRACT

We study the effects of caches on basic graph and hashing algorithms and show how cache effects influence the best solutions to these problems. We study the performance of basic data structures for storing lists of values and use these results to design and evaluate algorithms for hashing, Breadth-First-Search (BFS) and Depth-First-Search (DFS).

For the basic data structures we show that array-based lists are much faster than linked list implementations for sequential access (often by a factor of 10). We also suggest a linked list variant which improves performance. For lists of boolean values, we show that a bit-vector type approach is faster for scans than either an integer or character array. We give a fairly precise characterization of the performance as a function of list and cache size. Our experiments also provide a fairly simple set of tests to explore these basic characteristics on a new computer system.

The basic data structure performance results translate fairly well to DFS and BFS implementation. For dense graphs an adjacency matrix using a bit-vector is the universal winner (often resulting in speedups of a factor of 20 or more over an integer adjacency matrix), while for sparse graphs an array-based adjacency list is best.

We study three classical hashing algorithms: chaining, double hashing and linear probing. Our experimental results show that, despite the theoretical superiority of double hashing and chaining, linear probing often outperforms both for random lookups. We explore variations on these traditional algorithms to improve their spatial locality and hence cache performance. Our results also suggest the optimal table size for a given setting.

More details on these experiments can be found at: http://theory.cs.ucdavis.edu/

1. Introduction

Helping programmers create efficient code is an important goal of the study of algorithms. Many major contributions have been made in algorithm design, but some of these results need to be revisited to account for characteristics of modern processors. Design and analysis which focuses solely on instruction count may lead to faulty designs and misleading analysis. Caching as well as other features of the machine architecture may change the best strategies for designing algorithms. The importance of these effects is increasing as processor speeds outstrip memory access time with the cache-miss penalty now over 100 machine cycles on high-performance processors.

By measuring and analyzing the performance of fundamental data structures and algorithms we hope to provide a basic foundation for the design and evaluation of more complex algorithms. In this paper we study two fundamental topics using this approach: graph algorithms and hashing.

[1] This work was supported by NSF grant CCR 94-03651.

We show that significant performance improvements can be gained by using data structures and algorithms which take architectural features into account. We first study the performance of basic data structures in isolation and then use the indicated structures for Breadth-First-Search (BFS) Depth-First-Search (DFS) [7] and hashing. We also consider ways to predict performance and give indications of the effectiveness of these predictions by measurements of both actual run time and other statistics such as cache misses and number of instructions executed.

Our results show that the data structures for graphs found in almost all the standard texts often have much worse performance than simple alternatives. Since the extra overhead for using the standard data structure can be a factor of ten or more, it is well worth considering the performance effects we study when efficiency is important.

The Dictionary problem, where keys may be inserted, deleted and looked up, is one of the most fundamental uses of computers, and hashing is often the method of choice for solving it. Thus it is important to find the best practical hashing schemes and to understand the empirical behavior of hashing. While hashing algorithms have been studied extensively under traditional cost models, there has been little prior work focusing on their cache effects.

The desire to understand how different algorithms perform in practice has led to a recent increase in the experimental study of algorithms. There have been a number of experimental studies of graph algorithms which focus on important problems such as shortest paths [9, 5], minimum spanning trees (MST) [17], network flow and matching [1, 4, 10, 19], and min-cut algorithms [6]. These experiments provide valuable insight into the performance of different algorithms and can suggest new algorithmic choices. The authors of these studies reasonably spend most of their effort on the higher level algorithm details, so these papers have typically had a very limited discussion of the basic representation issues we discuss in this paper. In addition, we hope our results will help future experimental studies by suggesting good supporting data structures.

When doing experiments it is important to understand which variables can affect results. In many experimental papers (including all of those listed in the prior paragraph), the cache characteristics of the machines used are not even listed. In addition, in some of these studies different data structures were used for different algorithms which were being compared (for example linked lists for one and arrays for another). This shows the general lack of focus on these issues by experimenters.

1.1. Related Work

Because of its importance, compiler writers have spent considerable effort on generating code with good locality [3], however substantial additional improvements can be gained by proper algorithm design. Moret and Shapiro discuss cache effects on graph algorithms in their MST paper [17]. They comment that data caching and performance is affected by the method used to store a graph. They attempt to normalize for machine effects by using running times relative to the time needed to scan the adjacency structure. More recently, several researchers have focused on designing algorithms to improve cache performance by improving the locality of the algorithms [16, 13, 14]. Lebeck and Wood focused on recoding the SPEC benchmarks, and also developed a cache-profiler to help in the design of faster algorithms. LaMarca and Ladner came up with improved heap and later sorting algorithms by improving locality. They also developed a new methodology for analyzing cache effects [15]. While we didn't directly use their analysis tools since they were for direct-mapped caches and somewhat different access patterns, our analysis does use some of their ideas.

These papers show that substantial improvements in performance can be gained by improving data locality. Our results are similar in spirit to these but tackle different data structures and target different algorithms. Also, our approach focuses more on trying to understand the basic effects architectural features can have by studying them in simple settings.

A recent hashing paper [20] develops a collision resolution scheme which can reduce the probes compared to double hashing for some very specialized settings. However, since they only look at probes rather than execution time they don't address the effects we study here.

1.2. Result Summary

We start with the most basic data structures and compare arrays and linked lists (LL) for storing a sequence of integers when the basic operation is to scan consecutive elements in the list (with scans of an adjacency list structure or linear probing/chaining in mind). In this case reading all elements of an array can be 10 times faster than the equivalent LL scan. In addition, the performance gap is larger for more recent machines compared with older ones, so this disparity may grow in the future. By studying the architecture and compiler effects which slow down the LL implementation we develop LL variants which are much faster (though still slower than an array).

For lists of boolean values (with an adjacency matrix for an unweighted graph as our target) we compare integer, character and bitpacked arrays (where all the bits in a 32 bit word are used to store individual data elements). We show that even though the bitpacked array scans have higher instruction counts (due to overhead for extracting the bits), they outperform the integer and (usually) character arrays when the non-boolean list is too large to fit in cache. The bit matrix is often twice as fast as alternatives for large lists.

We study algorithms for Depth-First-Search (DFS), Breadth-First-Search (BFS), and hashing. For the graph algorithms we show that using a bitpacked array always outperforms an integer array (often by a factor of 20+ for BFS). In addition, the bitpacked array outperforms an adjacency list except for sparse graphs. Our experiments also confirm the substantial gain from using an array rather than a linked list to represent an adjacency list. Our results suggest that the best data structure depends largely on graph size and average node degree but not on graph topology.

In our hashing experiments we show that among traditional schemes Linear Probing is a clear winner over double hashing and chaining for both successful and unsuccessful search when the access pattern is uniform and multiple table entries fit in a single cache line. We suggest alternatives to double hashing and chaining which reduce cache misses and improve performance. We are able to model some of these settings to predict performance and the optimal size of a hash table.

2. Experimental Setting

We ran our experiments on five platforms: two DECstations, a 5000/25 and a 5000/240 (henceforth referred to as DEC0 and DEC1), two types of DEC Alphas: an older one with a 21064 processor and a newer one with a 21164 processor (Alpha0 and Alpha1) and a Pentium II (Pentium). We list the cache characteristics of the machines below since those are used directly in our analysis. DEC0 and DEC1 have a 64K byte cache and use 16-byte cache blocks. Both Alphas have an 8K byte on-chip direct mapped L1 data-cache and a 96K 3-way set-associative L2 cache (21064 is off-chip, 21164 is on-chip) and both use 32-byte cache blocks [8]. The Pentium runs at 266 Mhz, has an L1 cache of 16K for instruction and 16K for data, both 4-way associative with 32-byte line size. The L2 cache is 512K bytes and has a direct 133 MHz bus. There is also a prefetch buffer which fetches cache block $k+1$ whenever the current access is to block k.

Our programs were all written in C and compiled under highest optimization using cc. We did try alternative compilers but found there wasn't too much variation in results. In reporting the results we focus primarily on the Alpha1 and Pentium II results since they are the most relevant for current (and likely future) machines. However, we also report results on the older Alpha0 and the DECstations to show the changes which occur with the move to newer architectures. Fortunately the best design choice varies little between the five machines, though the degree of benefit varies considerably.

3. Basic Data Structure Results

The following experiments study basic algorithms and data structures. This allow us to study the performance effects closely, and also to suggest simple and robust structures for others to use.

Finally, it also gives a set of benchmark routines which test the relative performance on different machines.

3.1. Arrays versus Linked Lists

We start by comparing two data structures for holding the integers 1 through n when our goal is to scan these numbers. The main data structures we consider are a length n array of 32-bit integers (ARRAY) and a linked list of n nodes each of which contains a 32-bit integer and a pointer to the next node (LL) (32-bit pointers on the DECstations and Pentium, 64-bit pointers on the two Alphas). Each LL node is allocated by a call to `malloc()`.

Once the data structures are allocated and initialized, we repeatedly process the elements in order and add up all the numbers in the list. This is used as a simple surrogate experiment to represent a sequence of consecutive memory reads (but no memory writes). The goal with respect to graph algorithms, is to represent the operation of scanning all neighbors in a node's adjacency list where the list might be represented by an array or linked list.

Table 1 summarizes our basic data structure timing results. We list the per-element time in nanoseconds and the range of values of n (the number of items in the data structure) for which this time holds (when times were similar over a range we report only the median value). LL-12 is a linked list with 12 integers and one pointer per node.

Times in Nanoseconds per Element										
	DEC0		DEC1		Alpha0		Alpha1		Pentium	
Integer List Results (Time followed by n = list size)										
	Time	n	Time	n	Time	n	Time	n	Time	n
Array	114	to 16K	71	to 16K	19	to 2K	10	to 2K	12	to 64K
	410	$>$ 16K	114	$>$ 16K	35	3-128K	13	3-16K	26	128K
					60	$>$ 128K	17	32-128K	36	$\geq$ 256K
							30	$>$ 1M		
LL	144	to 4K	90	to 4K	34	to 256	10	to 256	24	to 1024
	1339	$>$ 7K	400	$>$ 7K	141	512-16K	34-70	to 4K	49	2K-8K
					387	$>$ 16K	121-167	to 65K	119	32K
							229	$>$ 128K	160	$\geq$ 64K
LL-12	131	to 8K	81	to 8K	27	to 1K	14	to 1K	27	to 16K
	358	at 16K	115	at 16K	39	at 2K	17	2-16K	30	to 64K
	529	$>$ 16K	141	$>$ 16K	50	to 64K	26	to 128K	49	128K
					86	$>$ 128K	34	$>$ 128K	58	$>$ 128K
Boolean list results										
Char	114	to 64K	71	to 64K	41	to 8K	27	to 8K	9	to 256K
	184	$>$ 64K	81	$>$ 64K	45	to 512K	28	to 128K	15	$\geq$ 512K
					51	$>$ 512K	30	to 1M		
Bit	149	to 32K	93	all	35	all	12	all	15	all
	157	$>$ 64K								

Table 1. Timings of Basic Data Structures for 5 Processors

For example, the top left entry shows that for an array of integers, the per-element time on DEC0 is 114 nanoseconds per-element for lists of up to 16K integers (which occupy 64K bytes), but jumps to 410 nanoseconds per element when the list is much above 16K elements and exceeds the 64K byte cache.

The integer list results show ARRAY is strictly better than LL and can be more than 10 times faster on the Alpha1 (13ns versus 128ns at $n = 8$K, 17ns versus 229ns at $n = 128$K) or Pentium (12ns versus 160ns for $n = 64$K). This performance gap is almost entirely explained by cache effects. When both data structures are small enough to fit entirely in the fastest (L1) cache they have comparable running times (except on the Pentium where LL is twice as slow even on small lists). However, as soon as the size exceeds that of the L1 cache for LL, its performance jumps. This is particularly serious since the L1 cache is exceeded for quite small lists.

A close look explains this effect quite clearly. Recall that the Alpha and Pentium L1 cache use 32-byte blocks. Each cache miss brings eight 4-byte integers into the cache, and on the Pentium a prefetch starts for the next eight. The LL uses more space per data element, particularly since `malloc()` allocates 32 bytes on the Alphas and 16 bytes on the DECstations and Pentium, even though only 12 and 8 are needed. Thus, each node access is in a separate cache line (except 2 per line on the Pentium) and only one data element is brought into the cache on a miss. On all machines the better spatial locality of the array structure greatly reduces its total cache misses when the array is too large to be contained in the L1 cache.

For an array, on the Alpha the total number of L1 cache misses is roughly $n/8$ when the array exceeds the L1 cache size, while LL has almost exactly n cache misses. The 32-bytes/node for LL also explains why its performance degrades significantly when n goes to 512 on the Alpha and its memory use exceeds the size of the L1 cache, while the Array performance does not drop until n goes above 2048 (since 512×32 bytes $= 2048 \times 4$ bytes $= 8$K $=$ size of L1 cache). There is also a second (expected) drop in performance when the data structure exceeds the size of the L2 cache (at 96K bytes) on the Alphas.

On the Pentium we also see the LL performance drop when the size exceeds the L1 cache size (1024×16 bytes), and again when its size exceeds the L2 cache size of $512\text{K} = 32\text{K} \times 16$. However, the ARRAY performance does not drop when its size exceeds the L1 cache size, but only when it reaches the L2 cache size (at $128\text{K} \times 4$ bytes). This is a strong statement on the effectiveness of the prefetching for sequential access. Each time we enter a new cache line, those data are already in the prefetch buffer.

Despite the large difference in running times, profiling shows that the array and linked list scans use the same number of instructions. Thus the time difference is due to memory effects.

We also stored multiple data values in a single linked list node. This packing of data values can greatly improve the performance of the LL structure. When two data items are stored in each node, the number of cache misses is roughly cut in half since the LL nodes have the same size as in the single data-item case.

On Alpha1 two-item-packing cuts the scan time in half (when the data exceed the L1 cache size). In fact up to four 4-byte integers can be put in a node without increasing the 32-byte size allocated by `malloc()`. Using nodes with four data-items per node results in an almost 4-fold speedup. We also tried using 12 data items per node (12 is the maximum number for a 64 byte allocation by `malloc()`). This resulted in a 7-fold speedup compared to the single node LL, but this still makes it almost 50% slower than ARRAY.

Our experiments using Atom [21] to study the cache misses on Alpha1 shows very much what we predicted. When n exceeds the cache size there are almost exactly $n/8$ cache misses for ARRAY reflecting the 8-word cache block brought in by each miss. For LL there are almost exactly n cache misses since each node uses an entire 32-byte cache line. Similarly, when we pack four data items per node, we now get four data items per cache line and $n/4$ cache misses.

3.2. Boolean Arrays of Integers, Characters and Bits

We study the best way to store a boolean array. Our motivation is for an adjacency matrix of an unweighted graph (as in BFS, topological sorting, and matching). We tested: an array of 4-byte integers (INT), an array of 1-byte characters (CHAR), and an array of bits (BIT) with 32 boolean values packed into a 4-byte integer.

As in the prior section, our test is to step through the array sequentially and add up all the values (in this case the bits). Profiling this code shows that the bit scan takes 50% more instructions than for the integer array. However, the improved locality more than makes up for this extra work. We could have also sped up the bit extraction by exploiting word-level parallelism (e.g. by table lookup for 8 or 16 bit blocks), but we avoided this since it may not generalize to other uses of the boolean array.

3.3. Results

The boolean list results of Table 1 show the timing effects for our settings. INT had the same results as Array (top table line). The only difference is adding ones instead of integers 1 through n. For the bit array the time per element is almost constant on each of the machines for all array sizes. This is not surprising since a cache line brings in 128 (DEC) or 256 (Alphas/Pentium) bits so any cache miss penalty is amortized over so much work it has a negligible effect. For CHAR there is only a small variation in per-element cost on the Alphas and DEC1, and a 50% change in cost on the DECstation and the Pentium. The higher CHAR cost on the Alphas reflects their extra cost for byte-level operations. We expect only a small change in cost as n varies since 32 data elements are being brought in with a cache line on the Alphas and Pentium.

Thus BIT is an attractive solution for large boolean arrays, and has its biggest gains over an integer array on the newer machines. Two other plus factors for BIT: (1) accesses to BIT will be less likely to evict other data items in the cache, and (2) accesses may be able to exploit word-level parallelism to reduce running time (as we do in our BFS and DFS programs). However, if an application only uses a few bits of the word in a window of time, then the performance may be worse than in our tests.

4. Graph Algorithm Results

Our graph experiments focused on the main graph traversal algorithms: DFS and BFS. We show that the performance predicted by the analysis above is exhibited by these algorithms.

We ran experiments on graphs from Knuth's Stanford GraphBase [12] and internally generated random graphs. GraphBase generates a variety of graphs which are both standard and available. We considered a range of graphs, but there was little difference in performance between random and structured graphs of the same size and density. This suggests that our results are largely independent of topology and rest primarily on the graph's density as discussed below. Thus we present here a representative subset taken from random undirected unweighted graphs with edges uniformly distributed over the vertices.

We focused on five data structures to represent our graphs: three adjacency lists, and two were adjacency matrices. The three adjacency list implementations were (1) a linked list, called "LL" above, (2) an array-based adjacency list [9], and (3) a "blocked" linked list where each node of the list contains multiple data items. The two adjacency matrix implementations were (1) a two-dimensional array of integers, and (2) a two-dimensional array of bits.

Results were collected on all platforms, but we focus on the Alpha1 results, noting significant differences where they occur.

In BFS when we visit a vertex we check all of its neighbors to determine which are unvisited (and add these to a queue). This operation is close to the scans discussed in section 3, so we expect the performance to generally reflect the results we saw for those basic data structures. In DFS we check neighbors until we hit an unvisited vertex, then change vertices, so we see less spatial locality than with BFS. DFS is further from the basic experiments, but still close enough for those results to provide good indications of performance.

4.1. Overall Results

For an n node graph with average vertex degree d our experiments showed that d and n/d were the overriding factors affecting the relative performances of the algorithms on the various data structures. Because DFS and BFS each look at an edge only once, the issue of whether the graph fits in cache is less important than if the algorithm examined edges multiple times. The degree d affects the spatial locality (when a block of data is brought into cache, how many are neighbors of the current node), and when d is small the fraction of time spent on edge scanning is also smaller, so the total speedup of the algorithm due to improving the edge scans is reduced. The ratio n/d is critical for the classic tradeoff between adjacency matrix structures and adjacency list structures.

We did experiments on a range of graphs, but present here a representative set of data for a family of random graphs, all having 1,000 nodes and between 6,000 and 20,000 edges and generated by the SGB. We present figures for these relatively small graphs for several reasons: (1) the SGB allows a standard widely-available source for graphs, and we were not able to produce much larger random graphs with it on our platforms, (2) we were able to run experiments on all data structures at this size (we could not test an integer-based adjacency matrix for a 20,000 node graph since we don't have 160M of memory to store it), and (3) experiments on larger graphs exhibited consistent performance with these smaller graphs: using our internal graph generators (which are fast and memory-efficient) we tested random graphs with 5,000, 10,000, and 20,000 nodes; these graphs had n/d values in the neighborhood of their tradeoff points (see below). For example, on the Alpha1 platform, we tested graphs of 20,000 nodes and from 5.6 million to 6.4 million edges.

On all platforms the relative speedups between competing structures remained constant and the n/d tradeoff point discussed below also remained constant, for a given platform. On the Alpha1 the tradeoff constant was about 33, while on the Pentium it was about 29.

By holding the number of nodes constant and varying the number of edges in the experiments presented, we were able to find which of the various data structures performed best for each of the graph densities. A plot of the running times on Alpha1 is presented in Figure 1.

The Y axis is the time, in seconds, for a BFS traversal. The integer adjacency matrix results are omitted since they would plot a horizontal line at 0.055 level, well off the top of our figure.

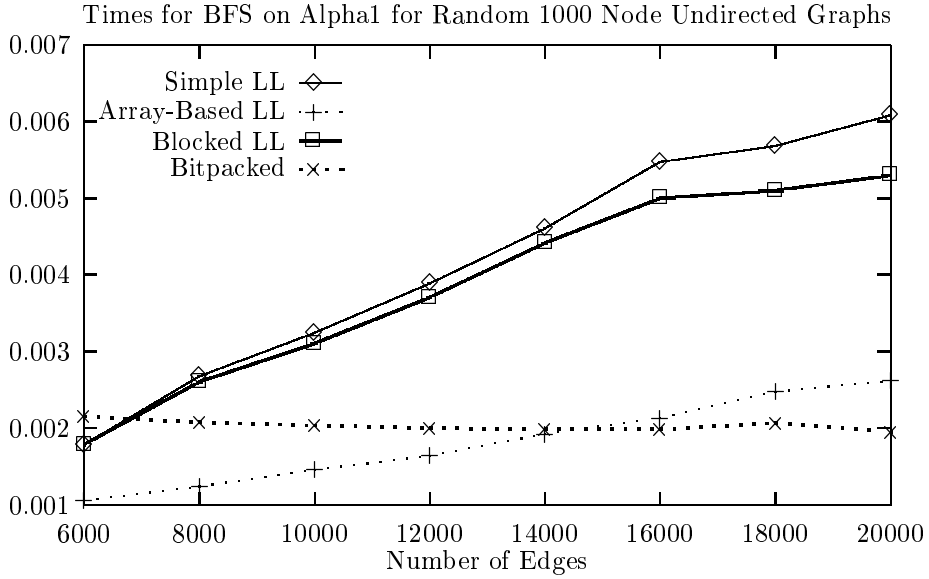


Figure 1

4.2. Bitpacked versus Integer Adjacency Matrices

Our bitpacked representation does vastly better than an integer array for all graphs tested, typically running in 1/20 the time or less. There are several reasons for this large disparity.

A principal cause is cache effects as discussed in section 3.2. The bitpacked representation also allows us to use several optimizations which exploit the inherent parallelism of word-level operations. For example, when visit the neighbors of a vertex, we can mark these as visited by a bitwise-OR of the adjacency bits into the bit array which tracks the visited vertices. We potentially mark 32 neighbors as visited in parallel.

The time required for packing and unpacking bits can be reduced substantially by a few tricks, which we also employed. It is not always necessary to shift and mask bits: we used techniques which extract a piece of the bit array (typically 4 or 8 bits) and then use a table lookup to quickly finish the operation.

4.3. Simple LL versus Array-Based and Blocked LLs

In Figure 1, we see that an array-based adjacency list always beats the LL structures, and a blocked LL beats a simple LL. The gains in adopting the array-based or blocked LLs are not as large as they were in section 3, since a smaller portion of the overall running time of this test is involved with scanning the adjacency lists. In section 3 the only task was to scan the list and add up numbers.

4.4. LLs versus Bitpacked Adjacency Matrix

The last two sub-sections gave clear indications regarding the correct choice of data structure when deciding between the pair being compared. In this setting, however, the better performer depends on the graph. This is completely expected: for sparse graphs with large n/d, the LL implementations need to scan only d items on average to process a given vertex whereas any adjacency matrix representation will require a scan of n objects. However, as we see from Figure 1, as n/d falls, the times for the LL structures increase linearly while the adjacency matrix times remain roughly constant. For our experiments the break-even point for the simple LL and the blocked LL is when n/d is about 70 (this is 7000 on the horizontal axis of Figure 1) and for the array-based LL the break-even point occurs at around $n/d = 33$. These numbers held constant over all graphs tested on Alpha1. On the other platforms there were similar constant trade-off points.

We should note that the algorithms tested here, BFS and DFS, were quite simple and perhaps atypical of many graph algorithms in that each data object was read only once. Although we did not experiment with other algorithms, we suspect that algorithms which access objects more than once, particularly with good spatial locality, would benefit even more from the nice behavior of the bitpacked adjacency matrix. This requires further exploration.

5. Hashing

Chaining, double hashing and linear probing [7, 11] are the three most classic hashing algorithms. Traditionally, chaining and double hashing are considered superior to linear probing because they disperse the keys better and thus require fewer probes. Our experiments show, however, that at least for uniform accesses, linear probing is fastest for insertions, successful searches and unsuccessful searches. This is true unless the table is almost 100% full or can be stored entirely in the L1 cache. Since it is rarely a good idea to have the hash table that full, linear probing seems to be the clear winner in the settings we considered.

5.1. Experimental Setting

We did experiments on both Alphas and the DECstations, but we focus here on the results for Alpha1 (the results for the other machines were generally similar). We used 8-byte keys on the

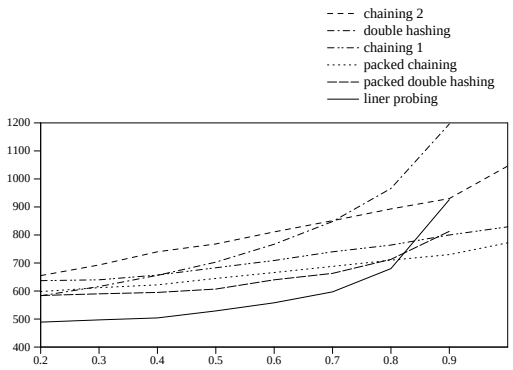


Figure 2. Time expense of insertions on the ALPHA. The X-axis is the load factor. The Y-axis is the average time in nanoseconds to insert a key. Insertions start with an empty table and end at the load factor on the X-axis. The table has 4M key slots.

Alpha1 and our tables contained only keys (plus 8-byte pointers for chaining). We chose hash(key) = key mod T as the hash function, where T is the table size. Table size is the maximum number of keys a table can hold. We distinguish table size from table space, which is the memory space a table occupies. To get a table with n keys we generate n random 32 bit integers and insert them into an empty table using the appropriate hashing scheme. Since we used random keys, the modulo function sufficed as our hash function.

5.1.1. Comparing Different Schemes

The performance of a hashing scheme will largely be determined by the number of probes and the number of cache misses. Double hashing (DH) and Chaining (C) do a good job of randomly distributing keys through the table. This reduces their number of probes, but also gives them very poor spatial locality. Successive probes looking for a key are almost certain to be in different cache blocks. Linear probing (LP) has excellent spatial locality but uses more probes. Figure 2 shows clearly that LP beats both DH and C for random insertions as long as the table is less than 80% full. The same results hold for successful search since the same probe sequences are used, and LP is also the best scheme for unsuccessful search, though its margin is somewhat smaller. Linear probing continued to be the best scheme over a wide range of table sizes where the number of keys did not fit into L1 cache but the table fit in main memory. When the key set size was small enough to fit entirely in L1 cache LP lost its advantage. LP's performance is well explained by its fewer cache misses which is detailed in the next section and by the analysis in section 5.3.2.

5.2. Analysis of the Number of Cache misses

We used Atom [21] to simulate our algorithms' cache behavior. We simulated a direct-mapped single-level cache which is the same as the DECstation cache. We chose to simulate the DECstation cache because a single-level cache would make our experimental results easier to analyze.

The number of cache misses roughly tracked the timing performance in Figure 2 and the analysis we do in section 5.3.2. Linear probing is a bit better than the cache-miss curves suggest and chaining somewhat worse. This may be due to the simpler address calculations in linear probing or due to easier optimizations of non-pointer based code by the compiler.

5.3. New Hashing Schemes

We designed variants on DH and C to improve their spatial locality. In *packed double hashing PDH*, we hash a key to a table entry which contains multiple key slots. The number of key slots in a table entry is set so that a table entry has exactly the same size as a cache block. When a key x is

hashed to entry i, the slots in that entry are examined sequentially. If x is not found and the entry has no empty slots, we compute a second hash function $h_2(x)$ as in double hashing, and examine entries $(i + increment)$ mod T, $(i + 2increment)$ mod T, ..., until x or an empty slot is found (note that now T represents the number of table entries, each of which contains multiple key slots).

In *Packed Chaining (PC)*, instead of storing one key and one pointer in a table entry or list node, we store multiple keys and one pointer, so that a table entry or list node has the same size as a cache block. When a key is hashed to a table entry, the key slots in that table entry are examined sequentially. If the the key or an empty slot is not found, we next check successive nodes in the associated linked list.

Figure 2 also reports the performance of PDH and PC. We see that they both improve on their normal versions, but they still perform less well than LP for moderate load factors. Measurements show that PDH and PC have the fewest cache misses of any of the schemes, however, they use more probes than linear probing for load factors below 0.8. However, these schemes do have the advantage that their performance is more stable as the load factor increases compared to LP.

Our successful and unsuccessful search results assume uniform access patterns. If the access pattern is skewed (as is true in many real applications) the number of cache misses will decrease and therefore chaining and double hashing may perform better. Minimizing probes will also be more important if key comparison is more expensive, and locality for linear probing will be reduced if other data is in the hash table or the keys are larger.

5.3.1. Optimal Table Size

In addition to the collision resolution scheme, the other main design choice in hashing is the table size. In a cache setting there is a tradeoff since a larger table will reduce the number of probes but may also increase the percentage of probes which are cache misses. We see in Figure 2 that the performance of all the hashing schemes monotonically decrease as the load factor increases. This same trend was seen for successful and unsuccessful search and for a wide range of key set sizes which did not fit in L1 cache. However, as in Figure 2, the performance was usually fairly flat for load factors in the 0.1 to 0.4 range.

It is also worth noting that since each scheme is best at relatively low load factors, and linear probing is a clear winner at these low load factors, linear probing looks to be a clear winner when one can predict the number of keys, and the desired hash table size is bigger than the L1 cache but fits in main memory.

5.3.2. Theoretical Analysis

The expected number of probes for uniform hashing is well analyzed [11], but adding a cache complicates things. However an approximate analysis helps explain the performance in Figure 2.

Assume we have n keys in a table of size T, and cache capacity C in units of table entries. Let $\alpha = n/T$ be the load factor and P the cache miss penalty. We assume a memory system with a single cache, where B is the number of table entries which fit into a cache block (so $B = 4$ on the Alpha1 if we store 8-byte keys in the table).

If T is much larger than C we assume that each access to a new cache block is a miss. Thus every probe for DH and C is a miss, and for LP if we do k probes in a single lookup, the first probe is a miss and subsequent probes have a $1/B$ chance of hitting a new block. Thus the expected number of misses is $1 + (k - 1)/B$.

We now consider DH and LP using the classic results for their expected number of probes for successful search. When $n/T = .5$ the expected number of probes (and cache misses) for DH is about 1.386 and 1.5 probes for LP. Thus if $B = 4$ as in our experiments, the expected number of cache misses for LP is $1 + .5/4 = 1.125$. When $n/T = .8$, DH takes 2 probes and LP 3, so LP has 1.5 expected misses compared to DH's 2. Thus it is not surprising LP is faster at both these load factors. A similar analysis can be done for chaining as well.

We can analyze *unsuccessful* searches for Double Hashing more exactly since each probe can be viewed as hitting a random location in the hash table, and each location in the hash table is equally likely to be in cache (these properties are not true for any other setting). For $T > C$, the probability that each location probed is not in the cache is approximated by $\frac{(T-C)}{T}$ (*). The expected number of probes for a random unsuccessful search is $\frac{1}{1-\alpha}$ [11]. So the expected cost of a random unsuccessful lookup is $\frac{1}{1-\alpha}\left(1 + \frac{T-C}{T}P\right)$

To study the behavior of this function with respect to T we take its derivative which is

$$\frac{PC - (P+1)n}{(T-n)^2}$$

The most interesting feature of the derivative is that it is always negative when $n > C$. Therefore if the keys do not fit in the cache, the expected cost keeps decreasing as we make the table bigger. This is true regardless of P, the cache miss penalty. We can extend this analysis to a two level cache as well, which shows that if n is larger than the size of the $L2$ cache it is optimal to keep increasing the table size (presumably up to the point where paging effects start and the models break down). If the key set is bigger than the L1 cache but smaller than the L2 cache, the models suggest setting T to the size of the L2 cache.

To test the predictions of the models, we used a key set which was larger than the $L2$ cache and varied the table size. The expected time for a random unsuccessful search did decrease as the table size increased, and at approximately the rate suggested by the models.

Unfortunately, other settings are more complex to model exactly. Consider random successful searches in double hashing. The expected number of probes for a random successful search is well known, but the probability that a probe will be a cache hit is more complicated than in the prior case. First, only those cache blocks which contain at least one key will ever be accessed during a successful search. Thus equation (*) is immediately invalid if we perform only successful searches. In addition, cache blocks which contain different number of keys have different probabilities of being in the cache. Consider the case where there is room for 4 keys in a cache block. A block B_4 with four keys is approximately four times as likely as a block B_1 with only one key to be in the cache, since it is almost four times as likely a key in B_4 was hit recently than the key in B_1.

6. Conclusions

We show that experimental studies of basic data structures provide useful insight into performance. This can be used to choose (and design) the proper data structures for larger applications.

There is considerable interesting followup work suggested by our results. For graphs the most immediate is to apply our design suggestions to other graph algorithms. Matching and unit Network flow are the most direct uses of our work since they use multiple scans of the arcs in an unweighted graph, possibly with multiple calls to BFS/DFS. Yet another extension is to apply a similar investigation to other basic data structures.

For hashing there are several important additional areas to study. First, it is important to consider various data sizes associated with the keys and larger keys such as strings. Second, it would be good to consider skewed access patterns, which occur quite often in real applications. Third, it would be good to study hashing when other memory intensive operations are being used.

References

[1] R. Ahuja, M. Kodialam, A. Mishra and J. Orlin. Computational testing of maximum flow algorithms. Sloan working Paper, MIT, 1992.

[2] Eli Biham. A fast new DES implementation in software. Technion, Computer Science Dept. Technical Report CS0891-1997.

[3] S. Carr, K. Mckinley, and C. Tseng. Compiler optimizations for improving data locality. In *Sixth ASPLOS*, 252-262, 1994.

[4] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or push? A computational study of bipartite matching and unit capacity flow algorithms. Technical Report 97-127, NEC Research Institute, Inc., August 1997.

[5] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, Vol. 73, 129-174, June 1996.

[6] C. Chekuri, A. Goldberg, D. Karger, M. Levine, and C. Stein, Experimental study of minimum cut algorithms. Technical Report 96-132, NEC Research Institute, Inc., October 1996.

[7] T. Cormen, C. Leiserson, R. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.

[8] Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual. Digital Equipment Corporation, Maynard, MA, 1997.

[9] G. Gallo and S. Pallottino. Shortest paths algorithms. *Annals of Operations Research*, vol. 13, pp. 3-79, 1988.

[10] D. Johnson and C. McGeoch Ed. Network Flows and Matching. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1993.

[11] Donald Knuth. Sorting and searching, the art of computer programming, Volume 3. Addison-Wesley Publishing Company, 1973.

[12] D. Knuth. The Stanford GraphBase. ACM Press, 1994.

[13] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *Journal of Experimental Algorithms*, Vol.1, 1996.

[14] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. In the *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 370-9, 1997.

[15] A. LaMarca and R. Ladner. Cache Performance Analysis of Algorithms. Preprint, 1997.

[16] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27(10):15-26, 1994.

[17] B. Moret and H. Shapiro. An Empirical Assessment of Algorithms for Constructing a Minimum spanning tree. *DIMACS Series in Discrete Math and Theoretical CS*, vol. 15, 99-117, 1994.

[18] R. Orni and U. Vishkin. Two computer systems paradoxes: serialize-to-parallelize and queuing concurrent-writes. Preprint 1995.

[19] J. Setubal. Sequential and parallel experimental results with bipartite matching algorithms. Technical Report EC-96-09, Institute of Computing, University of Campinas, Brasil, 1996.

[20] B. Smith, G. Heileman, and C. Abdallah. The Exponential Hash Function. *Journal of Experimental Algorithms*, Vol.2, 1997.

[21] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *ACM Symposium on Programming Language Design and Implementation*, 196-205, 1994.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 49–61

Implementation and Experimental Evaluation of Flexible Parsing for Dynamic Dictionary Based Data Compression

(extended abstract)

Yossi Matias [1]
Department of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel, and Bell Labs, Murray Hill, NJ, USA
e-mail: matias@math.tau.ac.il

Nasir Rajpoot [2]
Department of Computer Science, University of Warwick, Coventry, CV4-7AL, UK
e-mail: nasir@dcs.warwick.ac.uk

Süleyman Cenk Ṣahinalp [3]
Department of Computer Science, University of Warwick, Coventry, CV4-7AL, UK, and Center for BioInformatics, University of Pennsylvania, Philadelphia, PA, USA
e-mail: cenk@dcs.warwick.ac.uk

ABSTRACT

We report on the implementation and performance evaluation of greedy parsing with lookaheads for dynamic dictionary compression. Specifically, we consider the greedy parsing with a single step lookahead which we call Flexible Parsing ($\mathcal{FP}$) as an alternative to the commonly used greedy parsing (with no-lookaheads) scheme. Greedy parsing is the basis of most popular compression programs including UNIX `compress` and `gzip`, however it does not necessarily achieve optimality with regard to the dictionary construction scheme in use. Flexible parsing, however, is optimal, i.e., partitions any given input to the smallest number of phrases possible, for dictionary construction schemes which satisfy the prefix property throughout their execution. There is an on-line linear time and space implementation of the $\mathcal{FP}$ scheme via the trie-reverse-trie pair data structure [MS98]. In this paper, we introduce a more practical, randomized data structure to implement $\mathcal{FP}$ scheme whose expected theoretical performance matches the worst case performance of the trie-reverse-trie-pair. We then report on the compression ratios achieved by two $\mathcal{FP}$ based compression programs we implemented. We test our programs against `compress` and `gzip` on various types of data on some of which we obtain up to 35% improvement.

1. Introduction

The size of data related to a wide range of applications is growing rapidly. Grand challenges such as the human genome project involve very-large distributed databases of text documents, whose effective storage and communication requires a major research and development effort. From DNA and protein sequences to medical images (in which any loss of information content can not

[1] partly supported by Alon Fellowship
[2] supported by Quaid-e-Azam scholarship from the Government of Pakistan
[3] partly supported by NATO research grant CRG-972175 and ESPRIT LTR Project no. 20244 - ALCOM IT

be tolerated) vital data sources that will shape the information infrastructure of the next century require simple and efficient tools for lossless data compression.

A (lossless) compression algorithm $\mathcal{C}$ reads input string T and computes an output string, T', whose representation is smaller than that of T, such that a corresponding decompression algorithm $\mathcal{C}^{\leftarrow}$ can take T' as input and reconstruct T. The most common compression algorithms used in practice are the dictionary schemes (a.k.a. parsing schemes [BCW90], or textual substitution schemes [Sto88]). Such algorithms are based on maintaining a *dictionary* of strings that are called *phrases*, and replacing substrings of an input text with pointers to identical phrases in the dictionary. The task of partitioning the text into phrases is called *parsing* and the pointers replacing the phrases are called *codewords*.

A dictionary can be constructed in static or dynamic fashion. In *static* schemes, the whole dictionary is constructed before the input is compressed. Most practical compression algorithms, however, use *dynamic* schemes, introduced by Ziv and Lempel [ZL77, ZL78], in which the dictionary is initially empty and is constructed incrementally: as the input is read, some of its substrings are chosen as dictionary phrases themselves. The dictionary constructed by most dynamic schemes (e.g., [ZL77, ZL78, Wel84, Yok92]) satisfy the *prefix property* for any input string: in any execution step of the algorithm, for any given phrase in the dictionary, all its prefixes are also phrases in the dictionary.

In this paper we focus only on the the two most popular dictionary based compression methods: LZ78 [ZL78], its LZW variant [Wel84], and LZ77 [ZL77]. A few interesting facts about LZ78 and LZ77:

- The LZW scheme is the basis for UNIX `compress` program, `gif` image compression format, and is used in the most popular fax and modem standards (V42bis). LZ77 algorithm is the basis for all `zip` variants.
  Both algorithms: (1) are asymptotically optimal in the information theoretic sense, (2) are efficient, with $O(1)$ processing time per input character, (3) require a single pass over the input, and (4) can be applied on-line.

- LZ78 (and the LZW) can be implemented by the use of simple trie data structure with space complexity proportional to the number of codewords in the output. In contrast, a linear time implementation of the LZ77 builds a more complex suffix tree in an on-line fashion, whose space complexity is proportional to the size of the input text [RPE81].

- It is recently shown that LZ78 (as well as LZW) approaches the asymptotic optimality faster than LZ77: the average number of bits output by LZ78 or LZW, for the first n characters of an input string created by an i.i.d. source is only $O(1/\log n)$ more than its entropy [JS95, LS95]. A similar result for more general, unifilar, sources has been obtained by Savari [Sav97] - for the average case. For the LZ77 algorithm, this redundancy is as much as $O(\log\log n/\log n)$ [Wyn95]. Also, for low entropy strings, the worst case compression ratio obtained by the LZ78 algorithm is better (by a factor of 8/3) than that of the LZ77 algorithm [KM97].

- The practical performances of these algorithms vary however depending on the application. For example the LZ77 algorithm may perform better for English text, and the LZ78 algorithm may perform better for binary data, or DNA sequences. [4]

[4] A simple counting argument shows that there cannot exist a single dictionary construction scheme that is superior to other schemes for all inputs. If a compression algorithm performs well for one set of input strings, it is likely that it will not perform well for others. The advantage of one dictionary construction scheme over another can only apply with regard to restricted classes of input texts. Indeed, numerous schemes have been proposed in the scientific literature and implemented in software products, and it is expected that many more will be considered in the future.

Almost all dynamic dictionary based algorithms in the literature including the Lempel-Ziv methods ([ZL77, ZL78, Wel84, MW85, Yok92]) use *greedy parsing*, which takes the uncompressed suffix of the input and parses its longest prefix, which is a phrase in the dictionary. The next substring to be parsed starts where the currently parsed substring ends. Greedy parsing is fast and can usually be applied on-line, and is hence very suitable for communications applications. However, greedy parsing can be far from optimal for dynamic dictionary construction schemes [MS98]: for the LZW dictionary method, there are strings T which can be (optimally) parsed to some m phrases, for which the greedy parsing obtains $\Omega(m^{3/2})$ phrases.

For static dictionaries -as well as for the off-line version of the dynamic dictionary compression problem-, there are a number of linear time algorithms that achieve optimal parsing of an input string, provided that the dictionary satisfies the prefix property throughout the execution of the algorithm (see, for example, [FM95]). More recently, in [MS98], it was shown that it is possible to implement the one-step lookahead greedy parsing (or shortly flexible parsing -$\mathcal{FP}$) for the on-line, dynamic problem, in amortized $O(1)$ per character. This implementation uses space proportional to the number of output codewords. It is demonstrated that $\mathcal{FP}$ is optimal for dictionaries satisfying the prefix property in every execution step of the algorithm: it partitions any input string to minimum number of phrases possible while constructing the same dictionary. (For instance, the algorithm using the LZW dictionary together with flexible parsing inserts to the dictionary the exact same phrases as would the original LZW algorithm with greedy parsing.) The implementation is based on a rather simple data structure, the *trie-reverse-trie-pair*, which has similar properties with the simple trie data structure used for greedy parsing. It is hence expected that $\mathcal{FP}$ would improve over greedy parsing without being penalized for speed or space.

In this study, we report an experimental evaluation of $\mathcal{FP}$ in the context of LZW dictionary construction scheme. We implement compression programs based on $\mathcal{FP}$ (the implementations are available on the WWW [Sou]), and study to what extent the theoretical expectations hold on "random" or "real-life" data. In particular, we consider the following questions:

1. Is it possible to obtain a new dictionary construction scheme based on $\mathcal{FP}$? If yes, how would it perform in comparison to $\mathcal{FP}$ with LZW dictionary construction or the LZW algorithm itself - both asymptotically and in practice? (Note that the general optimality property of $\mathcal{FP}$ does not apply once the dictionary construction is changed.)

2. The trie-reverse-trie-pair is a pointer based data structure whose performance is likely to suffer from pointer jumps in a multi-layer memory hierarchy. Are there alternative data structures to obtain more efficient implementations of $\mathcal{FP}$ - in particular can we employ hashing to support dictionary lookups without all the pointer jumps?

3. What are the sizes of random data on which the better average case asymptotic properties of the LZ78 over LZ77 start to show up?

4. Does the worst case optimality of $\mathcal{FP}$ translate into improvement over greedy parsing on the average case?

5. Do better asymptotic properties of LZW in comparison to LZ-77 and $\mathcal{FP}$ in comparison to LZW show up in any practical domain of importance? Specifically how well does $\mathcal{FP}$ perform on DNA/protein sequences and medical images?

We address each one of these issues as follows:

1. We consider a data compression algorithm based on $\mathcal{FP}$, which constructs the dictionary by inserting it the concatenation of each of the substrings parsed with the character following them

- as in the case of LZW algorithm. We will refer this algorithm as the $\mathcal{FP}$-based-alternative-dictionary-LZW algorithm, or FPA. The dictionary built by FPA on any input still satisfies the prefix property in every execution step of the algorithm. In our experiments we consider the implementation of FPA as well as the implementation of the compression algorithm which builds the same dictionary as LZW, but uses $\mathcal{FP}$ for output generation which we refer as LZW-$\mathcal{FP}$. We compare the compression ratios obtained by LZW-$\mathcal{FP}$ and FPA with that of UNIX `compress` and `gzip`.

2. We present an on-line data structure based on Karp-Rabin fingerprints [KR87], which implements both LZW-$\mathcal{FP}$ and FPA in expected $O(1)$ time per character, by using space proportional to the size of the codewords in the output. We are still in the process of improving the efficiency of our implementations; we leave to report our timing results to the full version of this paper. We note, however, that our algorithms run about $3-5$ times slower than `compress` which is the fastest among all algorithms, both during compression and decompression. We also note that all the software, documentation, and detailed experimental results available on the WWW [Sou]. The readers are encouraged to check updates to the web site and try our software package.

3. We first demonstrate our tests on pseudorandom (non-uniform) i.i.d. bit strings with a number of bit probabilities. We observe that the redundancy in the output of each of the four programs we consider approach to the expected asymptotic behavior very fast - requiring less than 1KB for each of the different distributions, and better asymptotic properties of LZW in comparison to LZ77 can be very visible. For files of size $> 1MB$, `compress` can improve over `gzip` up to 20% in compression achieved. A next step in our experiments will involve pseudo-random sources of limited markovian order.

4. We report on our experimens with several "real-life" data files as well; those include DNA/protein sequences, medical images, and files from the Calgary corpus and Canterbury corpus benchmark suites. These results suggest that both LZW-$\mathcal{FP}$ and FPA are superior to LZW (UNIX `compress`) in compression attained, up to 20%. We also observe that both LZW-$\mathcal{FP}$ and FPA are superior to `gzip` for most non-textual data and all types of data of size more than $1MB$. For pseudo-random strings and DNA sequences the improvement is up to 35%. On shorter text files, `gzip` is still the champion, which is followed by FPA and LZW-$\mathcal{FP}$.

2. The Compression Algorithms

In this section we describe how each of the algorithms of our consideration, i.e., (1) the LZ77 algorithm (the basis for `gzip`), (2) the LZW variant (the basis for `UNIX compress`) of the LZ78 algorithm, (3) LZW-$\mathcal{FP}$ algorithm and (4) FPA algorithm, work. Each of the algorithms fit in a general framework that we describe below.

We denote a compression algorithm by $\mathcal{C}$, and its corresponding decompression algorithm by $\mathcal{C}^{\leftarrow}$. The input to $\mathcal{C}$ is a string T, of n characters, chosen from a constant size alphabet Σ; in our experiments Σ is either ascii or is $\{0,1\}$. We denote by $T[i]$, the i^{th} character of T $(1 \leq i \leq n)$, and by $T[i:j]$ the substring which begins at $T[i]$ and ends at $T[j]$; notice that $T = T[1:n]$.

The compression algorithm $\mathcal{C}$ compresses the input by reading the input characters from left to right (i.e. from $T[1]$ to $T[n]$) and by partitioning it into substrings which are called blocks. Each block is replaced by a corresponding label that we call a codeword. We denote the j^{th} block by $T[b_j : b_{j+1} - 1]$, or shortly T_j, where $b_1 = 1$. The output of $\mathcal{C}$, hence, consists of codewords $C[1], C[2], \ldots, C[k]$ for some k, which are the codewords of blocks $T_1, T_2, \ldots, T_k$ respectively.

The algorithm $\mathcal{C}$ maintains a dynamic set of substrings called the dictionary, $\mathcal{D}$. Initially, $\mathcal{D}$ consists of all one-character substrings possible. The codewords of such substrings are their characters themselves. As the input T is read, $\mathcal{C}$ adds some of its substrings to $\mathcal{D}$ and assigns them unique codewords. We call such substrings of T phrases of $\mathcal{D}$. Each block T_j is identical to a phrase in $\mathcal{D}$: hence $\mathcal{C}$ achieves compression by replacing substrings of T with pointers to their earlier occurrences in T.

The decompression algorithm $\mathcal{C}^{\leftarrow}$ that corresponds to $\mathcal{C}$, takes $C[1:k]$ as input and computes $T[1:n]$ by replacing each $C[j]$ by its corresponding block T_j. Because the codeword $C[j]$ is a function of $T[1:b_j-1]$ only, the decompression can be correctly performed in an inductive fashion.

Below, we provide detailed descriptions of each of the compression algorithms.

Description of the LZW Algorithm. The LZW algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b_m : b_{m+1}]$. Hence the phrases of LZW are the substrings obtained by concatenating the blocks of T with the next character following them, together with all possible substrings of size one. The codeword of the phrase $T[b_m : b_{m+1}]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. Thus, the codewords of substrings do not change in LZW algorithm. LZW uses greedy parsing as well: the m^{th} block T_m is recursively defined as the longest substring which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[b_{m+1}-1]$. Hence, no two phrases can be identical in the LZW algorithm.

Description of the LZW-$\mathcal{FP}$ Algorithm. The LZW-$\mathcal{FP}$ algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b'_m : b'_{m+1}]$, where b'_m denotes the beginning location of block m if the compression algorithm used were LZW. Hence for dictionary construction purposes LZW-$\mathcal{FP}$ emulates LZW: for any input string LZW and LZW-$\mathcal{FP}$ build identical dictionaries. The output generated by these two algorithms however are quite different. The codeword of the phrase $T[b'_m : b'_{m+1}]$ is the integer $|\Sigma|+m$, where $|\Sigma|$ is the size of the alphabet Σ. LZW-$\mathcal{FP}$ uses flexible parsing: intuitively, the m^{th} block T_m is recursively defined as the substring which results in the longest advancement in iteration $m+1$. More precisely, let the function f be defined on the characters of T such that $f(i) = \ell$ where $T[i:\ell]$ is the longest substring starting at $T[i]$, which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[\ell]$. Then, given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[\alpha - 1]$.

We demonstrate how the execution of the LZW and LZW-$\mathcal{FP}$ algorithms differ in the figure below.

Figure 1: Comparsion of $\mathcal{FP}$ and greedy parsing when used together with the LZW dictionary construction method, on the input string $T = a, b, a, b, a, b, a, a, b, a, a, b, a, a, a, b$.

Description of the FPA Algorithm. The FPA algorithm reads the input characters from left to right while inserting in $\mathcal{D}$ all substrings of the form $T[b_m : f(b_m) + 1]$, where the function f is as described in LZW-$\mathcal{FP}$ algorithm. Hence for almost all input strings, FPA constructs an entirely different dictionary with that of LZW-$\mathcal{FP}$. The codeword of the phrase $T[b_m : f(b_m) + 1]$ is the integer $|\Sigma| + m$, where $|\Sigma|$ is the size of the alphabet Σ. FPA again uses flexible parsing: given b_m, the integer b_{m+1} is recursively defined as the integer α for which $f(\alpha)$ is the maximum among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$.

Description of the LZ77 Algorithm. The LZ-77 algorithm reads the input characters from left to right while inserting all its substrings in $\mathcal{D}$. In other words, at the instance it reads $T[i]$, all possible substrings of the form $T[j : \ell]$, $j \leq \ell < i$ are in $\mathcal{D}$, together with all substrings of size one. The codeword of the substring $T[j : \ell]$, is the 2-tuple, $(i - j, \ell - j + 1)$, where the first entry denotes the relative location of $T[j : \ell]$, and the second entry denotes its size. LZ77 uses greedy parsing: the m^{th} block $T_m = T[b_m : b_{m+1} - 1]$ is recursively defined as the longest substring which is in $\mathcal{D}$ just before $\mathcal{C}$ reads $T[b_{m+1} - 1]$.

3. Data Structures and Implementations of Algorithms

In this section we describe both the trie-reverse-trie data structure, and the new fingerprints based data structure for efficient on-line implementations of the LZW-$\mathcal{FP}$, and FPA methods. The trie-reverse-trie pair guarantees a worst case linear running time for both algorithms as described in [MS98]). The new data structure based on *fingerprints* [KR87], is randomized, and guarantees expected linear running time for any input.

The two main operations to be supported by these data structures are (1) insert a phrase to $\mathcal{D}$ (2) search for a phrase, i.e., given a substring S, check whether it is in $\mathcal{D}$ and return a pointer. The standard data structure used in many compression algorithms including LZW, the compressed trie $\mathcal{T}$ supports both operations in time proportional to $|S|$. A compressed trie is a rooted tree with the following properties: (1) each node with the exception of the root represents a dictionary phrase; (2) each edge is labeled with a substring of characters; (3) the first characters of two sibling edges can not be identical; (4) the concatenation of the substrings on a path of edges from the root to a given node is the dictionary phrase represented by that node; (5) each node is labeled by the codeword corresponding to its phrase. Dictionaries with prefix properties, such as the ones used in LZW and LZ78 algorithms, build a regular trie rather than a compressed one. The only difference is that in a regular trie the substrings of all edges are one character long.

In our data structures, inserting a phrase S to $\mathcal{D}$ takes $O(|S|)$ time as in the case of a trie. Similarly, searching S takes $O(|S|)$ time if no information about substring S is provided. However, once it is known that S is in $\mathcal{D}$, searching strings obtained by concatenating or deleting a character to/from both ends of S takes only $O(1)$ time. More precisely, our data structures support two operations *extend* and *contract* in $O(1)$ time. Given a phrase S in $\mathcal{D}$, the operation extend(S, a) for a given character a, finds out whether the concatenation of S and a is a phrase in $\mathcal{D}$. Similarly, the operation contract(S), finds out whether the suffix $S[2 : |S|]$ is in $\mathcal{D}$. Notice that such operations can be performed in a suffix tree, if the phrases in $\mathcal{D}$ are all the suffixes of a given string as in the case of the LZ77 algorithm [RPE81]. For arbitrary dictionaries (such as the ones built by LZW) our data structures are unique in supporting contract and extend operations in $O(1)$ time, and insertion operation in time linear with the size of the phrase, while using $O(|\mathcal{D}|)$ space, where $|\mathcal{D}|$ is the number of phrases in $\mathcal{D}$.

Trie-reverse-trie-pair data structure. Our first data structure builds the trie, $\mathcal{T}$, of phrases as described above. In addition to $\mathcal{T}$, it also constructs $\mathcal{T}^r$, the compressed trie of the *reverses* of all phrases inserted in the $\mathcal{T}$. Given a string $S = s_1, s_2, \ldots, s_n$, its reverse S^r is the string $s_n, s_{n-1}, \ldots, s_2, s_1$. Therefore for each node v in $\mathcal{T}$, there is a corresponding node v^r in $\mathcal{T}^r$ which represents the reverse of the phrase represented by v. As in the case of the $\mathcal{T}$ alone, the insertion of a phrase S to this data structure takes $O(|S|)$ time. Given a dictionary phrase S, and the node u which represents S in $\mathcal{T}$, one can find out whether the substring obtained by concatenating S with any character a in is $\mathcal{D}$, by checking out if there is an edge from u with corresponding character a; hence extend operation takes $O(1)$ time. Similarly the contract operation takes $O(1)$ time by going from u to u', the node representing reverse of S in $\mathcal{T}^r$, and checking if the parent of u' represents $S[2:|S|]^r$.

Fingerprints based data structure. Our second data structure is based on building a hash table H of size p, a suitably large prime number. Given a phrase $S = S[1:|S|]$, its location in H is computed by the function h, where $h(S) = (s[1]|\Sigma|^{|S|} + s[2]|\Sigma|^{|S|-1} + \ldots + s[|S|])\ mod\ p$, where $s[i]$ denotes the lexicographic order of $S[i]$ in Σ [KR87]. Clearly, once the values of $|\Sigma|^k\ mod\ p$ are calculated for all k up to the maximum phrase size, computation of $h(S)$, takes $O(|S|)$ time. By taking p sufficiently large, one can decrease the probability of a collision on a hash value to some arbitrarily small ϵ value; thus the average running time of an insertion would be $O(|S|)$ as well. Given the hash value $h(S)$ of a string, the hash value of its extension by any character a can be calculated by $h(Sa) = (h(S)|\Sigma| + lex(a))\ mod\ p$, where $lex(a)$ is the lexicographic order of a in Σ. Similarly, the hash value of its suffix $S[2:|S|]$ can be calculated by $h(S[2:|S|]) = (h(S) - s[1]|\Sigma|^{|S|})\ mod\ p$. Both operations take $O(1)$ time.

In order to verify if the hash table entry $h(S)$ includes S in $O(1)$ time we (1) give unique labels to each of the phrases in $\mathcal{D}$, and (2) in each phrase S in H, store the label of the suffix $S[2:|S|]$ and the label of the prefix $S[1:|S|-1]$. The label of newly inserted phrase can be $|\mathcal{D}|$, the size of the dictionary. This enables both extend and contract operations to be performed in $O(1)$ expected time: suppose the hash value of a given string S is h_S, and the label of S is ℓ. To extend S with character a, we first compute from h_S, the hash value h_{Sa} of the string Sa. Among the phrases whose hash value is h_{Sa}, the one whose prefix label matches the label of S gives the result of the extend operation. To contract S, we first compute the hash value $h_{S'}$ of the string $S' = S[2:|S|]$. Among the phrases whose hash value is $h_{S'}$, the one whose label matches the suffix label of S gives the result of the extend operation. Therefore, both extend and contract operations take expected $O(1)$ time.

Inserting a phrase in this data structure can be performed as follows. An insert operation is done only after an extend operation on some phrase S (which is in $\mathcal{D}$) with some character a. Hence, when inserting the phrase Sa in $\mathcal{D}$ its prefix label is already known: the label of S. Once it is decided that Sa is going to be inserted, we can spend $O(|S|+1)$ time to compute the suffix label of Sa. In case the suffix $S[2:|S|]a$ is not a phrase in $\mathcal{D}$, we temporarily insert an entry for $S[2:|S|]a$ in the hash table. This entry is then filled up when $S[2:|S|]$ is actually inserted in $\mathcal{D}$. Clearly, the insertion operation for a phrase R and all its prefixes takes $O(|R|)$ expected time.

A linear time implementation of LZW-$\mathcal{FP}$. For any input T LZW-$\mathcal{FP}$ inserts to $\mathcal{D}$ the same phrases with LZW. The running time for insertion in both LZW and LZW-$\mathcal{FP}$ (via the data structures described above) are the same; hence the total time needed to insert all phrases in LZW-$\mathcal{FP}$ should be identical to that of LZW, which is linear with the input size. Parsing with $\mathcal{FP}$ consists of a series of extend and contract operations. We remind that: (1) the function f on characters of T is described as $f(i) = \ell$ where $T[i:\ell]$ is the longest substring starting at $T[i]$, which is in $\mathcal{D}$. (2) given b_m, the integer b_{m+1} is inductively defined as the integer α for which $f(\alpha)$ is the maximum

among all α such that $T[b_m : \alpha - 1]$ is in $\mathcal{D}$. In order to compute b_{m+1}, we inductively assume that $f(b_m)$ is already computed. Clearly $S = T[b_m : f(b_m)]$ is in $\mathcal{D}$ and $S' = T[b_m : f(b_m) + 1]$ is not in $\mathcal{D}$. We then contract S by i characters, until $S' = T[b_m + i : f(b_m) + 1]$ is in $\mathcal{D}$. Then we proceed with extensions to compute $f(b_m + i)$. After subsequent contract and extends we stop once $b_m + i > f(b_m)$. The last value of i at which we started our final round of contracts is the value b_{m+1}. Notice that each character in T participates to exactly one extend and one contract operation, each of which takes $O(1)$ time via the data structures described above. Hence the total running time for the algorithm is $O(n)$.

A linear time implementation of FPA. Parsing in FPA is done identical to LZW-$\mathcal{FP}$ and hence takes $O(n)$ time in total. The phrases inserted in $\mathcal{D}$ are of the form $T[b_m : f(b_m) + 1]$. Because in parsing step m, the phrase $T[b_m : f(b_m)]$ is already searched for, it takes only $O(1)$ time per phrase to extend it via our data structures. Hence the total running time for insertions is $O(n)$ as well.

Linear time implementations of decompression algorithms for LZW-$\mathcal{FP}$ and FPA. The decompression algorithms for both methods simply emulate their corresponding compression algorithms hence run in $O(n)$ time.

4. The Experiments

In this section we describe in detail the data sets we used, and discuss our test results testing how well our theoretical expectations were supported.

4.1. The test programs

We used `gzip`, `compress`, LZW-$\mathcal{FP}$ and FPA programs for our experiments. The `gzip` and `compress` programs are standard features of UNIX operating system. In our LZW-$\mathcal{FP}$ implementation we limited the dictionary size to 2^{16} phrases, and reset it when it was full as in the case of `compress`. We experimented with two versions of FPA, one whose dictionary was limited to 2^{16} phrases, and the other with 2^{24} phrases.

4.2. The data sets

Our data sets come from three sources: (1) Data obtained via UNIX `drand48()` pseudorandom number generator. (2) DNA and protein sequences provided by Center for BioInformatics, University of Pennsylvania and CT and MR scans provided by the St. Thomas Hospital, UK [Sou]. (3) Text files from two data compression benchmark suites: the new Canterbury corpus and the commonly used Calgary corpus [Sou].

The first data set was designed to test the theoretical convergence properties of the redundancy in the output of the algorithms and measure the constants involved. The second data set was designed to measure the performance of our algorithms for emerging bio-medical applications where no loss of information in data can be tolerated. Finally the third data set was chosen to demonstrate whether our algorithms are competitive with others in compressing text.

Specifically, the first data set includes three binary files generated by the UNIX `drand48()` function. The data distribution is i.i.d. with bit probabilities (1) $0.7 - 0.3$, (2) $0.9 - 0.1$, and (3) $0.97 - 0.03$. The second data set includes two sets of human DNA sequences from chromosome 23 (*dna1, dna2*), one MR (magnetic resonance) image of human (female) breast in uncompressed `pgm` format in ASCII (*mr.pgm*), and one CT (computerized tomography) scan of a fractured human hip

ct.pgm in uncompressed `pgm` format in ASCII [Sou]. The third set includes the complete Calgary corpus, which is the most popular benchmark suite for lossless compression. It includes a bibliography file (*bib*), two complete books (*book1, book2*), two binary files (*geo, pic*), source codes in `c, lisp, pascal` (*progc, progl, progp*), and the transcript of a login session (*trans*). The third set also also includes all files of size $> 1MB$ from the new Canterbury corpus: a DNA sequence from E-coli bacteria (*E.coli*), the complete bible (*bib.txt*) , and (*world192.txt*).

4.3. Test results

In summary, we observed that FPA implementation with maximum dictionary size 2^{24} performs the best on all types of files with size $> 1MB$ and shorter files with non-textual content. For shorter files consisting text, `gzip` performs the best. Also the theoretical expectations for the convergence rate in the redundancy of the output for i.i.d. data were consistent with the test results. We observed that the constants involved in the convergence rate for FPA and LZW-$\mathcal{FP}$ were smaller than that of LZW, and `gzip` was worse than all.

Our tests on the human DNA sequences with LZW-$\mathcal{FP}$ and FPA show similar improvements over *compress* and *gzip* - with a dictionary of maximum size 2^{16}, the improvement is about 1.5% and 5.7% respectively. Some more impressive results were obtained by increasing the dictionary size to 2^{24}, which further improved the compression ratio to 9%. The performance of LZW-$\mathcal{FP}$ and FPA on *mr* and *ct* scans differ quite a bit: LZW-$\mathcal{FP}$ was about $4\% - 6\%$ better than `compress` and was comparable to `gzip`; FPA's improvement was about 15% and 7% respectively. As the image files were rather short, we didn't observe any improvement by using a larger dictionary. One interesting observation is that the percentage improvement achieved by both FPA and LZW-$\mathcal{FP}$ increased consistently with increasing data size. This suggests that we can expect them to perform better in compressing massive archives as needed in many biomedical applications such as the human genome project.

Our tests on pseudorandom sequences were consistent our theoretical expectations: the asymptotic properties were observed even in strings of a few KB size. In general, all LZW based schemes performed better than `gzip`, which is based on LZ77. Our plots show that the redundancy in the output is indeed proportional to $1/\log n$ with the smallest constant achieved by FPA - in both cases, the constant is very close to 1.0; the constant for LZW-$\mathcal{FP}$ and LZW are about 1.5 and 2.0 respectively. This suggests that for on-line entropy measurement, FPA may provide a more reliable alternative to LZ78/LZW or LZ77 (see [FNS+95] for applications of LZW and LZ77 for entropy measurement in the context of DNA sequence analysis).

Our results on text strings varied depending on the type and size of the file compressed. For short files with long repetitions, `gzip` is still the champion. However, for all text files of size $> 1MB$, the large dictionary implementation of FPA scheme outperforms `gzip` by $4.7\% - 8.5\%$, similar to the tests for DNA sequences.

References

[BCW90] T. Bell, T. Cleary, and I. Witten. *Text Compression.* Academic Press, 1990.

[FM95] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression. In *ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[FNS+95] M. Farach, M. Noordeweir, S. Savari, L. Shepp, A. J. Wyner, and J. Ziv. The entropy of DNA: Algorithms and measurements based on memory and rapid convergence. In *ACM-SIAM Symposium on Discrete Algorithms*, 1995.

File	*Size*	*gzip*	*compress*	*LZW-FP*		*FPA*		*FPA-24*	
	(KB)	(KB)	(KB)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
bib	109	34	45	-26.01	5.04	-19.69	9.80	-19.69	9.80
book1	751	306	324	-3.81	2.03	-2.48	3.29	3.94	9.34
book2	597	202	245	-16.61	3.89	-12.32	7.42	-7.47	11.42
geo	100	67	76	-11.90	1.46	-11.66	1.67	-11.66	1.67
pic	501	55	61	-6.64	3.25	-5.31	4.47	-5.31	4.47
progc	39	13	19	-37.25	4.85	-33.40	7.52	-33.40	7.52
progl	70	16	26	-56.83	5.99	-49.29	10.51	-49.29	10.51
progp	48	11	19	-59.70	6.54	-53.75	10.02	-53.75	10.02
trans	91	18	37	-87.12	7.12	-73.96	13.65	-73.96	13.65

Table 1: Compression evaluation using files in the Calgary corpus.
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, and *FPA-24* over *gzip* and *compress*

File	*Size*	*gzip*	*compress*	*LZW-FP*		*FPA*		*FPA-24*	
	(KB)	(KB)	(KB)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
E.coli	4530	1310	1226	6.91	0.56	6.43	0.05	8.48	2.24
bible.txt	3953	1163	1369	-12.87	4.11	-7.79	8.42	4.68	19.01
world192.txt	2415	708	964	-31.70	3.32	-20.36	11.64	6.54	31.39

Table 2: Compression evaluation using files in the Canterbury corpus (Large Set)
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, and *FPA-24* over *gzip* and *compress*

[JS95] P. Jacquet and W. Szpankowski. Asymptotic behavior of the Lempel-Ziv parsing scheme and digital search trees. *Theoretical Computer Science*, (144):161–197, 1995.

[KM97] S. R. Kosaraju and G. Manzini. Some entropic bounds for Lempel-Ziv algorithms. In *Sequences*, 1997.

[KR87] R. Karp and M. O. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[LS95] G. Louchard and W. Szpankowski. Average profile and limiting distribution for a phrase size in the Lempel-Ziv parsing algorithm. *IEEE Transactions on Information Theory*, 41(2):478–488, March 1995.

[MS98] Y. Matias and S. C. Sahinalp. On optimality of parsing in dynamic dictionary based data compression. Unpublished Manuscript, 1998.

[MW85] V.S. Miller and M.N. Wegman. Variations on a theme by Lempel and Ziv. *Combinatorial Algorithms on Words*, pages 131–140, 1985.

[RPE81] M. Rodeh, V. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, January 1981.

[Sav97] S. Savari. Redundancy of the Lempel-Ziv incremental parsing rule. In *IEEE Data Compression Conference*, 1997.

[Sou] http://www.dcs.warwick.ac.uk/people/research/Nasir.Rajpoot/work/fp/index.html.

File	*Size* (*KB*)	*gzip* (*KB*)	*compress* (*KB*)	*LZW-FP*		*FPA*		*FPA-24*	
				$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
dna1	3096	978	938	5.59	1.54	5.75	1.70	8.91	5.00
dna2	2877	846	813	4.64	0.75	4.33	0.43	5.89	2.05
mr.pgm	260	26	29	-7.23	3.60	6.38	15.84	6.38	15.84
ct.pgm	1039	110	110	4.10	3.61	14.56	14.12	14.56	14.12

Table 3: Compression evaluation using experimental biological and medical data
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, and *FPA-24* over *gzip* and *compress*

File	*Size* (*KB*)	*gzip* (*B*)	*compress* (*B*)	*LZW-FP*		*FPA*		*FPA-24*	
				$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)	$\uparrow_g$ (%)	$\uparrow_c$ (%)
P(0)=0.7	1	212	217	2.83	5.07	2.83	5.07	2.83	5.07
P(1)=0.3	10	1644	1554	7.48	2.12	9.85	4.63	9.85	4.63
	100	15748	13558	15.23	1.54	17.58	4.27	17.58	4.27
	1024	**160011**	**132278**	**18.35**	**1.23**	**20.40**	**3.71**	**20.60**	**3.95**
P(0)=0.9	1	142	143	5.63	6.29	6.34	6.99	6.34	6.99
P(1)=0.1	10	1024	924	12.79	3.35	16.80	7.79	16.80	7.79
	100	9839	7781	23.11	2.78	26.06	6.50	26.06	6.50
	1024	**99853**	**74075**	**27.48**	**2.25**	**30.21**	**5.92**	**30.21**	**5.92**
P(0)=0.97	1	99	107	3.03	10.28	4.04	11.21	4.04	11.21
P(1)=0.03	10	508	503	5.91	4.97	10.24	9.34	10.24	9.34
	100	4625	3754	22.42	4.42	28.06	11.37	28.06	11.37
	1024	**45957**	**33420**	**29.76**	**3.41**	**34.64**	**10.12**	**34.64**	**10.12**

Table 4: Compression evaluation using independent identically distributed random files containing only zeros and ones with different probability distributions
The original file size (with some prefixes), compressed file size by *gzip* and *compress*, and the improvement (%) made by *LZW-FP*, *FPA*, and *FPA-24* over *gzip* and *compress*; random data generated by drand48().

[Sto88] J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[Wel84] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, pages 8–19, January 1984.

[Wyn95] A. J. Wyner. *String Matching Theorems and Applications to Data Compression and Statistics.* Ph.D. dissertation, Stanford University, Stanford, CA, 1995.

[Yok92] H. Yokoo. Improved variations relating the Ziv-Lempel and welch-type algorithms for sequential data compression. *IEEE Transactions on Information Theory*, 38(1):73–81, January 1992.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.

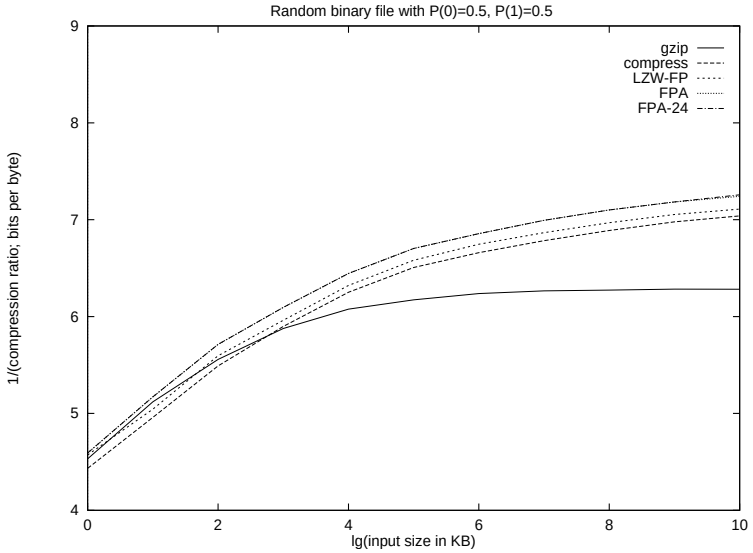


Figure 2: The compression ratios attained by all five programs on random i.i.d. data with bit probabilities $P(0) = P(1) = .5$.

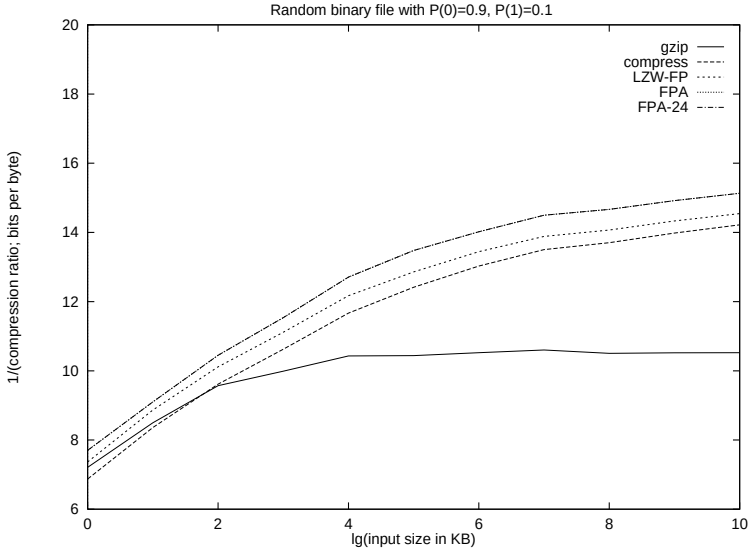


Figure 3: The compression ratios attained by all five programs on random i.i.d. data with bit probabilities $P(0) = .9$ and $P(1) = .1$.

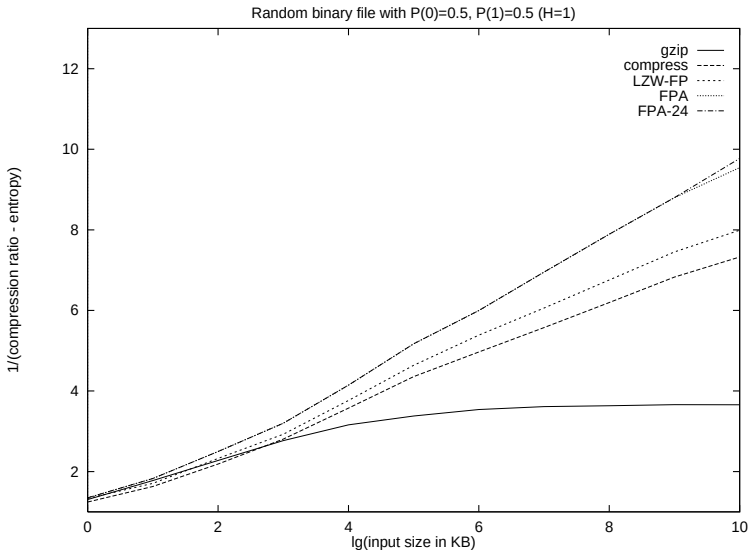


Figure 4: The 1/redundancy of all five programs on random i.i.d. data where redundancy is described as (actual compression ratio)-(bit-entropy). The bit probabilities are $P(0) = P(1) = .5$.

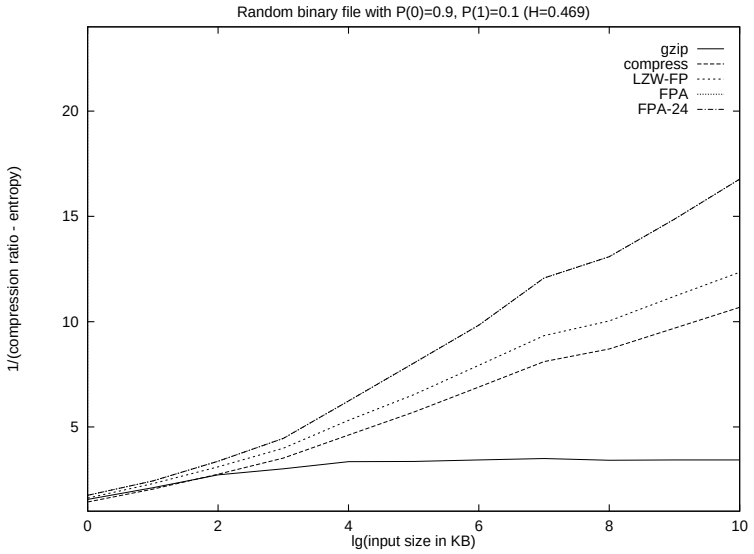


Figure 5: The 1/redundancy of all five programs on random i.i.d. data where redundancy is described as (actual compression ratio)-(bit-entropy). The bit probabilities are $P(0) = .9$ and $P(1) = .1$.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 62–73

Computing the width of a three-dimensional point set: an experimental study

Jörg Schwerdt, Michiel Smid
Department of Computer Science, University of Magdeburg
Magdeburg, D-39106 Magdeburg, Germany
e-mail: {schwerdt,michiel}@isg.cs.uni-magdeburg.de

Jayanth Majhi, Ravi Janardan
Department of Computer Science and Engineering, University of Minnesota
Minneapolis, MN 55455, U.S.A.
e-mail: {majhi,janardan}@cs.umn.edu

ABSTRACT

We describe a robust, exact, and efficient implementation of an algorithm that computes the width of a three-dimensional point set. The algorithm is based on efficient solutions to problems that are at the heart of computational geometry: three-dimensional convex hulls, point location in planar graphs, and computing intersections between line segments. The latter two problems have to be solved for planar graphs and segments on the unit sphere, rather than in the two-dimensional plane. The implementation is based on LEDA, and the geometric objects are represented using exact rational arithmetic.

1. Introduction

StereoLithography is a relatively new technology which is gaining importance in the manufacturing industry. (See e.g. the book by Jacobs [7].) The input to the StereoLithography process is a surface triangulation of a CAD model. The triangulated model is sliced by horizontal planes into layers, and then built layer by layer in the positive z-direction, as follows. The StereoLithography apparatus consists of a vat of photocurable liquid resin, a platform, and a laser. Initially, the platform is below the surface of the resin at a depth equal to the layer thickness. The laser traces out the contour of the first slice on the surface and then hatches the interior, which hardens to a depth equal to the layer thickness. In this way, the first layer is created, which rests on the platform. Next, the platform is lowered by the layer thickness and the just-vacated region is re-coated with resin. The next layers are then built in the same way.

An important step in this process is choosing an orientation for the model, i.e., the *build direction*. Among other things, the build direction affects the number of layers needed to build the model, a factor which impacts the time of the manufacturing process.

In our recent papers [8, 9], we have studied the problem of computing an optimal build direction for (combinations of) several design criteria from a theoretical point of view.

In this paper, we discuss an implementation and experimental results of an algorithm that computes all build directions that minimize the number of layers. This problem turns out to be equivalent to computing the width of a polyhedron [1, 6], and it leads to several problems that are at the heart of computational geometry: three-dimensional convex hulls, point location in planar graphs, and computing intersections between line segments. The latter two problems, however, have to be solved for graphs and segments (more precisely, great arcs) on the *unit sphere* rather than in the two-dimensional plane.

1.1. The width problem

Throughout this paper, $\mathcal{P}$ denotes a polyhedron, possibly with holes, and n denotes the number of its facets. We denote the unit sphere by $\mathbb{S}^2$. The *upper hemisphere* is defined as

$$\mathbb{S}^2_+ := \mathbb{S}^2 \cap \{(x, y, z) \in \mathbb{R}^3 : z \geq 0\}.$$

Similarly, we define the *lower hemisphere* as

$$\mathbb{S}^2_- := \mathbb{S}^2 \cap \{(x, y, z) \in \mathbb{R}^3 : z \leq 0\}.$$

Finally, the *equator* is the intersection of $\mathbb{S}^2$ with the plane $z = 0$. We will represent directions in $\mathbb{R}^3$ as points $\mathbf{d}$ on the unit sphere.

Often, layer thickness in the StereoLithography process is measured in thousandths of an inch. As a result, the number of layers needed to build a model can run into the thousands if the part is oriented along its longest dimension. If the layer thickness is fixed, then the number of layers for a given build direction $\mathbf{d}$ is proportional to the smallest distance between two parallel planes that are normal to $\mathbf{d}$, and which enclose $\mathcal{P}$. We call this smallest distance the *width* of $\mathcal{P}$ *in direction* $\mathbf{d}$, and denote it by $w(\mathbf{d})$. Note that $w(\mathbf{d}) = w(-\mathbf{d})$.

The *width* $W(\mathcal{P})$ of the polyhedron $\mathcal{P}$ is defined as the minimum distance between any two parallel planes that enclose $\mathcal{P}$, i.e.,

$$W(\mathcal{P}) = \min\{w(\mathbf{d}) : \mathbf{d} \in \mathbb{S}^2\}.$$

In this paper, we will consider the following problem: Given the polyhedron $\mathcal{P}$, compute all build directions $\mathbf{d}$ for which $w(\mathbf{d}) = W(\mathcal{P})$. Houle and Toussaint [6] gave an algorithm which solves this problem. We have implemented a variant of their algorithm. Our implementation uses LEDA [10], *exact arithmetic*, and efficient data structures. This implementation solves the problem exactly, and is *robust*, in the sense that it is correct even for a degenerate polyhedron (e.g., several neighboring facets can be co-planar). As far as we know, this is the first exact and robust implementation of an algorithm for computing the width of a three-dimensional point set.

Why do we use exact arithmetic? Our implementation uses data structures such as binary search trees, that are based on non-trivial ordering relations. The order of two objects is determined by making one or more orientation tests. Implementing these tests using exact arithmetic guarantees that our compare functions define "real" ordering relations, i.e., they are reflexive, anti-symmetric, and transitive. As a result, data structures whose correctness heavily depends on properties of an ordering relation can use these compare functions without having to worry about rounding errors.

The rest of this paper is organized as follows. In Section 2, we describe the algorithm. Section 3 discusses the implementation, especially the primitive operations where two objects on the unit sphere are compared. In Section 4, we present the results of our experiments on randomly generated point sets of size up to 100,000. We conclude in Section 5 with directions for future work.

2. The algorithm

The asymptotically fastest known algorithm for computing the width of a three-dimensional point set is due to Agarwal and Sharir [1]; its expected running time is roughly $O(n^{1.5})$. Our implementation is based on the algorithm of Houle and Toussaint [6], which has $O(n^2)$ running time in the worst case. The reason we implemented the latter algorithm is that (i) it is much simpler, (ii) in practice, the running time is much less than quadratic, as our experiments show (Tables 1 and 2), and (iii) it finds *all* directions that minimize the width. (Finding all optimal directions has applications when computing a build direction that minimizes a multi-criteria function, see [9].)

To compute the width of the polyhedron $\mathcal{P}$, we do the following. First, we compute the convex hull $CH(\mathcal{P})$ of (the vertices of) $\mathcal{P}$. It is clear that the set of directions that minimize the width of $\mathcal{P}$ is equal to the set of directions that minimize the width of $CH(\mathcal{P})$.

Let V be a vertex and F a facet of $CH(\mathcal{P})$. We call (V, F) an *antipodal vertex-facet pair* (or VF-pair), if the two parallel planes containing V and F, respectively, enclose $CH(\mathcal{P})$. We say that these parallel planes *support* $CH(\mathcal{P})$.

Similarly, two *non-parallel* edges e_0 and e_1 of $CH(\mathcal{P})$ are called an *antipodal edge-edge pair* (or EE-pair), if the two parallel planes containing e_0 and e_1, respectively, enclose $CH(\mathcal{P})$. Again, we say that these parallel planes *support* $CH(\mathcal{P})$.

In [6], it is shown that any direction minimizing the width of $\mathcal{P}$ is perpendicular to the parallel planes associated with some VF- or EE-pair. Therefore, we compute all VF- and EE-pairs, and for each of them compute the distance between the corresponding supporting parallel planes. The smallest distance found is the width $W(\mathcal{P})$ of the polyhedron $\mathcal{P}$.

We now describe how the VF- and EE-pairs can be computed. The *dual graph* G of $CH(\mathcal{P})$ is the planar graph on the unit sphere $\mathbb{S}^2$ that is defined as follows. The vertices of G are the facet outer unit normals of $CH(\mathcal{P})$, and two vertices are connected by an edge in G, if the corresponding facets of $CH(\mathcal{P})$ share an edge. Note that edges of this dual graph are great arcs on $\mathbb{S}^2$. Moreover, edges (resp. faces) of G are in one-to-one correspondence with edges (resp. vertices) of $CH(\mathcal{P})$.

We transform G into a planar graph G' on $\mathbb{S}^2$, by cutting all edges that cross the equator, and "adding" the equator to it. Hence, G' contains all vertices of G, and all edges of G that do not cross the equator. Additionally, each edge e of G that crosses the equator is represented in G' by two edges that are obtained by cutting e with the equator. Moreover, by following edges of G', we can completely walk around the equator. Note that edges of G that are on the equator are also edges in G'. (Adding the equator is not really necessary—in fact, in our implementation, we do not even add it. Adding the equator makes the description of the algorithm cleaner, because all faces of the graphs that are defined below are bounded by "real" edges.)

Let G'_u be the subgraph of G' containing all vertices and edges that are in the upper hemisphere $\mathbb{S}^2_+$. Let G^0_l be the subgraph of G' containing all vertices and edges that are in the lower hemisphere $\mathbb{S}^2_-$. (Hence, all edges and vertices of G' that are on the equator belong to both G'_u and G^0_l.) Finally, let G'_l be the mirror image of G^0_l, i.e., the graph obtained by mapping each vertex $\mathbf{v}$ in G^0_l to the vertex $-\mathbf{v}$. Note that both graphs G'_u and G'_l are in the upper hemisphere $\mathbb{S}^2_+$.

2.1. Computing VF-pairs

Consider a vertex V and a facet F of $CH(\mathcal{P})$ that form a VF-pair. Let f_V be the face of G that corresponds to V, and let $\mathbf{d}_F$ be the vertex of G that corresponds to F.

Case 1: $\mathbf{d}_F$ is on or above the equator. Then $\mathbf{d}_F$ is a vertex of G'_u. Let f^0_V be the face of G^0_l that is contained in f_V. (Face f_V is completely or partially contained in the lower hemisphere. If f_V was not cut when we transformed G into G', then $f^0_V = f_V$. Otherwise, f^0_V is that part of f_V that is in the lower hemisphere.) Let f'_V be the face of G'_l that corresponds to f^0_V. Since the (unique) parallel planes that contain V and F are supporting, vertex $\mathbf{d}_F$ of G'_u is contained in face f'_V of G'_l.

Case 2: $\mathbf{d}_F$ is strictly below the equator. Then $\mathbf{d}_F$ is a vertex of G^0_l, and $-\mathbf{d}_F$ is a vertex of G'_l. Let f'_V be the face of G'_u that is contained in f_V. Since the (unique) parallel planes that contain V and F are supporting, vertex $-\mathbf{d}_F$ of G'_l is contained in face f'_V of G'_u.

It follows that we can find all VF-pairs, by performing a point location query with each vertex of G'_u in the graph G'_l, and performing a point location query with each vertex of G'_l in the graph G'_u.

We consider some special cases for Case 1. First assume that $\mathbf{d}_F$ is strictly above the equator, and is in the interior of an edge of G'_l bounding f'_V. Let g be the other face of G'_l that has this edge on its boundary. Let W be the vertex of $CH(\mathcal{P})$ that corresponds to g. Then the distance between V and the plane through F is the same as the distance between W and the plane though F. Therefore, when locating vertex $\mathbf{d}_F$ in G'_l, it does not matter if we get V or W as answer.

Next assume that $\mathbf{d}_F$ is on the equator, and is in the interior of an edge bounding f'_V. Since G'_l is considered as a graph in the upper hemisphere, the face of G'_l containing $\mathbf{d}_F$ is uniquely defined.

Finally, consider the case when $\mathbf{d}_F$ coincides with a vertex $\mathbf{d}_V$ of f'_V. Let $g \neq f'_V$ be an arbitrary face of G'_l having $\mathbf{d}_V$ as a vertex on its boundary. Let W be the vertex of $CH(\mathcal{P})$ that corresponds to g. Then the distance between V and the plane through F is the same as the distance between W and the plane though F. Therefore, when locating vertex $\mathbf{d}_F$ in G'_l, it does not matter if we get V or W as answer.

2.2. Computing EE-pairs

Consider two edges e_0 and e_1 of $CH(\mathcal{P})$ that form an EE-pair. Recall that these edges are not parallel. Let g_0 and g_1 be the edges of G that correspond to e_0 and e_1, respectively. Then g_0 and g_1 are not both on the equator, and they can only have one point in common.

Assume w.l.o.g. that g_0 is (completely or partially) contained in the upper hemisphere. Then g_1 is (again, completely or partially) contained in the lower hemisphere. Let g'_0 be the part of g_0 that is contained in $\mathbb{S}^2_+$. Then g'_0 is an edge of G'_u. Let g^0_1 be the part of g_1 that is contained in $\mathbb{S}^2_-$, and let g'_1 be its mirror image. Then g'_1 is an edge of G'_l. Since the (unique) parallel planes that contain e_0 and e_1 are supporting, the edges g'_0 and g'_1 intersect.

We consider some special cases. Assume that one endpoint, say $\mathbf{d}_0$, of g'_0 coincides with one endpoint, say $\mathbf{d}_1$, of g'_1. Note that $\mathbf{d}_0 = \mathbf{d}_1$ is a vertex in both G'_u and G'_l. If it is also a vertex in G, then $\mathbf{d}_0$ and $\mathbf{d}_1$ correspond to two facets F_0 and F_1 of $CH(\mathcal{P})$. In this case, the distance between the planes containing F_0 and F_1 is equal to the distance between the parallel planes through e_0 and e_1. Let V be any vertex of F_1. Then we have found the direction $\mathbf{d}_0$ already because of the VF-pair (V, F_0). Hence, we do not have to worry about intersections of this type. So assume that $\mathbf{d}_0$ is not a vertex of G. Then it is on the equator. In this case, we have to find the intersection between g'_0 and g'_1.

Next consider the case when one endpoint, say $\mathbf{d}_0$ of g'_0, is in the interior of g'_1. Assume that $\mathbf{d}_0$ is a vertex of G. Let F be the facet of $CH(\mathcal{P})$ that corresponds to $\mathbf{d}_0$. Also, let V be one of the endpoints of edge e_1. Then the distance between V and the plane through F is equal to the distance between the parallel planes through e_0 and e_1. Hence, we have found this distance already because of the VF-pair (V, F). So assume that $\mathbf{d}_0$ is not a vertex of G. Then $\mathbf{d}_0$ is on the equator, and g'_1 is also on the equator. In this case the edge e_1 is orthogonal to the plane $z = 0$, and we have to find the intersection between g'_0 and g'_1.

Hence, we can find all necessary EE-pairs, by computing all edge pairs (g'_0, g'_1) such that (i) g'_0 is an edge of G'_u, (ii) g'_1 is an edge of G'_l, and (iii) g'_0 and g'_1 intersect in their interiors or the common point of g'_0 and g'_1 is on the equator.

2.3. The running time of the algorithm

Houle and Toussaint show in [6] that the entire algorithm can be implemented such that the worst-case running time is bounded by $O(n^2)$. Moreover, they show that the number of directions minimizing the width can be as large as $\Theta(n^2)$.

In order to get a better understanding of the performance of our implementation, we express the running time as a function of (i) the number n of facets of $\mathcal{P}$, (ii) the number h of facets of $CH(\mathcal{P})$, and (iii) the number k of intersections between edges of G'_u and edges of G'_l.

We compute the convex hull of $\mathcal{P}$ using LEDA, in $O(n \log n)$ time. The graphs G'_u and G'_l have total size $O(h)$. For each of these graphs, we build a point location data structure that is based on the slab method of Dobkin and Lipton [5]. (See also Section 2.2.2.1 in [12].) These data structures can be computed in $O(h^2)$ worst-case time, using a sweep algorithm. Note that this upper bound of $O(h^2)$ is tight only if a large number of edges cross a large number of slabs. Given the data structures, one point location query can be answered in $O(\log h)$ time. Hence, the total time for computing all VF-pairs is bounded by $O(h^2)$ in the worst-case.

We compute the intersections between edges of G'_u with edges of G'_l in $O(h \log h + k \log h)$ time, using a variant of the implementation of Bartuschka et al. [3] of the Bentley-Ottmann algorithm [4], adapted to great arcs on the unit sphere. This gives all EE-pairs.

The overall worst-case running time of our algorithm is thus bounded by

$$O(n \log n + h^2 + k \log h).$$

Remark 2.1. We have reduced the problem of computing the width to problems on the upper hemisphere. The reader may wonder why we do not use *central projection* [12] to map points and great arcs on $\mathbb{S}^2$ to points and line segments in the plane, respectively. The problem with this approach is that points on the equator are projected to points at infinity. As a result, we need compare functions that determine the order of different points at infinity.

3. The implementation

We now give some more details about our implementation. As mentioned already, it is based on LEDA and exact arithmetic. The program takes as input a set S of three-dimensional points (which are the vertices of the polyhedron $\mathcal{P}$). The coordinates of these points are represented using exact rational arithmetic (`d3_rat_point`) from LEDA.

First, we use the LEDA implementation of an incremental algorithm that computes the convex hull of the points of S. Given this convex hull, we compute an *implicit* representation of the graph G. Recall that each vertex $\mathbf{v}$ of G is a unit outer normal vector of a hull facet, and is a point on the unit sphere. This point can be computed from the cross product of three of the vertices of the hull facet. Instead of normalizing this point $\mathbf{v}$, we represent it as a non-zero vector having the same direction as the ray from the origin through $\mathbf{v}$. That is, this vector does not necessarily have length one. In this way, we avoid using expensive and inexact arithmetic operations such as square roots. Moreover, all our geometric primitives—which actually operate on unit vectors—can be implemented using these vectors.

The graph G is stored as a `planar_map` from LEDA. Given G, the graphs G'_u and G'_l can easily be computed. Again, the vertices of these two graphs are represented as vectors that do not necessarily have length one.

As explained before, we can now compute all width-minimizing directions, by (i) locating all vertices of G'_u in G'_l and vice versa, and (ii) computing intersections between edges of G'_u with edges of G'_l.

We do this by using the algorithms of Dobkin-Lipton, and Bentley-Ottmann. Both these algorithms are based on the *plane sweep paradigm*—in particular, they both use the same types of primitive operations. Because our objects are on the unit sphere, however, we have to adapt these algorithms.

In a plane sweep algorithm for two-dimensional objects, we solve the problem at hand by sweeping a vertical line from left to right over the scene. Sweeping on the upper hemisphere can be thought of as follows. Let the z-axis be vertical. Moreover, let the x-axis be in the horizontal plane $z = 0$, going from left to right. Finally, let the y-axis be in the plane $z = 0$, going from bottom to top. When sweeping on the upper hemisphere, we move a half-circle from the left part of the equator, along the upper hemisphere to the right part of the equator, while keeping the two endpoints of the half-circle on the y-axis. Since we represent points on $\mathbb{S}^2$ as non-zero vectors having arbitrary lengths, we can also regard this as rotating a half-plane around the y-axis. It suffices to implement two types of *compare functions*.

3.1. Comparing two points

We are given two non-zero vectors $\mathbf{u}$ and $\mathbf{v}$, which represent points on the unit sphere, and want to decide which vector is visited first when rotating a half-plane around the y-axis clockwise by 360 degrees, starting from the negative x-axis. Assume $\mathbf{u}$ and $\mathbf{v}$ are visited simultaneously, and let H be the corresponding half-plane. That is, H is the half-plane that contains the y-axis, and the vectors $\mathbf{u}$ and $\mathbf{v}$. Then the order of $\mathbf{u}$ and $\mathbf{v}$ is determined by rotating a ray in H—starting at the negative

y-axis—around the origin. Note that this is equivalent to rotating a half-plane that is orthogonal to H and that contains this ray.

To compute the *VF*- and *EE*-pairs, it suffices to be able to compare two vectors that are on or above the equator. For completeness, however, we define our compare function for any two non-zero vectors **u** and **v**.

Note that vectors that represent the directions $(0, -1, 0)$ and $(0, 1, 0)$ are always contained in the rotating half-plane. It is natural to define $(0, -1, 0)$ as the minimum of all directions, and $(0, 1, 0)$ as the maximum of all directions.

The basic tool used when comparing two vectors is an orientation test, i.e., deciding whether a point is to the left, on, or to the right of a three-dimensional plane. The complete code for the compare function can be found at the end of this paper.

We now briefly discuss this code. A non-zero vector **u** is given as an instance `u` of type `sphere_point`. This point has homogeneous coordinates `u.X()`, `u.Y()`, `u.Z()`, and `u.W()`, which are of LEDA-type `integer`. The value of `u.W()` is always positive.

Consider the two points `u` and `v`. First, it is tested if one of `u` and `v` is the minimal or maximal direction. If this is not the case, then we compute `sweep`, which is a plane of LEDA-type `d3_rat_plane` containing `u` and the y-axis. The normal vector `sweep.normal()` of this plane is "in the sweep direction".

Assume that `u.Z()` is positive. Using LEDA's orientation test `sweep.side_of(v)`, we find the position of `v` w.r.t. the plane `sweep`.

Case 1: `sweep.side_of(v)` is positive. Then `u` comes before `v` in the sweep process.

Case 2: `sweep.side_of(v)` is zero. Then `v` is contained in the plane `sweep`.

Assume that `v.Z()` is positive. We compute `sp`, which is a point of LEDA-type `d3_rat_point` having the same coordinates as `u`. Then, we compute `Nsweep`, which is the plane through the origin and `sp`, and that is orthogonal to the plane `sweep`. The result of the comparison follows from the position of `v` w.r.t. `Nsweep`.

Otherwise, `v.Z()` is less than or equal to zero. Since `v` is not the minimal or maximal direction, and `sweep` is not the plane $z = 0$, point `v` is below the plane $z = 0$, i.e., `v.Z()` is negative. Hence, `u` comes before `v` in the sweep process.

Case 3: `sweep.side_of(v)` is negative. In this case, the result of the comparison follows from the position of `v` w.r.t. the plane $z = 0$.

The cases when `u.Z()` is negative or zero are treated in a similar way.

3.2. Comparing two edges

Here, we are given two edges s_1 and s_2, which represent great arcs on the upper hemisphere. Each edge is specified by its two endpoints, which are given as instances of type `sphere_point`. Both these edges have at least one point in common with the sweep half-circle, and at least one of their endpoints is on the sweep half-circle. We want to determine the order of s_1 and s_2 along the sweep half-circle.

The implementation of this compare function is based on the corresponding function in [3]. It uses orientation tests, and the compare function of Section 3.1.

Having implemented these two compare functions, we can immediately use LEDA for building and querying the point location data structures. Since the implementation of Bartuschka et al. [3] works for line segments in the plane, we had to recode this implementation. Our implementation, however, closely follows that of [3], and it uses the above compare functions.

n	h	k	min	max	average
1,000	132	67	6.2	8.7	7.9
5,000	211	98	10.7	15.1	13.2
10,000	231	109	11.4	17.0	14.7
20,000	260	114	13.0	19.0	17.5
30,000	277	128	15.0	23.4	19.6
40,000	283	126	18.8	24.0	21.0
50,000	276	123	19.5	23.8	21.5
60,000	276	128	19.7	25.6	22.9
70,000	275	125	21.6	24.3	23.4
80,000	277	123	23.1	26.2	24.9
90,000	282	123	23.8	28.0	26.3
100,000	283	122	23.6	29.8	27.2

Table 1: *Performance of our implementation for points randomly chosen in a cube. For each value of n, we randomly generated ten point sets of size n. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. Although k could be $\Theta(h^2)$, this table shows that in practice, it is slightly less than $h/2$.* min, max, *and* average *denote the minimum, maximum, and average time in seconds, respectively.*

4. Experimental results

Since we only have a limited number of polyhedral models, we tested our implementation on point sets of size $n \in \{10^3, \ldots, 10^5\}$ on a SUN Ultra 1 (143 MHz, 128 MByte RAM).

In our first experiment, we used LEDA's point generator `random_points_in_cube` to generate the points from a uniform distribution in the cube $[-1000, 1000]^3$. For each value of n, we generated ten point sets. We measured the time after these points were generated. Table 1 shows the running time in seconds. Also, the average values of h (the number of facets of the convex hull), and k (the number of *EE*-pairs) are given. Note that for this distribution, the expected value of h is bounded by $O(\log^2 n)$, see Section 4.1 in [12].

Although the worst-case running time of the algorithm can be quadratic in the number n of points, our experimental results show that on random inputs, the algorithm is much faster. As we saw before, the actual worst-case performance is bounded by $O(n \log n + h^2 + k \log h)$. As we can see in Table 1, the value of h is much smaller than n. Also, the value of k—which could be as large as $\Theta(h^2)$—is in fact slightly less than $h/2$. The running times in Table 1, however, show that the running time is not proportional to $n \log n$: otherwise, doubling n should at least double the running time. This implies that the constant factors corresponding to the terms h^2 and $k \log h$ are large, and these two terms basically determine the running time in practice. This is not surprising, because our compare functions are fairly complex.

In our second experiment, we generated the points from a uniform distribution in the ball centered at the origin and having radius 1000, using LEDA's point generator `random_points_in_ball`. Again, for each value of n, we generated ten point sets, and measured the time after these points were generated. The results are given in Table 2. For this distribution, the expected value of h is bounded by $O(\sqrt{n})$, see Section 4.1 in [12]. In this case, the value of k is slightly larger than $h/2$. Again, the running time is not proportional to $n \log n$, but is determined by the terms h^2 and $k \log h$, which have large constants.

n	h	k	min	max	average
1,000	156	86	8.8	10.5	9.7
5,000	348	189	21.1	24.8	23.2
10,000	491	259	32.0	35.5	33.5
20,000	699	379	45.7	51.4	49.9
30,000	844	451	58.9	64.6	62.4
40,000	988	528	69.6	76.9	74.9
50,000	1102	596	80.6	92.2	85.6
60,000	1209	644	92.7	98.1	95.2
70,000	1310	693	98.9	107.6	104.2
80,000	1401	752	107.4	116.7	113.1
90,000	1495	803	117.5	129.5	123.0
100,000	1564	844	124.4	134.0	130.8

Table 2: *Performance of our implementation for points randomly chosen in a ball. For each value of n, we randomly generated ten point sets of size n. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. In this case, the value of k is slightly larger than $h/2$.* min, max, *and* average *denote the minimum, maximum, and average time in seconds, respectively.*

5. Concluding remarks

We have given a robust, exact and efficient implementation of an algorithm that solves an important problem in computational geometry. This problem has applications to rapid prototyping and motion planning.

Our current implementation solves point location queries using Dobkin and Lipton's data structure. Since in the worst-case, this data structure takes $\Theta(h^2)$ time and space to build, we plan to replace it by persistent search trees [13]. These can be built in $O(h \log h)$ time, need only $O(h)$ space, and can be used to solve point location queries in $O(\log h)$ time. Basically, persistent search trees give a space-economic implementation of the slab method. The algorithms that have to be implemented follow the same sweep principle as in our current implementation. Hence, we can use our primitive operations of Sections 3.1 and 3.2.

Determining a good build direction in rapid prototyping leads to several other problems for objects on the unit sphere. For example, in [9], we show how *spherical Voronoi diagrams* can be used to compute among all directions that minimize the width, a direction for which the so-called *stair-step error* is minimal. We plan to implement these Voronoi diagrams, again by implicitly representing points on $\mathbb{S}^2$ as vectors.

Our implementation of the algorithm for computing the intersections between the edges of G'_u and those of G'_l only works for edges that are great arcs. We plan to extend this such that it can handle arbitrary arcs on $\mathbb{S}^2$. For this, we need to design a more complex function that compares two edges. (See Andrade and Stolfi [2] for a first step in this direction.)

The ideas presented in this paper can be used to solve problems involving line segments in the plane that are possibly unbounded (e.g., point location queries in a Voronoi diagram): Using central projection, we map these segments to great arcs on the upper hemisphere. Then, we can apply our techniques for solving the problem at hand on the unit sphere.

In [11], Mehlhorn et al. argue that *program checking* should be used when implementing geometric algorithms. We leave open the problem of designing a fast algorithm that checks whether the output of a width-minimizing algorithm is correct.

Acknowledgements

This work was funded in part by a joint research grant by DAAD and by NSF. The work of JM and RJ was also supported in part by NSF grant CCR-9712226.

References

[1] P. K. Agarwal and M. Sharir. Efficient randomized algorithms for some geometric optimization problems. *Discrete Comput. Geom.*, 16:317–337, 1996.

[2] M. V. A. Andrade and J. Stolfi. Exact algorithms for circles on the sphere. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, 1998. 126–134.

[3] U. Bartuschka, K. Mehlhorn, and S. Näher. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. In *Proc. Workshop on Algorithm Engineering*, pages 124–135, Venice, Italy, 1997.

[4] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[5] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.

[6] M. E. Houle and G. T. Toussaint. Computing the width of a set. *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-10:761–765, 1988.

[7] P. F. Jacobs. *Rapid Prototyping & Manufacturing: Fundamentals of StereoLithography.* McGraw-Hill, New York, 1992.

[8] J. Majhi, R. Janardan, M. Smid, and P. Gupta. On some geometric optimization problems in layered manufacturing. In *Proc. 5th Workshop Algorithms Data Struct.*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 136–149. Springer-Verlag, 1997.

[9] J. Majhi, R. Janardan, M. Smid, and J. Schwerdt. Multi-criteria geometric optimization problems in layered manufacturing. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 19–28, 1998.

[10] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38:96–102, 1995.

[11] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking geometric programs or verification of geometric structures. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 159–165, 1996.

[12] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1988.

[13] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

```
int compare(const sphere_point& u, const sphere_point& v)
{
  if( u.X() == 0 && u.Z() == 0)
    {
      if( u.Y() < 0 )      // is u the minimal point?
        {
          if( v.X() == 0 && v.Y() < 0 && v.Z() == 0 )
            {
              return(0);  // u == v
            }
          else
            {
              return(-1);  // u < v
            }
        }
      else
        {  // u.Y() > 0   u is the maximal point
          if( v.X() == 0 && v.Y() > 0 && v.Z() == 0 )
            {
              return(0);  // u == v
            }
          else
            {
              return(1);  // u > v
            }
        }
    }
  if( v.X() == 0 && v.Z() == 0 )
    {
      if( v.Y() < 0 )     // is v the minimal point?
        {
          return(1);  // u > v
        }
      else
        {
          return(-1);  // u < v
        }
    }

  d3_rat_plane sweep( d3_rat_point(0,1,0,1),
                      d3_rat_point(0,-1,0,1),
                      u.rat_point() );

  if(u.Z() > 0)
    {
      if( sweep.side_of( v.rat_point() ) > 0 )
        {
          return(-1);  // u < v
        }
      else
        {
          if( sweep.side_of( v.rat_point() ) == 0 )
            {
              if( v.Z() > 0 )
```

```
            {
              d3_rat_point sp( u.X(), u.Y(), u.Z(), u.W() );
              d3_rat_plane Nsweep( sp + sweep.normal(),
                                   d3_rat_point(0,0,0,1),
                                   sp );
              return(Nsweep.side_of(v.rat_point()));
            }
          else
            {
              return(-1);  // u < v
            }
        }
      else  // sweep.side_of(v) < 0
        {
          if( v.Z() < 0 )
            {
              return(-1);  // u < v
            }
          else
            {
              return(1);  // u > v
            }
        }
    }
  }
else
  {
    if( u.Z() < 0 )
      {
        if( sweep.side_of( v.rat_point() ) < 0 )
          {
            return(1);  // u > v
          }
        else
          {
            if( sweep.side_of( v.rat_point() ) == 0 )
              {
                if( v.Z() < 0 )
                  {
                    d3_rat_point sp( u.X(), u.Y(), u.Z(), u.W() );
                    d3_rat_plane Nsweep( sp + sweep.normal(),
                                         d3_rat_point(0,0,0,1),
                                         sp );
                    return(Nsweep.side_of(v.rat_point()));
                  }
                else
                  {
                    return(1);  // u > v
                  }
              }
            else  // sweep.side_of(v) > 0
              {
                if( v.Z() < 0 )
                  {
                    return(-1);  // u < v
```

```
                }
              else
                {
                  return(1);  // u > v
                }
            }
        }
    }
  else  // u.Z() == 0
    {
      if( u.X() < 0 )
        {
          if( v.Z() == 0 && v.X() < 0 )
            {
              d3_rat_point sp(u.X(), u.Y(), u.Z(), u.W());
              d3_rat_plane Nsweep( sp + sweep.normal(),
                                   d3_rat_point(0,0,0,1),
                                   sp );
              return(Nsweep.side_of(v.rat_point()));
            }
          else
            {
              return(-1);  // u < v
            }
        }
      else
        {
          if( u.X() > 0 )
            {
              if(v.Z() < 0)
                {
                  return(-1);  // u < v
                }
              else
                {
                  if( v.Z() == 0 && v.X() > 0 )
                    {
                      d3_rat_point sp(u.X(), u.Y(), u.Z(), u.W());
                      d3_rat_plane Nsweep( sp + sweep.normal(),
                                           d3_rat_point(0,0,0,1),
                                           sp );
                      return(Nsweep.side_of(v.rat_point()));
                    }
                  else
                    {
                      return(1);  // u > v
                    }
                }
            }
        }
    }
}
```

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 74–85

Implementation and testing eavesdropper protocols using the DSP tool [1]

Kostas Hatzis
Computer Technology Institute, Kolokotroni 3
Patras, 26221, Greece
e-mail: hatzis@cti.gr

George Pentaris
Computer Technology Institute, Kolokotroni 3
Patras, 26221, Greece
e-mail: pentaris@cti.gr

Paul Spirakis
Computer Technology Institute, Kolokotroni 3
Patras, 26221, Greece
e-mail: spirakis@cti.gr

and

Vasilis Tampakas
Technological Educational Institute (TEI) of Patras, M. Alexandrou 1
Patras, Koukouli, 26334, Greece
e-mail: tampakas@cti.gr

ABSTRACT

The Distributed Systems Platform (DSP) is a software platform that has been designed for the implementation, simulation and testing of distributed protocols. It offers a set of subtools which permit the researcher and the protocol designer to work under a familiar graphical and algorithmic environment. In this work we use the DSP and study the pursuit evasion problem in distributed environments : Members of a team of guards (e.g. antivirus software) traverse the links of the network in pursuit of the fugitive (e.g. worm) which moves along the links of the graph without any other knowledge about the locations of the guards than whatever it can collect as it moves (i.e. the worm is oblivious to dynamic network behaviour). The fugitive's purpose is just to read local information at each node and to stay in the network as long as possible. When a guard meets a fugitive the fugitive is destroyed. We state the problem in detail, compinatorially characterize it and compare various solutions in [Spirakis et al. 95], [Spirakis, Tampakas 94], [Spirakis, Tampakas 98]. The use of the DSP tool gave us considerable input and permitted us to improve and extend the design of our protocols and algorithms, experimentally test them, validate their performance, and investigate the problem considering more practical (and thus more applied) variations.

[1] This work was partially supported by the EU ESPRIT LTR ALCOM-IT. (contract No. 20244).

1. Introduction

Generally, there is a considerable gap between the theoretical results of Distributed Computing and the implemented protocols, especially in the case of networks of thousands of nodes. On the other hand, well-designed tools would possibly offer to the researchers a more practical view of the existing problems in this area, and this, in turn, could give better (in the content of flexibility and efficiency) protocol design. Our work shows that a platform, suitably designed, can become a flexible tool for the researcher and offer a valuable help both in the verification and the extension of his theoretical results.

The Distributed Systems Platform (DSP) is a software tool developed during the sequel of ALCOM projects and took its current form as a platform during the ALCOM-IT project. It provides an integrated environment for the implementation, simulation and testing of distributed systems and protocols. The DSP offers an integrated graphical environment for the design and implementation of simulation experiments of various ranges. It can provide visualization (animation) for networks of restricted number of nodes, or support experiments with networks of hundreds or thousands of nodes. It provides a set of simple, algorithmic languages which can describe the topology and the behaviour of distributed systems and it can support the testing process (on-line simulation management, selective tracing and presentation of results) during the execution of specific and complex simulation scenarios. The DSP tool can support the hierarchical simulation of more than one type of protocols at the same execution. The latter is suggested in the case of pipelined protocols (the protocols of the upper level use the final output of the protocols of the lower case, e.g. leader election and counting protocols) or layered protocols (the protocols of the upper level call and use in every step the protocols of the lower case, e.g. synchronizers). Moreover, in its last version DSP supports the simulation of mobile protocols. The reader can find more about DSP in section 2 and in [DSP design 96], [DSP specs 96].

In this work we use DSP tool for the design, testing and verification of distributed protocols related with network security. Security of networks has triggered a number of fundamental studies in recent years. [Franklin et al. 93] considered the problem of maintaining privacy in a network that is threatened by mobile eavesdroppers, i.e., by an adversary that can move its bugging equipment within the system. Mobile adversaries in the context of secure computation were introduced in [Ostrovsky, Yung 91].

We also adopt the notion of a mobile "eavesdropper" which moves in the network without having available to it an instantaneous description of the whole network state. Our goal is to implement and simulate network protocols which result in the elimination of the mobile adversary. Our assumption is that the network links can also traversed by mobile guards (e.g., antivirus software), any of which will eliminate the bug if they are both at the same node at the same time. We consider the case of a single mobile bug. Note that, due to the mobility of the bug, the actual number of nodes being targeted can change over time (e.g., in each network round). Once at a node, the (non-disrupting) bug gets to learn *all incoming and outgoing messages and the memory contents.* The bug can flow with message traffic to neighbours only, during *one computation step.* It can not forge messages originating by guards (some form of electronic signature service is guaranteed by the network at a lower level).

The guards (mobile antivirus software) are assumed each to have their own *on-line* source of randomness (i.e. they can independently of each other draw random numbers) and are also assumed to be able to *erase* information. Thus, when a guard departs from a node, it makes sure that nothing remains in the node which would tell anybody later on (in particular the eavesdropper) that the particular guard was there. The code of each guard contains internal data structures and variables (in an object - like form) able to be used only by the guard itself. According to our model, if a guard is at a neighbour node to the current position of the fugitive and if, in addition, the guard decides to move to the position of the fugitive next, then the fugitive *is able to sense this* (by reading incoming messages at the node it currently occupies) and will escape in a direction different than that of the

incoming messages (*alerting guard*). Note that the eavesdropper does not know the current values of the local variables of any guard and it cannot, in particular, guess the random choices made by the guards or even read them. The network nodes are assumed to have distinct id's and the timing is synchronous. The interested reader can find more information about the problem and our theoretical results in [Spirakis, Tampakas 94], [Spirakis et al. 95], [Spirakis, Tampakas 98].

2. The Basic Protocol of the Guards

The main idea of the protocol is to partition the guards into two groups: the *waiting guards* and the *searchers*. The waiting guards are spread over the network by a distributed randomized protocol, occupy some (randomly selected) final positions and stay there. They act as *traps*. If the fugitive passes through any of these positions, it is eliminated. The searchers are also spread over the network by the same randomized protocol. Both searchers and waiting guards do not communicate among themselves, in order to prevent the bug from learning their "plans". This part of the protocol is called the *spreading protocol*.

Due to the randomized nature of the spreading protocol the fugitive cannot guess the final positions of the waiting guards even if it knows the protocol. Notice that the final positions of the waiting guards partition the network into statistically similar pieces. Since the fugitive may stay inside such a piece (or even oscillate between two nodes) forever, the searchers are used to counteract this.

Each searcher is doing a *random walk* on the graph. The collection of random walks will intercept any position, that the fugitive could keep, in short time (polynomial in the size of the network). The fugitive cannot guess the current positions of the searchers, due to the random, independent, choices that they make during their walks. (It can, however, sense if one or more neighbouring searchers have decided to move to its position next).

By trying to escape, the fugitive may either coincide at a node with another searcher, or fall into a node trapped by a waiting guard. In both cases, the fugitive will be eliminated. (Actually the fugitive may also be eliminated by having the searchers accidentally occupy all neighbours of the node that the fugitive is currently in and then move one searcher into that node. This is similar to the traditional graph searching paradigm but our analysis indicates that the formerly described cases have higher probability to happen earlier).

We give below the basic protocol.

Basic Protocol

A. Waiting guard

```
visited_nodes : array [1..n] of Boolean; /*init. all entries false */
i, x : 1..n;
procedure : move_guards();
begin      choose randomly an adjacent vertex x;
    move to x;
    visited_nodes[x]:= true;
    if (∃ i : visited_nodes[i]=false) then move_guards(); end;
move_guards();
repeat for ever
begin
    search memory for fugitive and eliminate it
end.
```

B. Searcher

```
visited_nodes : array [1..n] of Boolean; /*init. all entries false */
```

```
i, x : 1..n;
procedure : move_searchers();
begin      choose randomly an adjacent vertex x;
    move to x;
    visited_nodes[x]:= true;
    if (∃ i : visited_nodes[i]=false) then move_searchers(); end;
begin      move_searchers();
    repeat for ever (in each clock tick)
      begin    search memory for fugitive and eliminate it;
      choose randomly an adjacent vertex x;
      move to x;
      end.
end.
```

3. A brief description of the DSP tool

DSP is a software tool that provides an integrated environment for the simulation and testing of distributed protocols. The DSP aims in providing a simple, general model, which is well accepted by algorithm designers and seems to be a viable candidate for the role of a bridging model of distributed computation. The adoption of such a model can be expected to insulate software and hardware developments from one another and make possible both general purpose distributed systems and transportable software. The DSP model follows the principles proposed by the books of G. Tel ([Tel 94])and N. Lynch ([Lynch 96]) and aims in describing precisely and concisely the relevant aspects of a whole class of distributed computing systems. A distributed computation is considered to be a collection of discrete events, each event being an atomic change in the state of the whole system. This notion is captured by the definition of transmission systems. What makes such a system distributed is that each transmission is only influenced by, and only influences, part of the state, basically the local state of a single process. In DSP, each process is represented by a finite state machine with a local transmission table. Events affecting transmissions include the arrival of a message at a node, time-outs, node (and link) failures and mobile process movement. The power of the processes is not limited in any other way since they are allowed to have local memory and (unbounded) local registers. The general model adopted by DSP supports many interesting variations (all are tuneable by the user):

- Non determinism (by allowing the designer of a protocol to specify a time distribution for the processing steps of a node and/or the message transmission delays of links).
- General topologies of networks (node interconnections).
- Asynchronous, perfectly synchronous or limited asynchronous computations with clean semantics in each case.
- A clean and abstract failure model for nodes and links (permanent, temporary or malicious failures).
- Clear semantics for fairness in computations, safety and liveness.
- A simple message passing subsystem with clean robustness properties and user-defined message types.

The platform allows in addition the modelling of mobile processes, the calling of a DSP library protocol from another user protocol and user control of local (virtual) clocks. The DSP platform thus

differs from all existing "simulators" or "languages" of distributed systems because of its generality, algorithmic simplicity and clarity of semantics of its supported features. It aims in providing to the distributed algorithms designer a suitable environment of what a general distributed system "is expected to be".

The basic components of the platform include:

I. A set of algorithmic languages that allow the description of a distributed protocol and the specification of a distributed system:

1. Protocol description language. Using the protocol description language, the distributed protocol is specified as a set of processes residing on nodes (static processes) or moving throughout the network (mobile processes). These processes communicate by messages defined by the user or shared memory.

2. Topology description language. The distributed system is specified as a set of nodes connected with links. The user is able to define several types of nodes with different characteristics (e.g. computation step, size of message queues) and links (e.g. transmission delay, FIFO or non-FIFO link, link with conflicts or not) and complex structures constructed by these items such as rings, trees and complete graphs. The size of the specified networks is unlimited.

3. Initialization language. The user is able to define the initial settings of a simulation experiment(assign a static process on a node, place mobile processes on nodes, define the initialization time of each node) and call a protocol from the DSP library that will be executed first and will return the results (e.g. an elected leader) to the user protocol.

4. Actions language. The user is able to define external events (actions) applied to the system during the simulation (such as interrupts on nodes, node and link failures).

II. A discrete event simulator that simulates the execution of a specified distributed protocol on a specified distributed system. The simulator models each process as a communicating finite state automaton which changes its state triggered by simulator events. It supports synchronous, asynchronous and limited asynchronous timing, several types of faults for nodes and links.

III. A data base for distributed protocols that can also be used as a distributed protocol library. For each protocol, the data base keeps the protocol specification as described by the protocol specification language, information about the environment required for the protocol execution (e.g. timing, communication model, topology) and other related information (e.g. authors, abstract, publication date etc.).

IV. Graphical user interface. The user interface provides text editors for the languages described (protocol and topology specification, initialization and action definition). The topology specification is also supported by a graphical editor. The user interface also supports communication with the data base. The visualization of a simulation experiment is supported by an interactive graphical environment.

4. The implementation of the protocol under the DSP environment

Protocol specification

The implementation is based on the protocol description language of DSP. This language provides the ability to describe a protocol in an algorithmic form, similar to the one met in the literature for distributed systems. It includes usual statements met in programming languages like C and Pascal, and special structures for the description of distributed protocols (e.g. states, timers) and communication primitives (e.g. shared variables, messages). The statements of the language support the use of these structures during the specification of a distributed protocol in the areas of data modelling,

communication, queuing, process and resource management (e.g. send a message through a link, execute a function as an effect of a process state transition). Some of the language statements support the interface of the specified protocol with the DSP graphical environment (e.g. show a node or a link with different color). The BNF notation of this language is presented in [DSP: BNF semantics, 96] and a more detailed description is given in the user manual ([DSP manual 98]).

The basic object of the DSP protocol description language is the *process*. The processes can be static (residing permanently on a node) or mobile (moving throughout the network). The eavesdroppers protocol involves three types of mobile processes, namely *Waiting Guards*, *Searchers* and *Fugitives*. Each one of them is described separately in the protocol file as a distinct object. The implementation is presented in the Appendix (see Figure 10) and refers to the simpler form of the protocol (with a single spreading process) including the notion of the *alerting guards*.

The protocol starts with the *TITLE* definition. It is followed by the definition of the message *Searcher_Coming* which are used in order to simulate the behaviour of the *alerting* searcher. The *waiting_guard* is defined to be a *mobile* process. The basic structure of this process is the array *visited_nodes* keeping track of visited nodes (in order to know when the random walk is completed). The *INIT* procedure is executed during the initialization of the process and includes the initialization of the process variables by assigning them constant values, or values created by specific language statements. In this case all entries of the *visited_nodes* array are initialized to *false* and the process stores its identity in local memory.

The procedure *move_guards()* is used to select a random neighbour of the current node (the node that the waiting guard currently resides) and then move to that neighbour. The procedure *PROTOCOL* includes the core algorithm executed by the process. It is an event-driven procedure, which means that for every specific event that the process is expected to handle, a corresponding set of statements is executed. The *waiting_guard* process handles only the events *WAKE_UP* (which is automatically scheduled by the simulator at the beginning in order to initialize the process, causing the *waiting_ guard* to begin its random walk) and the *MOBILE_PROCESS_ARRIVAL* which occurs whenever the process arrives on a node or another process (a searcher or a fugitive) has arrived on its current node. The logical structure of the statements executed by the *waiting_guard* in this case is the following: If the arriving process is the waiting guard itself, it examines whether it has completed the random walk. If not, it continues the walk. If the arriving process is a fugitive, the specific fugitive is removed and the simulation ends.

The implementation of the *searcher* process is similar. The major difference is that the *searcher* continues its random walk infinitely. The searcher sends also a *Searcher_Coming* message to a node before it moves to that node, in order to simulate the notification of a fugitive for the arrival of an alerting searcher. The strategy followed by the fugitive is an infinite random walk. Note that in this case the notion of the alerting searchers has no effect, since the fugitive moves on each round regardless the arrival of a searcher on its host node.

Protocol initialization

The initialization language of DSP provides statements that assigns process types (as specified in the protocol) to nodes of the topology and create and place mobile processes on nodes. An instance of an initialization file used during a simulation can be found in the Appendix (see Figure 8). It is used to create two processes of type *searcher*, two processes of type *waiting_guard* and one process of type *fugitive*. All processes are initiated at arbitrary time instances (the fugitive is initiated at a time instance that the waiting guards are expected to have completed their random walks).

Topology generation

The topologies used for the experiments were generated using the topology description language of the DSP. This facility supports the generation of large topologies using both a randomized and/or a hierarchical approach. In the first case the user specifies the number of nodes and the existence probability of every edge of the graph, according to the $G_{n,p}$ model. Since such random graphs have proven to be very regular, the hierarchical approach was used for the generation of more irregular graphs. According to this approach, a graph is described as a collection of interconnected subgraphs.

An instance of a DSP topology file is presented in the Appendix (see Figure 9).

5. The steps towards a more efficient Protocol. Refinements and verification

The implementation of the algorithm allowed the collection of testing results using the DSP debugging facilities. A series of interesting observations verified the applied enhancements, improving the algorithm's efficiency.

5.1. Case 1: Fugitives moving in small cycles

If the mobile bug has fixed local memory, able to hold only (part of) the contents of any single node but not able to locally store global network information, then there is a possible optimal strategy for the fugitive: instead of selecting the next node randomly among the neighbours of the current node, the Fugitive could try to detect cycles of small length (say three or four) that do not contain nodes occupied by waiting guards and keep cycling on the detected cycle forever. This strategy requires a small amount of memory and prevents the Fugitive from running into a waiting guard.

In order to conduct the experiment, the fugitive protocol was modified as follows: The Fugitive has limited memory (the array *cycle*) of size 3. By using this memory the fugitive tries to discover safe (without waiting guards) "cycles" of length 3. Whenever a fugitive discovers such a cycle it stops the random walk and starts moving into the cycle infinitely (see Figure 1).

```
INT ...,cycle_pos,next_node; BOOLEAN do_cycle; cycle ARRAY [0,2] OF INT;

PROCEDURE shift_cycle();
  BEGIN cycle[2]=cycle[1]; cycle[1]=cycle[0]; END;

INIT();
  BEGIN
    FOR i=1 TO 3
      cycle[i]=-1;
    do_cycle=FALSE; cycle_pos=0; PUT_MY_ID TO my_id;
  END;

PROCEDURE move();
  BEGIN
    IF do_cycle==TRUE THEN
      BEGIN next_node=cycle[cycle_pos]; cycle_pos=(cycle_pos+1) MOD 3; END;
    ELSE
      BEGIN
        PUT_RANDOM_NEIGHBOUR TO x;
        IF x==cycle[0] THEN BEGIN       /* A cycle has been detected */
            do_cycle=TRUE; next_node=x; cycle_pos=(cycle_pos+1) MOD 3; END;
        ELSE       /* The next node does not form a cycle with the two previously visited nodes */
          BEGIN CALL shift_cycle(); cycle[0]=x; next_node=x; END;
        END;
      END;
    MOVE_TO next_node;
  END;
```

Figure 1: The modified fugitive protocol

Both versions of the protocol were run 100 times each on 5 different topologies of 100 nodes. In all cases the protocol was initialized with 5 Waiting Guard and 2 Searcher processes. All topologies were dense ($\sim$ 1000 links). The experimental results obtained are presented in Figure 2. The extermination time represents simulation rounds after the initialization of the fugitive process and does not include Waiting Guard spreading time. The results show that the cycling Fugitive has a fair advantage over the simple one. In all cases, the Fugitive managed to survive much longer, especially in more irregular topologies. The very large mean values in the topologies 3 and 5 were caused by runs where the Fugitive managed to survive very long.

In order to observe the effect of the number of Waiting Guards on the cycling Fugitive extermination time, the following experiment was conducted: Both versions of the protocol were run on the first of the 5 topologies for different number of Waiting Guards. The average extermination time of 100 runs for each case is presented in Figure 3.

Topology	Memoryless	Cycling
1	145.224	4.581.776
2	113.044	5.609.807
3	122.197	8.802.905
4	167.581	3.098.712
5	232.912	9.871.445

Figure 2: Average extermination time of 100 experiments for memoryless and cycling fugitive

#WGs	Memoryless	Cycling
2	283.391	7.559.827
8	97.463	6.895.322
12	86.833	5.328.830
16	45.938	6.146.287
20	37.782	4.681.611

Figure 3: Average extermination time of 100 experiments for different number of Waiting Guards

The results show that the Cycling Fugitive, unlike the memoryless one, is not affected seriously by the number of Waiting Guards. In fact, the only cases where a Fugitive was exterminated by a Waiting Guard, were the ones where the Fugitive run into a Waiting Guard before finding a triangle.

The only way for the guards to get over the fugitive strategy is to eliminate its advantage. This can be achieved by allowing the waiting guards to move repeatedly. In order to adopt this strategy without loosing the advantages obtained by being static, the notion of the "epoch" is introduced. After the spreading phase, the Waiting Guards remain idle for an epoch, namely a fixed or randomly chosen amount of steps. After this period, the Waiting Guard performs again a random walk and gets a new position. This procedure is repeated forever. By allowing the Waiting Guards to get new positions after the end of an epoch, the Fugitive can not rely on the knowledge of a Waiting Guard-free cycle anymore, since Waiting Guards may be placed on its way. On the other hand, the duration of each epoch would not allow the Waiting Guard to reside long enough on a node, thus remaining harmless for long periods of time, while a long epoch would omit the advantage of the Waiting Guards movement.

The implementation of the new strategy lead to the following changes to the protocol of the Waiting Guard process: The waiting guard process starts a timer (*epoch_timer*) whenever it completes a random walk. After *epoch_duration* rounds, the timer causes a *TIME_OUT* event and the waiting guard starts a new random walk (see Figure 4).

The results obtained for different epoch selections are presented in Figure 5. It is clear that regardless the epoch duration selection, the fugitive gets caught earlier. As expected, the average extermination time varies, depending on the epoch duration. It is interesting to observe that the obtained results suggest the existence of an optimal epoch duration, depending on the underlying graph. For this optimal epoch duration, the cycling fugitive extermination time is slightly longer, as compared to the one of the memoryless fugitive.

5.2. Case 2: Irregular graphs

A crucial issue of the protocol behaviour is the positioning of Waiting Guards. During the spreading phase, Waiting Guards perform a random walk in order to get a random position on G. The random walk is interrupted as soon as the Waiting Guard has visited all nodes at least once.

```
...CONST epoch_duration=200000; TIMER epoch_timer;
PROCEDURE init_random_walk();
  BEGIN
    FOR i=1 TO NODES
      visited_nodes[i]=FALSE;
  END;

INIT();
  BEGIN CALL init_random_walk(); PUT_MY_ID TO my_id; END;

PROTOCOL();
  BEGIN
    ON EVENT INITIALIZE DO CALL move_guards();
    ON EVENT TIME_OUT DO       /* An epoch has finished, start the new random walk */
      BEGIN CALL init_random_walk(); CALL move_guards(); END;
    ON EVENT MOBILE_PROCESS_ARRIVAL DO
      BEGIN
        id=GET_ID_OF_ARRIVING_PROCESS;
        IF id==my_id THEN
          BEGIN
            PUT_MY_NODE_ID TO host; visited_nodes[host]=TRUE;
            FOR i=1 TO NODES
              IF visited_nodes[i]==FALSE THEN
                CALL move_guards();
              ELSE START epoch_timer TIMEOUT epoch_duration; /* The random walk is completed, start the epoch_timer
                                                                to remain in the current position for an epoch */
          END;
        ELSE
          BEGIN
            type=GET_TYPE_OF_ARRIVING_PROCESS;
            IF type==fugitive THEN
              BEGIN DESTROY_MOBILE_PROCESS id; SIMULATION_END; END;
          END;
      END;
  END;
END;
```

Figure 4: Modified waiting guard protocol spreading in epochs

Topology	Epoch = 200.000	Epoch=2.000.000	Epoch = 6.000.000
1	1.304.327	284.796	833.484
2	1.784.585	346.658	943.221
3	2.164.738	183.563	1.531.374
4	1.947.302	528.993	1.374.664
5	3.317.633	834.479	4.265.494

Figure 5: Average extermination time of 100 experiments

However, our theoretical and experimental results show that the distribution of the Waiting Guards on the nodes is not uniform. In fact, depending on the regularity of the graph, Waiting Guards tend to reside on nodes that have low degree and are not "central" to G. The intuition behind this observation is that the Waiting Guards tend to remain for long periods on clusters with many connections, while nodes with low connectivity are visited rarely. Consider, for example, an extreme case, the line graph. In this case, a random walk that starts from a random node will reside with probability 1/2 on one of the end nodes. This weakness of the Waiting Guard spreading process could be used by the fugitive by adopting a strategy that prefers nodes with high degree and avoids possible dead ends. In order to observe the distribution of Waiting Guards, 10 different topologies of 1000 nodes with descending regularity were used. The topologies varied from a 50-regular graph (Topology 1) to a graph with many small clusters and 10 dead ends (Topology 10). For each topology, the spreading of 100 Waiting Guards was repeated for 100 times. The mean values over 100 runs of the variances of the distribution of Waiting Guards for the different topologies are presented in Figure 6.

As can be observed, the experiments verified that regular graphs show a more uniform distribution of the Waiting Guards. In most experiments on Topology 1, about 90% of the Waiting Guards occupied different nodes. There were only two cases where four Waiting Guards ended their random walk on the same node. For the most irregular topology, in all simulation at least 95% of the Waiting

Topology	Variance
1	0,12412
2	0,15229
3	0,15740
4	0,20020
5	0,27042
6	0,48511
7	0,84313
8	1,00501
9	1,66106
10	1,82583

Figure 6: Variance of number of Waiting Guards residing on each one of the 1000 nodes (mean value over 100 runs)

Guards ended their random walks on a dead end.

The previous observations lead to the following alteration of the Waiting guard strategy: Instead of interrupting the random walk as soon as all nodes are visited, the Waiting Guard chooses a number randomly from 1 to N and starts a random walk. The random walk is interrupted as soon as the Waiting Guard reaches the randomly chosen node. It is easy to observe that according to this strategy, the distribution of the final position of the Waiting Guards is uniform. Furthermore, the altered Waiting Guard requires less memory (since there is no need to keep track of visited nodes anymore) and leads to a simpler implementation.

The modifications to the waiting guard protocol are presented in Figure 7. The process chooses randomly a node whenever it starts a new random walk in a new epoch (procedure *init_random_walk*). Whenever it reaches this node the random walk is interrupted.

The previous experiment was repeated for the new spreading algorithm. As expected, the spreading of the Waiting Guards was uniform and the variances were small, ranging from $0,11$ to $0,13$ for all topologies.

```
 ...CONST epoch_duration=1000000;      VAR ... selected_node; TIMER epoch_timer;
PROCEDURE init_random_walk();
  BEGIN PUT_MY_ID TO my_id; PUT_RANDOM_BETWEEN 0 AND NODES-1 TO selected_node; END;

INIT();
  BEGIN CALL init_random_walk(); END;

PROTOCOL();
  BEGIN
    ON EVENT INITIALIZE DO CALL move_guards();
    ON EVENT TIME_OUT DO
      BEGIN CALL init_random_walk(); CALL move_guards(); END;
    ON EVENT MOBILE_PROCESS_ARRIVAL DO
       BEGIN
         id=GET_ID_OF_ARRIVING_PROCESS;
         IF id==my_id THEN
           BEGIN
             PUT_MY_NODE_ID TO host;
             IF host<>selected_node THEN
                CALL move_guards();
               ELSE START epoch_timer TIMEOUT epoch_duration; /* The target node has been reached, the random walk is finished */
           END;
         ELSE
           BEGIN
             type=GET_TYPE_OF_ARRIVING_PROCESS;
             IF type==fugitive THEN
               BEGIN DESTROY_MOBILE_PROCESS id; SIMULATION_END; END;
           END;
       END;
   END;
END;
```

Figure 7: Waiting guard protocol spreading in epochs with randomly selected destination

References

[DSP: BNF semantics, 96] *"The description of a distributed algorithm under the DSP tool: The BNF notation"*, ALCOM-IT Technical Report, 1996.

[DSP design 96] *"The design of the DSP tool"*, ALCOM-IT Technical Report, 1996.

[DSP manual 98] *"DSP: Programming manual"*, 1998.

[DSP specs 96] *"The specifications of the DSP tool*, ALCOM-IT Technical Report, 1996.

[Franklin et al. 93] M. Franklin, Z. Galil and M. Yung, ,*"Eavesdropping Games: A Graph-Theoretic Approach to Privacy in Distributed Systems"*, ACM FOCS 1993, 670–679.

[Lynch 96] Nancy Lynch, *"Distributed algorithms"*, Morgan Kaufmann Publishers, 1996.

[Ostrovsky, Yung 91] R. Ostrovsky and M. Yung, *"Robust Computation in the presence of mobile viruses"*, ACM PODC 1991, 51–59.

[Spirakis, Tampakas 94] P.Spirakis and B. Tampakas, *"Distributed Pursuit-Evasion: Some aspects of Privacy and Security in Distributed Computing"*, short paper, ACM PODC 94.

[Spirakis et al. 95] P.Spirakis, B. Tampakas and H. Antonopoulou, *"Distributed Protocols Against Mobile Eavesdroppers"*, 9th International Workshop on Distributed Algorithms (WDAG '95), France, September 1995, pp. 160-167.

[Spirakis, Tampakas 98] P.Spirakis, B. Tampakas , *"Efficient Distributed Protocols Against Mobile Eavesdroppers"*,CTI Technical Report, submitted, 1998.

[Tel 94] Gerard Tel, *"Introduction to distributed algorithms"*, Cambridge University Press, 1994.

Appendix

```
INITIALIZATION FILE "init file1" FOR PROTOCOL "distributed protocols against mobile eavesdroppers"
PUT MOBILE PROCESS waiting_guard ON NODE 0,14
PUT MOBILE PROCESS searcher ON NODE 7,8
PUT MOBILE PROCESS fugitive ON NODE 2
INIT MOBILE PROCESSES OF TYPE waiting_guard RANDOMLY FROM 5 TO 12
INIT MOBILE PROCESSES OF TYPE searcher RANDOMLY FROM 2 TO 40
INIT MOBILE PROCESSES OF TYPE fugitive RANDOMLY FROM 10000 TO 12000
```

Figure 8: An initialization file for a topology with 15 nodes

```
TOPOLOGY "Topology 1"

NODE node1
PROCESSING STEP 5
ENDNODE

LINK link1
TRANSMITION DELAY 5
BIDIRECTIONAL
ENDLINK

RANDOM GRAPH OF 60 NODES node1 AND LINKS link1 WITH EDGE_PROB 0.2 WITH IDS 0-59
RANDOM GRAPH OF 40 NODES node1 AND LINKS link1 WITH EDGE_PROB 0.1 WITH IDS 60-89
NODE 90-92 node1
LINK 60< - >90 link1
LINK 40< - >80 link1
LINK 90< - >91 link1
LINK 91< - >92 link1
```

Figure 9: A topology generated by the DSP topology description language

```
TITLE "Distributed protocols against mobile eavesdroppers";

MESSAGE Searcher_Coming END;

MOBILE PROCESS waiting_guard
 BEGIN
  VAR
   INT i,x,id,my_id,type,host; visited_nodes ARRAY [1,NODES] OF BOOLEAN;

  INIT();
   BEGIN
    FOR i=1 TO NODES
      visited_nodes[i]=FALSE;
    PUT_MY_ID TO my_id;
   END;

  PROCEDURE move_guards();
   BEGIN PUT_RANDOM_NEIGHBOUR TO x; MOVE_TO x; END;

  PROTOCOL();
   BEGIN
    ON EVENT INITIALIZE DO CALL move_guards();
    ON EVENT MOBILE_PROCESS_ARRIVAL DO
      BEGIN
       id=GET_ID_OF_ARRIVING_PROCESS;
       IF id==my_id THEN
         BEGIN
          PUT_MY_NODE_ID TO host; visited_nodes[host]=TRUE;
          FOR i=1 TO NODES
            IF visited_nodes[i]==FALSE THEN
              CALL move_guards();
         END;       /* If all nodes have been visited do nothing (the walk is completed), else move to next node */
       ELSE
         BEGIN
          type=GET_TYPE_OF_ARRIVING_PROCESS;
          IF type==fugitive THEN
            BEGIN DESTROY_MOBILE_PROCESS id; SIMULATION_END; END;
         END;
      END;
   END;
 END;

MOBILE PROCESS searcher
 BEGIN
  VAR
   INT i,x,id,my_id,type,host;

  INIT();
   BEGIN PUT_MY_ID TO my_id; END;

  PROCEDURE move_searchers();
   BEGIN PUT_RANDOM_NEIGHBOUR TO x; SEND_NEW_MESSAGE Searcher_Coming TO x; MOVE_TO x; END;

  PROTOCOL();
   BEGIN
    ON EVENT INITIALIZE DO CALL move_searchers();
    ON EVENT MOBILE_PROCESS_ARRIVAL DO
      BEGIN
       id=GET_ID_OF_ARRIVING_PROCESS;
       IF id==my_id THEN CALL move_searchers();
       ELSE
         BEGIN
          PUT_MY_NODE_ID TO host; type=GET_TYPE_OF_ARRIVING_PROCESS;
          IF type==fugitive THEN
            BEGIN DESTROY_MOBILE_PROCESS id; SIMULATION_END; END;
         END;
      END;
   END;
 END;

MOBILE PROCESS fugitive
 BEGIN
  VAR
   INT i,x,id,my_id,type,host;

  INIT();
   BEGIN PUT_MY_ID TO my_id; END;

  PROCEDURE move();
   BEGIN PUT_RANDOM_NEIGHBOUR TO x; MOVE_TO x; END;

  PROTOCOL();
   BEGIN
    ON EVENT INITIALIZE DO CALL move();
    ON EVENT MOBILE_PROCESS_ARRIVAL DO
      BEGIN
       id=GET_ID_OF_ARRIVING_PROCESS;
       IF id==my_id THEN CALL move();
       ELSE
         BEGIN
          PUT_MY_NODE_ID TO host; type=GET_TYPE_OF_ARRIVING_PROCESS;
          IF type==searcher THEN
            BEGIN DESTROY_MOBILE_PROCESS id; SIMULATION_END; END;
         END;
      END;
   END;
 END;
```

Figure 10: The implementation of the protocol in the DSP protocol description language

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 86–97

Implementing Weighted b-Matching Algorithms: Towards a Flexible Software Design[1]

Matthias Müller–Hannemann[2]
Department of Mathematics, Technische Universität Berlin, Straße des 17. Juni 136
D 10623 Berlin, Germany
e-mail: mhannema@math.tu-berlin.de

and

Alexander Schwartz[2]
Department of Mathematics, Technische Universität Berlin, Straße des 17. Juni 136
D 10623 Berlin, Germany
e-mail: schwartz@math.tu-berlin.de

ABSTRACT

We present a case study on the design of an implementation of a fundamental combinatorial optimization problem: weighted b-matching. Although this problem is well-understood in theory and efficient algorithms are known, only little experience with implementations is available. This study was motivated by the practical need for an efficient b-matching solver as a subroutine in our approach to a mesh refinement problem in computer-aided design (CAD). The intent of this paper is to demonstrate the importance of flexibility and adaptability in the design of complex algorithms, but also to discuss how such goals can be achieved for matching algorithms by the use of design patterns. Starting from the basis of Pulleyblank's blossom algorithm we explain how to exploit in different ways the flexibility of our software design which allows an incremental improvement of efficiency by exchanging subalgorithms and data structures.

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 *b-matching problem* is to find a maximum weight integral vector $x \in \mathbb{Z}^{\mathbb{E}}$ satisfying $\sum_{e=(v,w)} x_e \leq b_v$ for all $v \in V$. If, in addition, equality is required to hold in these inequalities for all nodes, the b-matching problem is called *perfect*.

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. There are excellent surveys on matching theory, see for example Gerards [11] or Pulleyblank [25].

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 [16]. 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 [1].

[1] The full version of the paper can be obtained from ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-591-1998.ps.Z

[2] 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.

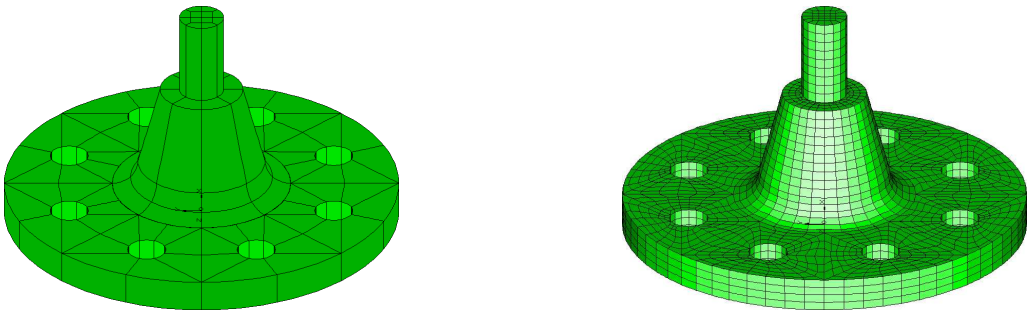


Figure 1: The model of a flange with a shaft and its refinement by our algorithm.

A somewhat surprising new application of weighted b-matching stems from quadrilateral mesh refinement in computer-aided design [19]. Given a surface description of some workpiece in three-dimensional space as a collection of polygons (for example, a model of a flange with a shaft, see Fig. 1), 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 $O(|V|)$) and change widely between nodes.

Both authors have been engaged in a research project together with partners in industry where this approach to mesh refinement has been developed. To the best of our knowledge, there is no publicly available code for weighted b-matching problems. Therefore, we first took the obvious formulation of weighted perfect b-matching problem as an integer linear program (ILP) and used CPLEX and its general purpose branch & and bound facilities to solve our instances. The corresponding experiences allowed us to conclude that the modeling of the mesh refinement problem as a weighted perfect b-matching problem captures the requirements of a high-quality mesh refinement very successfully [20]. The drawback, however, of such an ad-hoc solution was manifold: first, we encountered instances which could not be solved with CPLEX to optimality within reasonable time and space limits; second, we found the average running time too large for a convenient use in an interactive CAD-system, and third, our partners in industry prefer a software solution independent from commercial third-party products like CPLEX. Hence, the practical need to solve the mesh refinement problems in an efficient and robust way led us to work on our own implementation of weighted b-matching.

Design goals and main features of our implementation. As mentioned above, matching problems are well-understood in theory. Nevertheless the implementation of an algorithm for weighted b-matching is a real 'challenge'. An efficient and adaptable implementation requires a sophisticated software design. A recent systematic discussion of this design problem in the general context of graph algorithms can be found in [26]. In his paper, Weihe uses Dijkstra's algorithm as a running example to demonstrate what flexible adaptability of an algorithm component really means. Complications which arise in our context are due to the fact that we have to perform quite complex graph operations, namely shrinking and expanding of subgraphs, the famous "blossoms."

Design patterns capture elegant solutions to specific design problems in object-oriented software design which support reuse and flexibility. See the book of Gamma et al. [10] for an excellent introduction to design patterns. Our approach uses the design patterns *strategy, observer* and *iterator* which are well-known from [10] as well as *data accessor* and *adjacency iterator* introduced by Kühl & Weihe [13, 14]. The present paper serves as an empirical case study in the application of design principles.

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 all 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. We thereby got an efficient code which solves all instances from our mesh refinement application very well, but seems to be also fast on other classes of instances. Details can be found in an accompanying computational study [21].

Previous work. Most work on matching problems is based on the pioneering work of Edmonds [8]. "Blossom I" by Edmonds, Johnson, and Lockhart [9] was the first implementation for the *bidirected flow problem* (which is, as mentioned above, equivalent to the b-matching problem). Pulleyblank [24] 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 $O(|V||E|B)$ with $B = \sum_{v \in V} b_v$, and is therefore only pseudo-polynomial. The first polynomial bounded algorithm for b-matching has been obtained by Cunningham & Marsh [15] by scaling techniques.

Anstee [3] 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. The best strongly polynomial time bound for the (uncapacitated) Hitchcock transportation problem is $O((|V| \log |V|)(|E| + |V| \log |V|)$ by Orlin's excess scaling algorithm [22], and the second and third stage of Anstee's algorithm require at most $O(|V|^2|E|)$. Derigs & Metz [7] and Applegate & Cook [4] reported on the enormous savings using a fractional "jump start" of the blossom algorithm for weighted 1-matching. Miller & Pekny [17] 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 [23] 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. 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 [17] 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 [5] 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 [6]. In particular, the recent "Blossom IV" code of Cook & Rohe [6] seems to be the fastest available code for weighted 1-matching on very large scale instances. We believe that almost all previous approaches for 1-matching are not extendible to b-matching. One reason is that one usually exploits for efficiency reasons the fact that each node can have at most one incident matched edge. Some implementations also assume that edge costs are all non-negative. The mesh refinement application, however, uses arbitrary cost values. It seems that, in general, implementation studies focus on performance issues and do not address reuseability.

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 design goals in Section 3 and to outline our solution in Section 4 afterwards. Finally, in Section 5 we summarize the advantages and disadvantages of our approach.

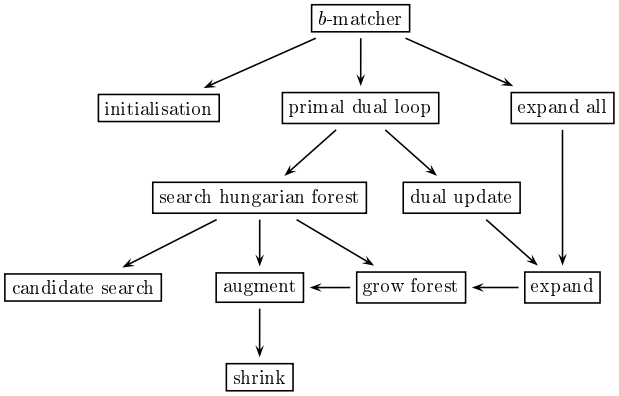


Figure 2: The structure of the blossom algorithm.

2. An Outline of Pulleyblank's Blossom Algorithm

In this section, we give a rough outline of the primal-dual algorithm of Pulleyblank [24]. The purpose of this sketch is only to give a basis for the design issues to be discussed later, and to point out some differences to the 1-matching case. A self-contained treatment is given in the full version of this paper. For an edge set $F \subseteq E$ and a vector $x \in \mathbb{N}^{|E|}$, 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 `maximize` $c^T x$ subject to (P1) $x(\delta(v)) = b_v$ for all $v \in V$, (P2) $x_e \geq 0$ for all $e \in E$, and (P3) $x(\gamma(S)) \leq q_S$ for all $S \in \Omega$.

The dual of this linear programming problem is `minimize` $y^T b + Y^T q$ subject to (D1) $y_u + y_v + Y(\Omega_\gamma(e)) \geq c_e$ for all $e = (u, v) \in E$, and (D2) $Y_S \geq 0$ for all $S \in \Omega$, 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: (CS1) for alle $\in E$, $x_e > 0$ implies $\bar{c}_e = 0$, and (CS2) for all $S \in \Omega$, $Y_S > 0$ implies $x(\gamma(S)) = q_S$.

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, $x \equiv 0$, $y_v := \frac{1}{2} \max\{c_e | e \in \delta(v)\}$ 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

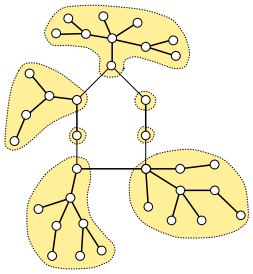


Figure 3: A sample blossom with an odd circuit of length seven. Each shaded region corresponds to a petal.

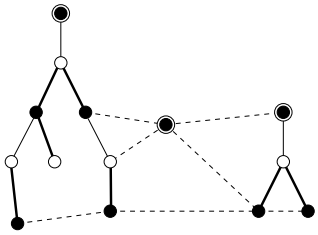


Figure 4: Example of an augmenting forest consisting of three trees. Even (odd) nodes are filled (non-filled), root nodes equipped with an extra circle, non-forest edges are dashed, matched (unmatched) edges are drawn with thick (thin) lines.

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 even nodes of different trees of F. 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 *pseudonodes*. 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, the so-called *petals*. The additional complication is that C must be the only circuit of the blossom. (See Figure 3 for a sample blossom.)

Hence, in order to detect such blossoms efficiently it is suitable to maintain additional invariants on the structure of the current non-perfect b-matching which are trivially fulfilled in the 1-matching case. Namely, each connected component M of matched edges in the surface graph contains no even circuit, at most one odd circuit and at most one deficit node. Even more, if such a component M contains an odd circuit, then M contains no deficit node.

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 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. See Figure 2 for an overview on the structure of the primal-dual algorithm.

3. Design Goals

Specialized to our application, the most important general requirements on the flexibility of a design imply that the implementation of the blossom algorithm framework should have the following features.

- **Exchangeable subalgorithms.** There are two good reasons:
  1. **Problem variants.** Suppose we apply our framework to a special case of b-matching, for example to ordinary 1-matching, to cardinality b-matching, or use additional edge capacities, in particular for *factor problems* where all edge capacities are set to one. For all such variants, the standard implementation of some subalgorithms (but only few!) should be exchangeable with an adapted version which is fine-tuned towards efficiency.
  2. **Algorithmic variants.** As software is often (in particular in our case) implemented first as a prototype, but later refined step-by-step to improve efficiency, the necessary modification should only affect small pieces of the code. In our example, we would like to test different dual update strategies, or exchange a forest with a single tree implementation, or apply heuristics to avoid the shrinking of blossoms.
- **Exchangeable data structures.** A basic design decision concerns the representation of graphs. The difficulty lies in the fact that the view on the graph objects changes throughout the algorithm: simultaneously, we have the original input graph, (moreover, in case of dense graphs it is useful to work on a sparse subgraph), then we have the *equality subgraph* (induced by edges of zero reduced costs), and finally the current *surface graph*, which is derived from the equality subgraph by blossom shrinking operations.
- **Evaluation strategies.** Certain mathematical functions and terms have to be evaluated so often during the execution of an algorithm, that different evaluation strategies may reduce the overall computational costs significantly. Well-known techniques such as "lazy evaluation" (calculate a value only when it is needed), "over-eager evaluation" (calculate a value before it is needed), and "caching" (store each calculated value as long as possible) can, for example, be applied to the evaluation of reduced costs, maintaining dual potentials or node deficits.
- **Separate initialization and preprocessing.** A blossom algorithm either starts with an empty matching, some greedy matching, or it uses a jump-start solution. In many applications, the size of an instance can be reduced in a preprocessing step, for example by a special handling of isolated or degree-1 nodes, parallel edges or more complicated special structures.
- **Exchangeable graph classes.** The framework has to be adaptable to special graph classes. The standard implementation does not assume anything about special properties of the graph classes. However, if it is known in advance, that one wants to solve matching problems on special graph classes, such as planar graphs, Eucledian graphs or complete graphs, it should be possible to exploit the additional structure of such a class.
- **Adding statistics.** We want to be able to get statistical information from the execution of our code. Operation counting [2] is a useful concept for testing algorithms, as it can help to identify asymptotic bottleneck operations in an algorithm, to estimate the algorithm's running time for different problem sizes, and to compare algorithmic variants. Furthermore, such statistics gives additional insight into the relationship of a class of instances and its level of difficulty for a blossom algorithm, for example by counting the number of detected blossoms or the maximum nesting level of blossoms.
- **Robust, self-checking.** A robust algorithm should (be able to) check all invariants and pre- and postconditions. It has to terminate with a deterministic behavior in case of a violation. In particular, each violation of one of these conditions that indicates an implementation bug is found immediately. This reduces the total time spend with debugging dramatically.

4. An Object-Oriented Implementation

In this section we will outline our solution with respect to the desired goals. This discussion does not exhaust all of our design goals, but will highlight those aspects which might be most interesting.

Decoupling algorithms from data structures: Iterators. The primal-dual algorithm uses different categories of graphs, namely the original graph, the equality subgraph and the surface graph. A closer look into the algorithm shows that we do not need to represent the equality graph explicitly. However, the internal representation of a graph where the node set remains static throughout the graph's lifetime is certainly different from a graph which must provide shrink and expand operations on its own subgraphs. Hence, we use two basic graph classes for the different cases (`unshrinkable_graph` and `surface_graph`). Below, we will give an example where the same algorithm is once used with with an instance of `unshrinkable_graph` and once with `surface_graph`. As shrinking of nodes can be nested, it is useful to have a map between an original node u and the corresponding pseudo-node or node in the surface graph, denoted by $\texttt{outer}(u)$.

An *iterator* provides a way to access the elements of an aggregate object sequentially. The underlying representation of the aggregate object remains hidden. Kühl & Weihe [13] applied this idea to graph algorithms. They introduced *adjacency iterators*. An *adjacency iterator* iterates over all edges and nodes which are adjacent to a fixed node. It provides operations for requesting if there is a current adjacent node, for requesting the current adjacent edge and the current adjacent node as well for constructing a new adjacency iterator which iterates over the adjacency of the current adjacent node.

In our context, we want to hide the concrete representation of our surface graph. For example, the client of an instance of a `surface_graph` should not know whether the adjacency list of a pseudo-node is built explicitly as a list or if it is only implicitly available by an iteration through the contained nodes.

In general, our adjacency iterators are implemented as *skip iterators* which run through the whole adjacency of a node, decide for each edge whether it is "present" in the current graph or not, and show an edge only in the affirmative case but skip it otherwise. The decision whether an edge is present or not is based on an evaluation of the predicate (reduced costs $== 0$) or ($\texttt{outer}(u) \neq \texttt{outer}(v)$) for an edge $e = (u, v)$. This means that we have different adjacency iterators for each specific view of a node onto its adjacency.

Recall that the surface graph contains two different types of nodes, namely original nodes and pseudo-nodes. This implies that one needs two different types of adjacency iterators. To be more precise, we use a pair of adjacency iterators, one for pseudo-nodes and one for original nodes. For each node of the surface graph, only one of them is valid. This pair of iterators is encapsulated in such a way that the client sees only a single iterator.

Data accessors. The *data accessor* pattern, introduced by Kühl & Weihe [14], provides a solution for the design problem to encapsulate an attribute of an object or a mathematical function and to hide the real representation or computation from the client.

There are several applications of this pattern in our context where it appears to be useful to hide the underlying representation of the data. A first example is the treatment of the deficit of a node. Possible choices are to store and to update the node deficit explicitly, or to calculate it when it is needed from the current matching x and the node capacity b_v. A second example concerns the maintenance of the cost of an edge if the edge costs are induced by some metric distance function between coordinates of its endpoints. Here, it might be useful to calculate edge costs only on demand.

Finally, it is a good idea to encapsulate the calculation of reduced costs. One reason is that there are several linear programming descriptions of the b-matching problem which can be used as alternatives in our algorithm. For simplicity, we only presented the blossom description, but the so-called *odd-cut description* [5, 6] can be used with minor changes. One concrete difference lies in

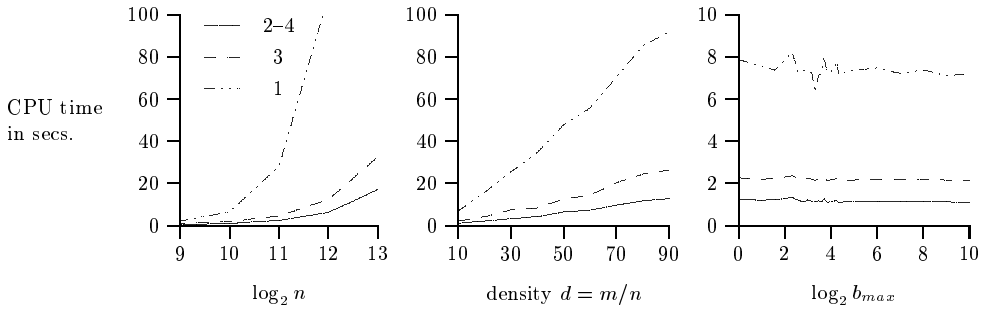


Figure 5: Experimental results for different strategies within the candidate search (strategy 1, strategy 3, and strategy 2 combined with strategy 3 and 4) for a test-suite on graphs with an Euclidean distance function. The charts show the run-time dependency of these strategies in seconds of CPU time for an increasing number of nodes (left), increasing density (middle), and increasing node capacities (right). Each data point is the average over runs on twenty independent instances.

the calculation of the reduced costs. Hence, in order to evaluate which description is superior to the other, one would like to exchange silently the internal calculation of the reduced costs. (For a replacement of the linear description a second small change is necessary in the dual update.)

Exchange of subalgorithms. The *strategy pattern* encapsulates each subalgorithm and defines a consistent interface for a set of subalgorithms such that an algorithm can vary its subalgorithms independently from the user of the algorithm. To apply the strategy pattern rigorously for subalgorithms of the framework it is helpful to implement each algorithm (and subalgorithm) as a separate algorithmic class. Thus, Fig. 2 which gives an overview on the structure of the primal-dual algorithm also represents our algorithmic classes. (A number of lower level classes are omitted for brevity).

We elaborate on the use of different strategies taking the example of the candidate search for edges in the tree growing part of the algorithm. (It should be noted that such candidates can effectively be determined within the dual update.)

1. 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.
2. Ball & Derigs [5] proposed to keep a partition of the edge set into subsets according to the labels of endpoints. This gives some overhead to update these lists, but avoids to examine all those edges for which one endpoint is labeled `even` and the other is labeled `odd`.
3. A refinement of both previous strategies 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. An application of the *observer pattern* (which we describe below) ensures that any change in the state of a node label triggers an appropriate update, i. e. nodes become unsafe and will be considered in the candidate search.
4. As shrinking and expanding of blossoms is computational very expensive it is useful to avoid shrinking operations heuristically. Each time a blossom forming edge has been detected, we do not shrink the blossom but store the edge instead. Only if no other candidate edges are left, we request the first blossom forming edge and shrink the corresponding blossom.
5. It is a fundamental strategic question whether one should perform the growing of trees simultaneously in a forest (as we did in our description) or to grow only a single tree. Whereas a forest version leads to shorter augmenting paths, a single tree version has the advantage of a reduced overhead.

Note that some of these strategies can also be used in combination. Figure 5 shows the impact of the different strategies on the run-time for a test-suite of b-matching problems on Euclidean graphs.

Exchange of data structures. We have already discussed the strategic question whether one should keep a partition of the edges according to the labels of their endpoints in the current forest or not. Computational results strongly indicated that explicitly maintaining such a edge partition is worth doing. But it is not at all clear which data structure to keep these edge sets is most efficient. Should we use simply a doubly-linked list structure which allows cheap insertion and deletion operations in $O(1)$ per edge but requires linear time to find the edges with minimum reduced costs in the update step? Or is a priority queue like a d-heap or some Fibonacci heap the better choice because of the $O(1)$ time to perform the minimum operation at the expense of more expensive insert and delete operations? Note that we usually have to perform much more insert/delete operations than minimum operations. Hence, an answer to these questions can only be given by computational testing. The important point from the software engineering perspective is that exchanging the data structures is an easy task.

Reusability of algorithms. We give one concrete example for a reuse of the blossom framework, namely fractional b-matching. Recall that fractional b-matching is the relaxation of b-matching which drops the integrality constraints on the matching x.

The adaption of our implementation to fractional b-matching becomes extremely easy. The only necessary modification is to exchange the data type of the matching x from `integer` to `double`, for example. (It is easy to see that one can keep the fractional matching half-integral throughout the algorithm). This change suffices because the calculation of the maximal value by which we can augment after some forest growing step now returns a value of $\frac{1}{2}$ in those cases where the integral version would round down to an integer and therefore return a zero which invokes a shrinking step afterwards. As shrinking is not necessary in a fractional algorithm, it is suitable for reasons of efficiency to start the algorithm with an instance of the graph class `unshrinkable_graph` instead of using the graph class `surface_graph` as the latter requires extra overhead for handling the `outer` information. Moreover, as the `shrinker` is never called in a normal execution of the algorithm it can be replaced by a `dummy_shrinker` which does nothing but returns an exception if it is called because this indicates an implementation bug.

Recall that fractional b-matching can be transformed into a Hitchcock transportation problem and therefore, in principle, be solved by any implementation for minimum cost flow problems, in particular by the network simplex. However, if we want to use the solution of the fractional matching problem as an improved basis for the integral matching algorithm, there is one pitfall. The problem is that if we use an algorithm for fractional matching as a black box, this algorithm certainly does not know that the input of the integral matching algorithm requires additional structural properties of the b-matching as preconditions. As a consequence, it is necessary to implement additional conversion algorithms which transform an integral matching obtained from rounding an optimal fractional b-matching into a starting matching fulfilling the requirements of Pulleyblank's algorithm. (This subtle detail is ignored in Anstee's paper [3].) We put emphasis on this point as the fractional algorithm obtained as an adaption of the integral one gives us the desired structural properties of the b-matching almost for free.

Initialization. Pulleyblank's algorithm can be decomposed into an initialization phase, the primal-dual loop, and a final expanding phase. As there are many different possibilities for a concrete initialization it is useful to separate these parts strictly form each other.

The benefit from an exchange in the initialization phase can be dramatic. The "jump start" with a fractional matching solver is one example which we discussed earlier. Strengthening of the initial dual solution such that for each node at least one adjacent edge lies in the initial equality subgraph also proved to be useful. Similar experiences have been reported by Miller & Pekny [17].

For cardinality matching problems, the first author's experiments with several greedy starting heuristics showed that it is often possible to do almost all work in the initialization phase. In fact, our heuristics have been so powerful that the loop kernel of the algorithm often only works as a checker for optimality in non-perfect maximum cardinality problems [18].

Observer pattern. The *observer pattern* defines a dependency between an observer class and an observing class such that whenever the state of the observed object changes, the observing class is notified about the change. We have already discussed one nice application of the observer pattern as a prerequisite of an advanced strategy for the candidate search.

In addition, observers allow us to get additional insights into the course of the algorithm by collecting data on function calls. Profilers such as *gprof* or *quantify* of Rational Software Corporation, could be used to count the number of function calls as well as to measure the time spent inside the functions. However, this gives only the overall sum of calls to a certain function and requires that the data we are interested in can be expressed in the number of function calls. Beyond mere operation counting observers can deliver much more detailed information. For example, we can determine the maximum nesting level of blossoms. This parameter is no operation and therefore not available to profilers, but is a valuable indicator for the hardness to solve some problem instance. For example, the nesting level is observed to be much lower in randomly generated instances than in structured instances.

Moreover, we may want to know how certain quantities change over time, in particular, we want to sample data from every iteration of the primal-dual loop (here we use also another pattern, the *loop kernel pattern* [12]). For example, we collect a series of data from each dual update to find out which updates are most expensive. This can even be used to control the selected strategy online. It has been observed [6] that the final ten augmentations usually require most of the overall computation time. Hence, if we can recognize with the help of an observer that the algorithm slows down it might be advisable to change the strategy. Cook & Rohe propose to switch from a single tree growing strategy to a forest growing strategy.

5. Summary and Outlook

We presented a case study oriented to weighted b-matching with emphasis on design problems. Our approach followed proposals of Weihe and co-workers to apply design patterns like graph iterators and data accessors in order to achieve a flexible design. The examples given in the previous section proved that we successfully realized flexibility with respect to several modifications.

Flexibility has its price. Applying all the desired design patterns requires excellent expertise in advanced programming techniques, at least to a much higher degree than traditional concepts. Hence, ease of use may be a critical issue.

We decided to take C++ as the programming language for our implementation, in particular, we heavily used templates (static polymorphism). Today, the main disadvantage of templates is that this feature is not fully supported by most compilers. In principle, compilers should be able to handle templated code as well as code without templates, and to optimize away the additional overhead imposed by encapsulation. However, the current compiler technology of *gcc/g++* of the Free Software Foundation Inc., version 2.8.1, as well as its offshoot *egcs*, version 1.0.2, does not seem to achieve this satisfactorily. Therefore, it is quite remarkable, that the current version of our code is already significantly faster than the code of Miller & Pekny [17], see Figure 6. This was definitely not true for the first prototype of our framework. However, through experiments we have been able to identify the bottlenecks of our implementation, and by exchanging subalgorithms and data structures the speed-up was made possible by the flexibility of our framework. And we believe that there is still potential for further improvements of efficiency. For an in-depth discussion of our computational results we refer to [21].

The fact that our code is already superior to the only b-matching executable available for a

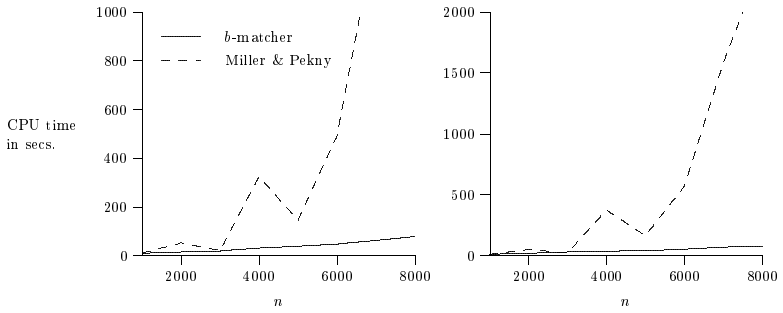


Figure 6: A comparison of the code from Miller & Pekny with our b-matcher on a test-suite of graphs with an Eucledian distance function, a density $d = m/n$ of 20 (left) and 30 (right), and b_v varying between 1 and 10.

comparison (of Miller & Pekny) encourages hopes that the design concepts is suitable for high performance computations. At least, we got an implementation which is mature enough to solve even the hardest instances of the mesh refinement application in less than 13 seconds for a sparse graph with more than 17000 nodes on a SUN UltraSPARC2 with 200 MHz running under Solaris 2.6. The solution for associated b-matching problem to the example shown in Figure 1 took only 3 seconds. Future work will show whether the flexibility also pays off for further specializations or extensions of the b-matching problem.

Acknowledgments

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

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

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

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

[4] 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.

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

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

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

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

[9] 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.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.

[11] A. M. H. Gerards, *Matching*, (M. O. Ball et al., ed.), Handbooks in Operations Research and Management Science, vol. 7, North-Holland, 1995, pp. 135–224.

[12] D. Kühl, M. Nissen, and K. Weihe, *Efficient, adaptable implementations of graph algorithms*, Workshop on Algorithm Engineering, 1997, http://www.dsi.unive.it/~wae97/proceedings.

[13] D. Kühl and K. Weihe, *Iterators and handles for nodes and edges in graphs*, Konstanzer Schriften in Mathematik und Informatik Nr. 15, Universität Konstanz, 1996, http://www.informatik.uni-konstanz.de/~weihe/manuscripts.html#paper24.

[14] D. Kühl and K. Weihe, *Data access templates*, C++–Report **9** (1997), no. 7, pp. 15 and 18–21.

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

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

[17] 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.

[18] R. H. Möhring and M. Müller-Hannemann, *Cardinality matching: Heuristic search for augmenting paths*, Technical report No. 439/1995, Fachbereich Mathematik, Technische Universität Berlin, 1995, ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-439-1995.ps.Z.

[19] 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.

[20] M. Müller-Hannemann, *High quality quadrilateral surface meshing without template restrictions: A new approach based on network flow techniques*, Proceedings of the 6th International Meshing Roundtable, Park City, Utah, Sandia National Laboratories, Albuquerque, USA, 1997, pp. 293–307.

[21] M. Müller-Hannemann and A. Schwartz, *Implementing weighted b-matching algorithms: Insights from a computational study*, in preparation, Fachbereich Mathematik, Technische Universität Berlin, 1998.

[22] 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.

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

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

[25] W. R. Pulleyblank, *Matchings and extensions*, (R. L. Graham, M. Grötschel, and L. Lovász, eds.), Handbook of Combinatorics, vol. 1, North-Holland, 1995, pp. 179–232.

[26] K. Weihe, *A software engineering perspective on algorithms*, Konstanzer Schriften in Mathematik und Informatik Nr. 50, Universität Konstanz, 1998, ftp://ftp.informatik.uni-konstanz.de/pub/preprints/1998/preprint-050.ps.Z.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 98–109

Matrix Multiplication: A Case Study of Algorithm Engineering

Nadav Eiron
Computer Science Department, Technion — Israel Institute of Technology
Haifa, 32000, Israel
e-mail: nadav@cs.technion.ac.il

Michael Rodeh
Computer Science Department, Technion — Israel Institute of Technology
Haifa, 32000, Israel
e-mail: rodeh@cs.technion.ac.il

and

Iris Steinwarts
Computer Science Department, Technion — Israel Institute of Technology
Haifa, 32000, Israel
e-mail: iriss@cs.technion.ac.il

ABSTRACT

Modern machines present two challenges to algorithm engineers and compiler writers: They have superscalar, super-pipelined structure, and they have elaborate memory subsystems specifically designed to reduce latency and increase bandwidth. Matrix multiplication is a classical benchmark for experimenting with techniques used to exploit machine architecture and to overcome the limitations of contemporary memory subsystems.

This research aims at advancing the state of the art of algorithm engineering by balancing instruction level parallelism, two levels of data tiling, copying to provably avoid any cache conflicts, and prefetching in parallel to algorithmic operations, in order to fully exploit the memory bandwidth. Measurements show that the resultant matrix multiplication algorithm outperforms IBM's ESSL by 6.8-31.8%, is less sensitive to the size of the input data, and scales better.

The techniques presented in this paper have been developed specifically for matrix multiplication. However, they are quite general and may be applied to other numeric algorithms. We believe that some of our concepts may be generalized to be used as compile-time techniques.

1. Introduction

As the gap between CPU and memory performance continues to grow, so does the importance of effective utilization of the memory hierarchy. This is especially evident in compute intensive algorithms that use large data sets, such as most numeric problems. The problem of dense matrix multiplication is a classical benchmark for demonstrating the effectiveness of techniques that aim at improving memory utilization. Matrix multiplication involves $O(N^3)$ scalar operations on $O(N^2)$ data items. Moreover, this ratio can be preserved when performing the multiplication operation as a sequence of operations on sub-matrices. This feature of the problem, which is shared by other numeric problems, allows efficient utilization of the memory subsystem.

The efficient implementation of compute-intensive algorithms that use large data sets present a unique engineering challenge. To allow the implementation to exploit the full potential of the

program's inherent instruction level parallelism, the adverse effects of the processor-memory performance gap should be minimized. A well engineered compute-intensive algorithm should:

- Manage with small caches;
- Avoid cache conflicts;
- Hide memory latencies associated with "cold-start" cache misses.

A broad set of techniques has been suggested to adapt numeric algorithms to the peculiarities of contemporary memory subsystems. These techniques include software pipelining [8], blocking (tiling) [9] and data copying [9, 17]. Each of these techniques was designed as a solution to one of the first two engineering challenges presented above. The relatively new method of selective software prefetching [2, 11, 12] aims at the third challenge. Software prefetching attempts to hide memory latencies by initiating a prefetch instruction sufficiently early, before the data item is used. However, the implementation should be carefully designed to avoid cache pollution by the prefetched data (see [10]). If the prefetch instruction is non-blocking, the memory access will be executed in parallel with the computations carried out by the CPU. Previous attempts to improve the performance of BLAS-2 routines are reported in [1]. Similar work on BLAS-3 routines is presented in [13]. Both these efforts employ a variety of techniques in order to overcome the memory hierarchy limitations. However, so far, they do not meet all three challenges simultaneously.

We propose a new cache-aware $O(N^3)$ matrix multiplication algorithm which builds upon known techniques to meet all of the three engineering goals. Our algorithm is based on two observations: (i) The fact that in matrix multiplication the ratio between the number of scalar operations and the data size remains high, even when the problem is divided into a sequence of multiplications of sub-matrices. (ii) The system bus is a valuable hardware resource that should be taken into account in the algorithm design.

Our algorithm uses a new blocking scheme that divides the matrices into relatively small non-square tiles, and treats the matrix multiplication operation as a series of tile multiplication phases. The data required for each phase is designed to completely fit in the cache. In addition, our scheme maintains a high ratio of scalar operations to the number of data items for each phase.

To maintain conflict-free mapping of the data regardless of the associativity level of the cache, the algorithm restructures the matrices into an array of interleaved tiles. The copying operation can be carried out during the multiplication process, using only a small copying buffer.

To cope with memory latency, all data required during phase i must be prefetched into the data cache during phase $i-1$. This is done simultaneously with the actual computation. The two activities must be well balanced. In particular, the smaller the latency and the higher the memory bandwidth — the smaller the portion of the cache needed. The order and timing of the prefetch instructions is designed to make sure that relevant data is not flushed from the cache. When doing so, the associativity level of the cache must be taken into account.

Our cache-aware $O(N^3)$ matrix multiplication algorithm does not suffer memory latency when running on an architecture that fits the assumptions of our machine model. The performance of our algorithm is not influenced by the size or layout of the input matrices. Assuming that the data set fits in the main memory of the machine, our algorithm maintains its behavior regardless of the data set size. In addition, unlike traditional blocking based algorithms, our algorithm shows little sensitivity to small changes in the input size.

We implemented our algorithm on an IBM RS/6000 PowerPC 604 based workstation. Our implementation allows instruction level parallelism by using tiling at the register level, combined with loop unrolling and software pipelining. The scheduling of machine instructions builds on the fact that in our algorithm, memory access operations in the inner-most loop are always serviced by the cache. Our implementation outperforms IBM's BLAS-3 matrix multiplication function by roughly 21.5%, on the average, for double precision data. For some values of N, our implementation runs 31.8% faster.

In this work we demonstrat memory hierarchy oriented optimizations for the $O(N^3)$ matrix multiplication algorithm. However, the same ideas can be used to achieve similar improvements in many other linear algebra algorithms that exhibit similar features (see [16]).

In the following section we describe the assumptions that we make on the machine architecture. In Section 3 we outline our techniques and their application for matrix multiplication, while in Section 4 we present our implementation for the IBM RS/6000 platform.

2. The Machine Model

When optimizing an algorithm, it is important to properly choose a simple, but sufficiently accurate machine model. The objective is then to define an abstract model such that an algorithm optimized for it will perform well in practice.

In our case, we deliberately decide to ignore the effects of the virtual memory subsystem (see, for example, [7]). Specifically, we ignore the paging mechanism, the use of virtual versus physical addresses for cache indexing, and the use of a Translation Look-aside Buffer (TLB) to shorten the address translation process. As a consequence, the negative effects of page faults and TLB misses are not taken into account. Furthermore, we assume a virtually-indexed data cache. This assumption is required to allow the algorithm to use virtual addresses when it restructures data in a manner that will assure conflict-free mapping into the cache. However, the algorithm may be adapted, under certain circumstances, to use physically-indexed data caches. Indeed, out implementation uses such a cache.

Other assumptions that we make regarding the target machine are:

- The memory subsystem includes at least one level of data cache. Our optimization techniques target only the first level (L1) data cache. We assume that slower caches do not degrade the performance of the L1 cache. Specificaly, we require that they do not affect the L1 replacement policy.
- The L1 data cache write policy is copy-back.
- The L1 data cache replacement policy is Least Recently Used (LRU).
- The processor supports a non-blocking cache fetch instruction. This may be a specialized prefetch instruction, or a simple non-blocking load instruction [4].
- The CPU follows a load/store (register-register) architecture.

We use the following parameters in the description of our algorithm: The L1 data cache holds C bytes arranged in lines of size L. The cache is K-way set associative. We denote by M the number of machine cycles required to fetch a complete cache line from memory (contemporary machines have $20 \leq M \leq 100$).

3. The Algorithm

Our algorithm is designed to carry out matrix multiplication of the form

$$\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$$

where $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are real matrices of sizes $N_1 \times N_2$, $N_2 \times N_3$ and $N_1 \times N_3$, respectively. We denote by I the matrices' element size, in bytes. We make no assumption regarding the layout, or relative location, of the input matrices in memory. In the following subsections we outline the algorithm, describing the use of each of the optimization techniques and the way in which the algorithm combines them. We end up with a matrix multiplication algorithm, that by construction, does not suffer memory latency when running on an architecture that fits the assumptions of our machine model.

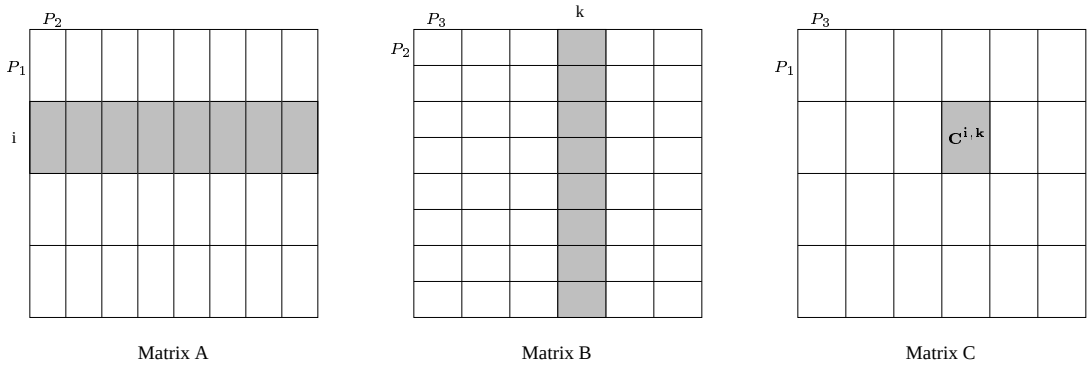


Figure 1: Tile multiplication

3.1. Tolerating Capacity and Cold Start Misses

In this subsection we assume that the cache is fully associative. We postpone the discussion on eliminating cache conflicts and pollution to the following subsections.

The algorithm partitions the input matrices into tiles (see Figure 1). The matrix $\mathcal{C}$ is divided into tiles of size $P_1 \times P_3$. Each one of these tiles is generated by a sum of products of a horizontal stripe of $\mathcal{A}$-tiles and a vertical stripe of $\mathcal{B}$-tiles. The matrix $\mathcal{A}$ is thus divided into tiles of size $P_1 \times P_2$ while $\mathcal{B}$ is divided[1] into tiles of size $P_2 \times P_3$. Denoting by $\mathcal{M}^{i,j}$ the (i,j)th tile of the matrix $\mathcal{M}$, we have:

$$\mathcal{C}^{i,k} = \sum_{j=0}^{N_2/P_2-1} \mathcal{A}^{i,j} \cdot \mathcal{B}^{j,k}$$

Since we have $N_1N_3/(P_1P_3)$ $\mathcal{C}$-tiles to compute and since the computation for each tile requires N_2/P_2 tile multiplications, we have a total of $(N_1N_2N_3)/(P_1P_2P_3)$ tile multiplication phases. In each phase we multiply an $\mathcal{A}$-tile of size $P_1 \times P_2$, by a $\mathcal{B}$-tile of size $P_2 \times P_3$, updating a $\mathcal{C}$-tile of size $P_1 \times P_3$, using $P_1P_2P_3$ scalar multiplications and $P_1P_2P_3$ scalar additions. The number of data items accessed in each phase is $P_1P_2 + P_2P_3 + P_1P_3$.

In order to achieve optimal performance, all data used in a specific phase must be present in the data cache at the beginning of the phase. If this condition is indeed met, the system bus remains unused by the tile multiplication code and can instead be used to bring the data required for the *next* phase into the cache. Let W denote the number of machine cycles it takes to multiply an $\mathcal{A}$-tile by a $\mathcal{B}$-tile and store the result in $\mathcal{C}$. Then:

$$W = \Theta(P_1P_2P_3)$$

The total amount of data (in bytes) required for each phase is:

$$I \cdot (P_1P_2 + P_2P_3 + P_1P_3)$$

Assuming that each prefetch instruction fills a single line of the cache, the number of prefetch instructions we must issue is

$$\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + P_1P_3).$$

If

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2P_1P_3) \leq W \tag{1}$$

[1] Naturally, this implies that P_1, P_2 and P_3 must all divide the respective dimension, N_i. Padding of the matrices may be used to meet this requirement.

```
for (i = 0 ; i < N1/P1 ; i++)
    for (j = 0 ; j < N3/P3 ; j++)
        for (k = 0 ; k < N2/P2 ; k++) {
            C^{i,j} = C^{i,j} + A^{i,k} · B^{k,j};
        }
```

Figure 2: The algorithm's outer loops.

then a single phase runs long enough to allow us to prefetch all the data required by the next phase on time. Note that in Equation (1) the size of the $\mathcal{C}$-tile is multiplied by 2. This is because $\mathcal{C}$-tiles are modified and therefore must be written back to memory, occupying the system bus.

To allow for the latency-free operation of the algorithm, the cache must be large enough to concurrently hold all the data required for a certain phase (i.e., one tile of each matrix: $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$) as well as the data that will be used in the next phase. All in all, the cache must be able to hold two tile triplets, so the total cache size must be at least

$$2I \cdot (P_1P_2 + P_2P_3 + P_1P_3) \leq C. \tag{2}$$

Note that Equation (2) places an upper bound on P_1, P_2 and P_3, while the timing considerations of Equation (1) place a lower bound on these values.

For machines with a large CPU-memory performance gap, it might happen that Euqations (1) and (2) cannot hold simultaneously for any value of P_1, P_2 and P_3. However, in Equation (1) we calculated the number of prefetch instructions required in a single phase assuming that all three tiles should be replaced. The number of prefetch instructions required may be reduced by ordering the phases so that one of the tiles is reused. The code in Figure 2 replaces $\mathcal{C}$-tiles only once in every N_2/P_2 phases. The reuse of $\mathcal{C}$-tiles is beneficial, since these tiles are the only ones that are modified and therefore must be written back to memory. Using the scheme presented in Figure 2, the number of prefetch instructions needed in a single phase is reduced to:

$$\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + \frac{P_1P_2P_3}{N_2}).$$

This means, that the condition of Equation (1) may be relaxed to:

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2\frac{P_1P_2P_3}{N_2}) \leq W \tag{3}$$

In case this does not yield feasible values of P_1, P_2 and P_3, the implementation's performance may degrade.

Assuming a memory bus that does not allow for pipelined memory access or outstanding requests, all prefetch instructions must be spaced at intervals of at least M units of time, to prevent stalling the CPU. With a more advanced bus that allows multiple outstanding memory requests, this requirement can be relaxed, as long as the request queue never overflows. For buses that allow pipelined memory access, a more detailed calculation of the effective value of M should be carried out.

3.2. Avoiding Cache Conflicts

Our algorithm is based on the assumption that any two triplets of tiles (one from each matrix) that are used in two consecutive tile multiplication phases can simultaneously reside in the cache. So far, we have assumed full associativity of the cache. Most practical caches have a limited degree

of associativity. Restructuring the data by copying it to properly designed locations can be used to avoid cache conflicts. [17] discusses the utilization of copying to avoid cache conflicts. In the case of matrix multiplication, copying is potentially beneficial, as it takes only $O(N^2)$ time while the computation takes $O(N^3)$ time, assuming that every data element is copied only once.

When using a K-way set associative cache ($K > 1$, K is even[2]), the following condition allows any two triplets of tiles to be mapped into the cache simultaneously:

Condition 3.1. *Associate with each of the three matrices a multi-set of cache set indices of size sufficient to hold one tile of that matrix. Use copying to map each tile of a specific matrix to the multi-set of cache sets that is associated with that matrix. The mapping is conflict-free if the multi-set union of the multi-sets for the three matrices does not contain any index more than $K/2$ times.*

This condition is sufficient but not necessary, since it is equivalent to having *any* two triplets fit in the cache, and not just the $(N_1N_2N_3)/(P_1P_2P_3)$ triplets used in matrix multiplication. Note that a scheme that complies with Condition 3.1 may be easily designed to copy each data element only once.

When using a 2-way set associative cache, Condition 3.1 may be met by mapping the tiles so that all tiles of a single matrix are mapped to the same sets in the cache, and each cache set is mapped only once. Such a mapping may be formed by interleaving the matrices' tiles, so that the offset between two tiles of the same matrix is a multiple of the way size of the cache. Since the cache is assumed to have two ways, we can have two tiles from each of the matrices resident in the cache simultaneously.

For a direct mapped cache, Condition 3.1 cannot be satisfied. To allow conflict-free mapping for direct mapped caches, we must use the fact that not every combination of three tiles from the matrices $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ is used. Each matrix will have two possible sets of cache set indices for its tiles, with half of the tiles using one set and the other half using the second set. The tile mapping is chosen so that whenever a tile is replaced by a tile from the same matrix, the two tiles use different sets, and therefore, do not conflict.

To design such a mapping, we divide the cache lines into two equal-sized subsets: a "black" subset and a "white" subset. Each of these subsets is designed to hold one tile from each matrix. Each of the matrices will have half of its tiles mapped to black cache sets and the other half mapped to white cache sets. For the sake of this explanation, we assume that each dimension of the matrices is divided into an even number of tiles (i.e., N_1/P_1, N_2/P_2 and N_3/P_3 are all even). The matrix $\mathcal{A}$ is copied so that all even-numbered *vertical stripes* of tiles are mapped to the black part of the cache and all odd-numbered vertical stripes of tiles are mapped to the white part of the cache. The matrices $\mathcal{B}$ and $\mathcal{C}$ are copied so that all even-numbered *horizontal stripes* of tiles are mapped to the black part of the cache and all odd-numbered horizontal stripes of tiles are mapped to the white part of the cache. It can now be easily verified, that when multiplying tiles in the order of Figure 2, whenever a tile is replaced by a tile from the same matrix, the tile replacing it is colored differently, and therefore the two tiles do not conflict. Note that this method can also be implemented while copying each data element only once.

While it is possible to copy the matrices to their new locations before engaging in the multiplication process itself, it is also possible to interleave the copy operation with the computation. For an elaborate discussion of copying on the fly, see [16].

3.3. Cache Pollution

As observed in [10, 13] care must be taken, when performing prefetch operations, in order to assure that essential data is not flushed out of the cache. Similarly, we have to assure that fresh prefetched data is not flushed out before it is used for the first time. Therefore, we have to examine

[2]This assumption on K is used for simplicity. If it does not hold, use $\lfloor K/2 \rfloor$ instead of $K/2$ in the discussion that follows.

what cache lines are chosen for flushing by the replacement policy when a new cache line is brought into the cache. Since the replacement policy is applied to each set separately, we limit our discussion to a single set.

Consider a K-way set associative cache. Recall that $K/2$ lines of each set are being used by the active triplet of tiles, and the other $K/2$ lines are used to fetch the next triplet before the current phase is over. Let $\{p_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which prefetch instructions have been executed during the jth phase, $\{f_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which the *first* access to each line holding data for the jth phase has been executed, and $\{l_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which the *last* access to each such line during the jth phase took place.

Let us now describe the conditions on the access pattern that will allow pollution-free prefetching. The first condition governs the prefetching code:

Condition 3.2. *Every block of the tiles used in the jth phase is accessed (prefetched) only once during phase $j-1$.*

Assuming that the above condition holds, the following condition is sufficient and necessary for pollution-free prefetching:

Condition 3.3. *For every $0 \leq i \leq K/2-1$ and every phase j, denote by r the least index for which $f_r^j > p_i^j$ and by s the least index for which it holds that $p_s^{j-1} > l_i^{j-1}$. If no such r exists, the prefetch instruction p_i^j is pollution-free. Otherwise, denote by F the cache lines accessed by $\{f_k^j\}_{k=0}^{r-1}$ and by P the cache lines accessed by $\{p_k^{j-1}\}_{k=0}^{s-1}$. The prefetch is pollution-free iff $P \subseteq F$.*

4. Implementation

To experiment with our approach, we implemented our matrix multiplication algorithm for both single and double precision square matrices. The target platform was a 133MHz PowerPC 604 based IBM RS/6000 43P Model 7248 workstation. The algorithm was implemented in C and was compiled using the IBM XL-C compiler [15]. Where necessary, hand-tuning of the resulting machine code was carried out. To gauge the performance improvements over known techniques, we compared our results to IBM's Engineering Scientific Subroutine Library (ESSL) [3], which is the state of the art implementation of BLAS provided by IBM for this platform.

4.1. Platform Description

The PowerPC 604 [14] is a superscalar, super-pipelined RISC processor. The CPU contains one floating point unit (FPU) and one load-store unit (LSU). It has 32 architectural floating point registers and 32 architectural integer registers. The processor has a 16KB on-chip instruction cache and a separate 16KB on-chip data cache ($C = 16384$). Both caches are *physically indexed*, four-way set associative ($K = 4$), and have a line size of 32 bytes ($L = 32$). The write policy for the on-chip data cache is write-back and the replacement policy is LRU. Access to the cache is done via non-blocking load/store instructions. Note that the PowerPC 604 processor adheres to all of the assumptions of our machine model, except for the use of a physically indexed L1 data cache.

In addition to the L1 cache the machine has an off-chip L2 *directly mapped and physically indexed,* unified cache of size 512KB. The line size of the L2 cache is 32 bytes. The L2 cache controller implements full inclusion of both on-chip L1 caches. Note that this L2 cache design contradicts our assumption that slower caches do not interfere with the replacement policy of the L1 cache.

The AIX operating system uses the PowerPC 604's MMU to implement demand paged virtual memory. The page size is 4KB, the same as the size of a way of the L1 data cache. This allows us to ignore the page-number part of the virtual address when mapping data to the L1 cache, making the distinction between physically and virtually indexed caches irrelevant.

To complete the picture, let us now examine the features of the PowerPC 604 instructions, which are relevant to our work. The PowerPC architecture supports a floating point multi-add (`fma`) instruction which performs two floating point operations: a multiplication and an addition. This instruction has four register operands and performs the following calculation: fp1 $\leftarrow$ fp2 $\cdot$ fp3 + fp4. The pipelined structure of the PowerPC CPU supports issuing one independent floating point instruction in every cycle. The usage of the `fma` instruction is most appropriate for matrix multiplication, since it allows the PowerPC 604 to complete two floating point operations in every cycle. By implementing the tile multiplication code using `fma` instructions, that use the FPU while loads and stores execute in parallel in the LSU, we assume that the value of W (the amount of work for a single tile multiplication phase) is roughly equal to $P_1P_2P_3$ machine cycles.

Floating point load instructions that hit in the on-chip L1 data cache usually complete in 3 cycles. Since the L1 data cache access is pipelined, one load/store instruction may complete in every cycle.

A load instruction that misses the L1 data cache but hits in the L2 data cache completes in approximately 20 cycles. A load instruction that misses both caches takes roughly 80 cycles (we therefore assume that $M = 80$). While the PowerPC 604 may have up to four outstanding load/store instructions, memory access is not pipelined.

As far as prefetching is concerned, the PowerPC 604 supports a special Data Cache Block Touch (`dcbt`) instruction that fetches the cache line which corresponds to its virtual effective address (VEA) into the cache. While our algorithm seems to be designed to use such an instruction, we decided to use a standard non-blocking register load instruction instead (see [16] for a discussion of this issue).

4.2. Implementation Details

We implemented the algorithm for both single and double precision floating point numbers. As described in Section 3.1, when breaking up matrix multiplication into phases, the algorithm designer can sequence the phases to maximize tile reuse. Since our target platform suffers from high memory latency, we indeed took advantage of this observation. In particular, we choose to reuse every $\mathcal{C}$-tile in every N/P_2 consecutive phases before replacing it. For the single precision implementation, we choose the following values for the tile-size parameters: $P_1 = P_3 = 32$ and $P_2 = 16$.

Proposition 4.1. *For single precision implementation on the IBM RS/6000 43P Model 7248, the choice $P_1 = P_3 = 32$ and $P_2 = 16$ complies with both Equations (2) and (3), assuming the input matrices are at least of size 6×6.*

Proof. Each $\mathcal{A}$-tile and each $\mathcal{B}$-tile have 512 elements (or 2KB in size), while each $\mathcal{C}$-tile has 1024 elements (or 4KB in size). We see that a triplet has 2KB + 2KB + 4KB = 8KB. Since the cache is 16KB in size, Equation (2) is satisfied.

The total amount of work per phase is $W = P_1P_2P_3 = 32 \cdot 16 \cdot 32 = 16384$. Plugging in Equation (3) the values for M, P_1, P_2, P_3, I, and L, we have:

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2\frac{P_1P_2P_3}{N_2}) = 80\frac{4}{32}(512 + 512 + 2\frac{16384}{N}) \leq 16384 = W$$

This inequality holds for all $N \geq 16/3$. Since we assume $N \geq 6$, Equation (3) is satisfied. ∎

When choosing to reuse $\mathcal{C}$-tiles the frequency of prefetches is dependent on N. Since N is an input parameter of our algorithm, prefetching data from $\mathcal{C}$ at the correct frequency would have required the use of code that checks, at relatively high granularity, whether a prefetch should be executed. For the efficiency of implementation, we design $\mathcal{C}$ prefetch code to execute outside of the inner loops, prefetching only $\mathcal{A}$- and $\mathcal{B}$-tiles inside the inner-most loop. Since we have 4KB of data, which fills 128 cache lines, to prefetch during 16384 cycles, a line should be prefetched once every 128 cycles. To efficiently interleave the prefetch instructions in the tile multiplication code, we unroll the inner-most loop so that its computations take 256 cycles, long enough for one prefetch of $\mathcal{A}$ and one

```
struct Triplet {
       float A_tile[P1][P2];
       float C_half1[P1][P3/2];
       float B_tile[P2][P3];
       float C_half2[P1][P3/2];
};
```

Figure 3: Conflict free mapping

prefetch of $\mathcal{B}$. As the processor supports outstanding memory requests, the timing of the relatively few prefetch instructions for the next $\mathcal{C}$-tile, executed outside of the inner loop, is not crucial.

To allow conflict-free mapping of two tile triplets into the cache, the matrices are first copied into a page-aligned array of the `Triplet` structure, as shown in Figure 3.

Proposition 4.2. *For single precision implementation on the IBM RS/6000 43P Model 7248, copying the tiles of the three matrices to a page-aligned array of the* `Triplet` *structure satisfies Condition 3.1.*

Proof. The array is page-aligned, each structure is exactly the size of two pages, and the page size is the same as the L1 way size. This implies that every $\mathcal{A}$-tile is mapped into sets 0 to 63 of the cache, exactly once, and so is every $\mathcal{B}$-tile. For sets 64 to 127 of the cache, there are two lines of each $\mathcal{C}$-tile mapped into each set. As the cache is 4-way set associative, Condition 3.1 is satisfied, allowing the simultaneous mapping of any two triplets of tiles into the cache. ∎

To implement the tile multiplication code, we used tiling at the register level. We divided $\mathcal{B}$-tiles into sub-tiles of size 4×4 elements each. $\mathcal{A}$- and $\mathcal{C}$-tiles are divided accordingly into vertical stripes. In our inner-most loops we load a single sub-tile of $\mathcal{B}$ into 16 registers and then traverse a 4-element wide vertical stripe of both $\mathcal{A}$ and $\mathcal{C}$. In each iteration, we multiply four elements of $\mathcal{A}$ by a sub-tile of $\mathcal{B}$, while updating four elements of $\mathcal{C}$, totaling 16 scalar multiplications (this loop is unrolled 16 times, so that it runs in 256 cycles).

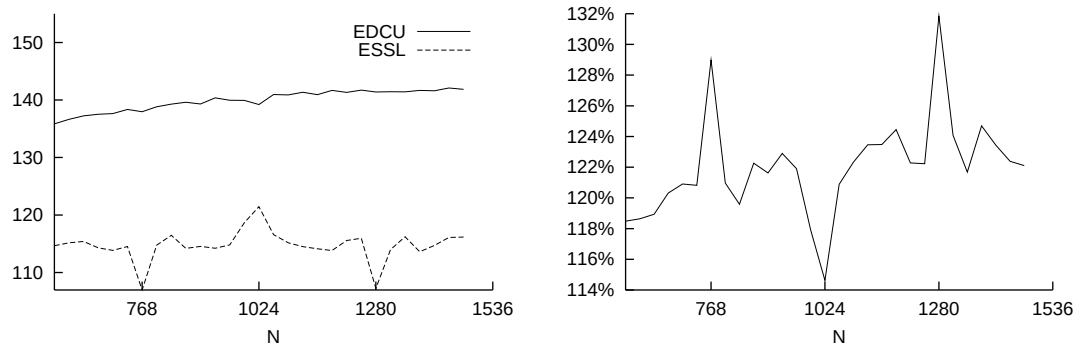
Mapping of elements within the tile storage area is designed such that first access made by the tile multplication code occur in increasing address order. $\mathcal{A}$ and $\mathcal{C}$-tiles are copied in vertical stripes, four element wide. Each such 4-element sub-row occupies consecutive memory addresses, and these sub-rows are ordered in column-major order. $\mathcal{B}$-tiles are copied such that every sub-tile of size 4×4 occupies a contiguous area of memory. These sub-tiles are again arranged in column-major order within the tile.

Prefetches are carried out according to the tiles' layout in memory, i.e., in order of increasing addresses. The prefetches of $\mathcal{A}$ and $\mathcal{B}$ are interleaved within the inner-most loop; one from $\mathcal{A}$ and then one from $\mathcal{B}$. Prefetches for the two halves of $\mathcal{C}$ are interleaved, with each half accessed in increasing adderss order. Every block is prefetched only once, satisfying Condition 3.2.

Proposition 4.3. *For single precision implementation on the IBM RS/6000 43P Model 7248, our implementation complies with Condition 3.3.*

For a complete proof of Proposition 4.3, see [16].

For double precision data, the bandwidth requirements are higher by a factor of 2 as compared to single precision. In addition, the space taken up in the cache is also doubled. This aggrevates the problem of hiding the memory latency. No values for P_1, P_2 and P_3 allows us to hold two triplets of tiles in the cache simultaneously and to hide memory latency completely. Therefore, we *forgo the*



(a) (b)

Figure 4: Performance of double precision matrix multiplication on the RS/6000 43P Model 7248: (a) In MFLOPS, (b) Relative to ESSL

prefetching of data from $\mathcal{C}$, relaxing Equation (2) to require that the cache be large enough to hold two pairs of $\mathcal{A}$- and $\mathcal{B}$-tiles and a single $\mathcal{C}$-tile simultaneously, and relaxing Equation (1) to require that a single phase runs long enough to prefetch one $\mathcal{A}$-tile and one $\mathcal{B}$-tile.

For the double precision implementation we choose the following values for the tile size parameters: $P_1 = P_3 = 32$ and $P_2 = 8$. Given these tile parameters, each $\mathcal{A}$-tile and each $\mathcal{B}$-tile is 2KB in size, while each $\mathcal{C}$-tile is 8KB in size. The total space required by two pairs of $\mathcal{A}$- and $\mathcal{B}$-tiles and one $\mathcal{C}$-tile is therefore 16KB, the same as the cache size.

The total amount of work per phase is $W = P_1 P_2 P_3 = 8192$. During this time we prefetch only one $\mathcal{A}$-tile and one $\mathcal{B}$-tile, totalling in 128 cache lines. This means that prefetch instructions should be executed once every 64 cycles. Since memory latency is $M = 80$ machine cycles, the prefetch instruction hide only 80% of the memory latency for accesses to $\mathcal{A}$ and $\mathcal{B}$.

To provide partial remedy to the performance penalty observed in the double precision case, we developed a variant that departs from our generic cache-aware matrix multiplication algorithm. We took advantage of the fact that $\mathcal{C}$ is used for output only, and therefore the initial values of $\mathcal{C}$ are all zeros. We therefore used a single memory buffer to hold $\mathcal{C}$-tiles, which is always resident in the cache. Whenever a new $\mathcal{C}$-tile is required, the old one is saved into the original $\mathcal{C}$ matrix and the buffer is cleared. By using a single, cache-resident, buffer to hold $\mathcal{C}$-tiles, we compensate for our inability to prefetch the data from $\mathcal{C}$. However, since the L1 cache uses allocate on write policy, copying back of the $\mathcal{C}$-tiles allocates cache lines to hold the modified data of the $\mathcal{C}$ matrix. This allocation may cause flushing of prefetched data from $\mathcal{A}$- and $\mathcal{B}$-tiles that is intended for use in the next tile multiplication phase. Since the combined size of an $\mathcal{A}$-tile and of a $\mathcal{B}$-tile is only 4KB, compared to 8KB for a single $\mathcal{C}$-tile, the saving of delays for access to $\mathcal{C}$-tiles is advantageous.

4.3. Results

Figure 4 (a) shows the performance in MFLOPS of our EDCU (Enhanced Data Cache Utilization) single $\mathcal{C}$-buffer double precision matrix multiplication vs. IBM's BLAS-3 (shown as ESSL). Figure 4 (b) shows the relative performance of these two implementations.

For double precision data we get the following average performance figures for the sizes of inputs we checked: IBM's ESSL implementation achieves 115 MFLOPS, our standard implementation achieves 130 MFLOPS, and our single $\mathcal{C}$-buffer implementation achieves 140.8 MFLOPS, a 21.5% average advantage over ESSL. Another feature of our algorithm that can be clearly seen from Figure 4 is that, performance-wise, our design is less sensitive to changes in the size of the data in comparison to ESSL. The performance instability evident in ESSL allows the single $\mathcal{C}$-buffer implementation to outperform the ESSL implementation by up to 31.8% on some double precision matrices.

For single precision data, the tile sizes we use are sufficient to allow prefetching of all the required data. However, the memory bandwidth is not taxed as heavily in the single precision implementation, as it is in the double precision implementation, and so the potential for performance improvement via memory latency hiding is smaller. This is clearly demonstrated by the performance achieved by the naive 3-loop $O(N^3)$ matrix multiplication algorithm. While for double precision the naive 3-loop implementation achieves only 13.6 MFLOPS on average, allowing us to achieve 935% performance increase over it, its performance is almost doubled to 23.7 MFLOPS on average for single precision numbers. For the single precision implementation, we got the following average performance figures for the sizes of input we checked: IBM's ESSL implementation achieves 154.9 MFLOPS while our EDCU implementation achieves 165.5 MFLOPS, a 7% average increase over ESSL, and up to 13% for some single precision matrices.

The main reasons for not reaching the full potential of the CPU are related to the specific machine which does not fit the assumptions taken by the algorithm. First, the machine has a relatively small L1 cache, when considering its memory latencies. Our double precision implementation could not prefetch the $\mathcal{C}$-tile, forcing us to use instead, a single $\mathcal{C}$ buffer that is copied back on every N/P_2 tile multiplication phases. The copy back operation pollutes the cache, causing delays in subsequent accesses to $\mathcal{A}$- and $\mathcal{B}$-tiles. In addition, since the tile sizes we use allow only 64 cycles between prefetch instructions, we cannot fully hide memory latency, which is as high as 80 cycles. Therefore, the inner-most loop of the double precision implementation is prolonged by roughly 25% (the difference between 80 and 64). Second, the direct mapped L2 cache forces a full inclusion policy on the instruction and data L1 caches. Therefore, some data may be flushed out of the L1 cache because of conflicts in the L2 cache, which may even result from instruction access. Third, as noted in Section 3.2, before engaging in the actual multiplication process, we copy our input matrices into an array of interleaved tiles. For the sake of simplicity, we choose to carry these copying operations off-line. These copying operations take time and our measurements indicate that the overhead for the input sizes we used is at least 14% of the peak for double precision data. Clearly, since this overhead is $O(N^2)$ its relative influence on performance diminishes as the size of the data increases.

5. Conclusion

To achieve good performance, numeric algorithms should balance computation with data movement. We have presented a new cache-aware $O(N^3)$ matrix multiplication algorithm. This algorithm can be proven to suffer no memory latency when running on an architecture that fits the assumptions of the machine model introduced. Furthermore, this algorithm uses only the smallest part of the cache that can still balance the memory bandwidth. Using a larger cache than the minimum required will have no further impact on performance.

Our experiments show that, even for platforms that are not idealy suited for the suggested techniques, the implementation of the matrix multiplication algorithm we present is competitive: It achieves performance that is higher by 6.8–31.8% than that attained by the vendor's state of the art implementation, which also uses advnaced memory hierarchy oriented optimizations [5].

While the technique presented in this work was demonstrated for $O(N^3)$ matrix multiplication, generic guidelines and conditions for cache-aware design of compute intensive algorithms were formulated and are presented in full in [16]. Furthermore, insight regarding effective exploitation of modern hardware has been gained. We believe that some of our concepts may be generalized for use as compile-time techniques.

6. Acknowledgements

Many members of the IBM Haifa Research Lab helped us significantly during this research. We are specifically indebted to David Bernstein for his helpful advice, that contributed greatly to this

work.

References

[1] R. C. Agarwal, F. G. Gustavson and M. Zubair, *Improving Performance of Linear Algebra Algorithms for Dense Matrices, Using Algorithmic Prefetch*, IBM J. Research & Development, **38(3)** (1994) 265–275.

[2] D. Callahan, K. Kennedy and A. Porterfield, *Software Prefetching*, proceedings of ASPLOS'91, 1991, 40–52.

[3] *IBM Engineering and Scientific Subroutine Library for AIX, Version 3 – Guide and Reference*, IBM Corp. 1997.

[4] K. Farkas and N. Jouppi, *Complexity/Performance Tradeoffs with Non-Blocking Loads*, Proceedings of ISCA, 1994, 211-222.

[5] F. G. Gustavson, *Personal Communications*, 1998.

[6] S. Hoxey, F. Karim, B. Hay and H. Warren (eds.), *The PowerPC Compiler Writer's Guide*, IBM Microelectronics Division, 1996.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitive Approach, 2 ed.*, 1996.

[8] M. S. Lam, *Software Pipelining: An Effective Technique for VLIW Machines*, SIGPLAN'88, 1988, 318–328.

[9] M. S. Lam, E. E. Rothenberg and M. E. Wolf, *The Cache Performance and Optimizations of Blocked Algorithms*, proceedings of ASPLOS'91, 1991, 63–74.

[10] J. H. Lee, M. Y. Lee, S. U. Choi and M. S. Park, *Reducing Cache Conflicts in Data Cache Prefetching*, Computer Architecture News, **22(4)**, (1994) 71–77.

[11] T. Mowry, *Tolerating Latency Through Software-Controlled Data Prefetching*, Ph.D. Thesis, Stanford university, 1994.

[12] T. C. Mowry, M. S. Lam and A. Gupta, *Design and Evaluation of a Compiler Algorithm for Data Prefetching*, proceedings of ASPLOS'92, 1992, 62–73.

[13] J. J. Navarro, E. García-Diego, J. R. Herrero, *Data Prefetching and Multilevel Blocking for Linear Algebra Operations*, proceedings of ICS'96, pp. 109–116, 1996.

[14] *PowerPC 604 RISC Microprocessor User's Manual*, IBM Microelectronics and Motorola Inc. 1994.

[15] K. E. Stewart, *Using the XL Compiler Options to Improve Application Performance*, PowerPC and POWER2, Technical Aspects of the new IBM RISC System/6000, IBM Corp. 1994.

[16] I. Steinwarts, *Matrix Multiplication: A Case Study of Enhanced Data Cache Utilization*, M.Sc. thesis, Department of Computer Science, The Technion — Israel Institute of Technology, 1998.

[17] O. Temam, E. D. Granston and W. Jalby, *To Copy or Not to Copy: A Compile-Time Technique for assessing When Data Copying Should be Used to Eliminate Cache Conflicts*, SUPERCOMPUTING'93 1993, 410–419.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 110–120

Guarding Scenes against Invasive Hypercubes

Mark de Berg[1]
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands
e-mail: markdb@cs.uu.nl

Haggai David[2]
Department of Mathematics and Computer Science, Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
e-mail: davidcha@cs.bgu.ac.il

Matthew J. Katz[2]
Department of Mathematics and Computer Science, Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
e-mail: matya@cs.bgu.ac.il

Mark Overmars[1]
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands
e-mail: markov@cs.uu.nl

A. Frank van der Stappen[1]
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands
e-mail: frankst@cs.uu.nl

and

Jules Vleugels[3]
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands
e-mail: jules@cs.uu.nl

ABSTRACT

A set of points $\mathcal{G}$ is a κ-guarding set for a set of objects $\mathcal{O}$, if any hypercube not containing a point from $\mathcal{G}$ in its interior intersects at most κ objects of $\mathcal{O}$. This definition underlies a new input model, that is both more general than de Berg's *uncluttered*ness, and retains its main property: a d-dimensional scene satisfying the new model's requirements is known to have a linear-size binary space partition.

We propose several algorithms for computing κ-guarding sets, and evaluate them experimentally. One of them appears to be quite practical.

[1]Supported by the ESPRIT IV LTR Project No. 21957 (CGAL).
[2]Supported by the Israel Science Foundation founded by the Israel Academy of Sciences and Humanities.
[3]Supported by the Netherlands' Organization for Scientific Research (NWO).

1. Introduction

Recently de Berg et al. [4] brought together several of the *realistic input models* that have been proposed in the literature, namely *fatness*, *low density*, *unclutteredness*, and *small simple-cover complexity* (see definitions below). They showed that these models form a strict hierarchy in the sense that fatness implies low density, which in turn implies unclutteredness, which implies small simple-cover complexity, and that the reverse implications are false.

For each of the models above, data structures and algorithms were proposed [1, 7, 8, 9, 10, 11], that perform better than their more general counterparts. Assume an efficient algorithm exists for some problem under one of the models, say, M_1. A natural question that arises is: Does there also exist an algorithm for the same problem under the more general model M_2, that is more or less comparable to the former algorithm in terms of efficiency. Clearly, it would be useful to have such an algorithm.

In [1] de Berg presents an algorithm for computing a linear-size binary space partition (BSP) for uncluttered d-dimensional scenes, which enables him (see [2]) to construct a linear-size data structure supporting logarithmic-time point location for such scenes. We have defined a new input model which is both more general than unclutteredness, and retains the property that a d-dimensional scene satisfying the new model's requirements is known to have a linear-size BSP, and consequently a linear-size data structure supporting logarithmic-time point location. However, in order to compute a linear-size BSP for such a scene, using de Berg's algorithm, we must first compute a linear-size *guarding set* for the scene (see definition below). This paper deals primarily with the problem of computing a small guarding set for such scenes. We describe three algorithms for this task and evaluate them both theoretically, but mainly experimentally. (We suspect that the problem of computing a smallest guarding set for such scenes is NP-complete.)

The notion of guarding sets was introduced very recently by de Berg et al. [3]. They showed that having a linear-size guarding set is essentially equivalent to having small simple-cover complexity, and used this result to prove that the complexity of the free space of a bounded-reach robot with f degrees of freedom moving in a planar uncluttered scene or in a planar scene with small simple-cover complexity is $\Theta(n^{f/2})$.

2. Preliminaries

We first define the two more general input models in the hierarchy of [4]. Unclutteredness was introduced by de Berg [1] under the name *bounding-box-fitness* condition. The model is defined as follows. (Throughout the paper, whenever we mention a square, cube, rectangle, etc., we assume it is axis-parallel.)

Definition 2.1. *Let $\mathcal{O}$ be a set of objects in $\Re^d$. We say that $\mathcal{O}$ is* κ-cluttered *if any hypercube whose interior does not contain a vertex of one of the bounding boxes of the objects in $\mathcal{O}$ is intersected by at most κ objects in $\mathcal{O}$. The* clutter factor *of $\mathcal{O}$ is the smallest κ for which it is κ-cluttered.*

We sometimes call a scene *uncluttered* if it is κ-cluttered for a small constant κ.

The following definition of simple-cover complexity is a slight adaptation of the original definition by Mitchell et al. [7], as proposed by de Berg et al. [4]. Given a scene $\mathcal{O}$, we call a ball *δ-simple* if it intersects at most δ objects in $\mathcal{O}$.

Definition 2.2. *Let $\mathcal{O}$ be a set of objects in $\Re^d$, and let $\delta > 0$ be a parameter. A* δ-simple cover *for $\mathcal{O}$ is a collection of δ-simple balls whose union covers the bounding box of $\mathcal{O}$. We say that $\mathcal{O}$ has* (s, δ)-simple-cover complexity *if there is a δ-simple cover for $\mathcal{O}$ of cardinality sn.*

We say that a scene has *small simple-cover complexity* if there are small constants s and δ such that it has (s, δ)-simple-cover complexity.

We close this preliminary section with a description of a variant of the first phase of de Berg's BSP-decomposition algorithm [1]. The algorithm has as input a set $\mathcal{P}$ of m points in $\Re^d$. Its output is a partitioning of a bounding cube of $\mathcal{P}$ into $O(m)$ hypercubes and L-shapes, with the property that they do not have a point from $\mathcal{P}$ in their interior. (An *L-shape* is the geometric difference of a hypercube σ with a hypercube $\sigma' \subseteq \sigma$ of less than half its size and sharing a vertex with it.) The partitioning is constructed with a recursive algorithm, whose planar version is given below.

Suppose we have a square σ at some stage in the subdivision process; initially σ can be any square containing all points from $\mathcal{P}$. Let $\mathcal{P}_\sigma$ denote the subset of points from $\mathcal{P}$ contained in the interior of σ. The square σ is handled according to the following rules.

1. If $\mathcal{P}_\sigma = \emptyset$ then σ is a cell in the final partitioning.
2. If $\mathcal{P}_\sigma \neq \emptyset$ and not all points of $\mathcal{P}_\sigma$ lie in the interior of a single quadrant of σ, then σ is subdivided into four quadrants, which are handled recursively.
3. If $\mathcal{P}_\sigma \neq \emptyset$ and all points of $\mathcal{P}_\sigma$ lie in the interior of a single quadrant of σ, then σ is subdivided as follows. Let σ' be the smallest square containing the points from $\mathcal{P}_\sigma$ that shares a vertex with σ. Note that σ' has a point from $\mathcal{P}_\sigma$ on its boundary. Now σ is handled recursively, and the L-shape $\sigma \setminus \sigma'$ is a cell in the final partitioning.

By charging each such subdivision step to either a partitioning of the set $\mathcal{P}$ or to a point from $\mathcal{P}$ falling onto a cell boundary, one can show that the number of cells is at most $2^d|\mathcal{P}| + 1$; see [1] for details. (This bound is slightly different from the bound reported by de Berg because he further subdivides the L-shapes into d hyperrectangles to obtain a BSP decomposition.) We obtain the following lemma.

Lemma 2.1. (de Berg [1]) *Let $\mathcal{P}$ be a set of points in $\Re^d$, and let $\sigma_\mathcal{P}$ be a hypercube containing all points from $\mathcal{P}$. Then there exists a partitioning of $\sigma_\mathcal{P}$ into $O(|\mathcal{P}|)$ hypercubes and L-shapes without points from $\mathcal{P}$ in their interior.*

3. Guarding sets

A guarding set for a collection of objects is, loosely speaking, a set of points that approximates the distribution of the objects. More precisely, guarding sets are defined as follows.

Definition 3.1. *Let $\mathcal{O}$ be a set of objects in $\Re^d$, let $\mathcal{R}$ be a family of subsets of $\Re^d$ called* ranges, *and let κ be a positive integer. A set $\mathcal{G}$ of points is called a* κ-guarding set for $\mathcal{O}$ against $\mathcal{R}$, *if any range from $\mathcal{R}$ not containing a point from $\mathcal{G}$ in its interior intersects at most κ objects from $\mathcal{O}$.*

We often call the points in $\mathcal{G}$ *guards*, and refer to ranges not containing guards as *empty ranges*.

Let us give an example. Suppose the set $\mathcal{O}$ is a set of n pairwise disjoint discs in the plane, and that the family of ranges is the family of all axis-parallel squares. For a disc D, define G_D to be the set of the following five points: the center of D plus the topmost, bottommost, leftmost, and rightmost point of D. When a square σ intersects D, and σ does not contain a point from G_D in its interior, then D contains a vertex of σ. (We assume all geometric objects are open.) Hence, the set $G = \{G_D | D \in \mathcal{O}\}$ is a 4-guarding set of size $5n$ for $\mathcal{O}$ against the family of squares.

From now on, we assume that the family of ranges consists of all axis-parallel squares (in $\Re^2$) and all axis-parallel hypercubes (in $\Re^d$). This is justified by a theorem proven in [3] that implies that it does not matter (from an asymptotic point of view) whether we study guarding against axis-parallel hypercubes, against balls, or against convex fat objects.

We are especially interested in scenes for which there exists a κ-guarding set of size cn, where κ and c are small constants. We say that such scenes are *guardable*. Let $\mathcal{O}$ be a guardable scene and let $\mathcal{G}$ be a κ-guarding set for $\mathcal{O}$ of size $m = O(n)$. Then according to Lemma 2.1, it is possible to

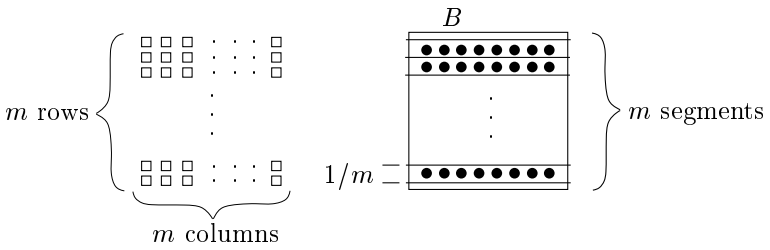


Figure 1: Guardable but not uncluttered

compute a set $\mathcal{S}$ of $O(m)$ empty hypercubes and L-shapes covering the scene. Any hypercube in $\mathcal{S}$ intersects at most κ objects from $\mathcal{O}$, and any L-shape in $\mathcal{S}$ intersects at most $(2^d-1)\kappa$ objects from $\mathcal{O}$ (since it can be covered by 2^d-1 hypercubes that are contained in it). Thus, loosely speaking, by cutting each of the hypercubes and L-shapes in $\mathcal{S}$ into a constant number of simple pieces, we obtain a linear-size BSP for $\mathcal{O}$; see [1]. This final phase however requires the objects to be either polyhedra of constant complexity or convex. We conclude that guardable d-dimensional scenes have a linear-size BSP, and therefore also a linear-size data structure for logarithmic-time point location (by using a balanced version of the underlying BSP tree [2]). In the next section we focus on the problem of computing small κ-guarding sets for guardable scenes. (We do not claim however that the best way to compute small BSPs for guardable scenes is by computing a small guarding set and applying de Berg's algorithm.)

Next we establish the relation between guardable scenes and uncluttered scenes, and between guardable scenes and scenes with small simple-cover complexity. (Recall that an uncluttered scene has small simple-cover complexity, but there exists scenes with small simple-cover complexity that are not uncluttered.) A scene is κ-cluttered if and only if the bounding-box vertices of the objects form a κ-guarding set (this is the definition of unclutteredness). Figure 1 shows that there exists scenes that are guardable but not uncluttered. Assume that we add to a given uncluttered scene $m = \sqrt{n}$ horizontal line segments, each of length $1+\varepsilon$, that do not intersect the bounding box of the original scene. (In Figure 1 the original scene consists of the $m \times m = n$ tiny cubes). Then the new scene is not uncluttered, since the unit square B is empty (of bounding-box vertices) and is intersected by m objects. However, by placing $O(m)$ equally-spaced guards between each pair of consecutive line segments, we obtain (together with the bounding-box vertices of the m^2+m objects) a linear-size κ-guarding set for the new scene, for some small constant κ. Thus the new scene is guardable.

Actually, a (somewhat surprising) theorem appearing in [3] shows that in the plane a scene is guardable if and only if it has small simple-cover complexity.

4. Computing guarding sets

Let $\mathcal{O}$ be a set of n objects in the plane, and assume that $\mathcal{O}$ has a finite κ-guarding set against axis-parallel squares. We describe three algorithms for computing a κ-guarding set, $\mathcal{G}$, for $\mathcal{O}$.

Let s_0 be a smallest bounding square of the input scene. All three algorithms construct a quad tree $\mathcal{T}$, through which a guarding set is computed. Each node of $\mathcal{T}$ represents a square that is contained in s_0, where the root represents s_0 itself. The collection of squares associated with the leaves of $\mathcal{T}$ forms a subdivision of s_0 into squares.

Algorithm I: In the first algorithm, $\mathcal{A}_1$, we construct the tree in the standard way, except that the stopping criterion is adapted to our purpose. Initially $\mathcal{T}$ consists of a single (root) node representing the bounding square s_0, and $\mathcal{G}$ is empty. Now, for a node v representing a square s_v, we check

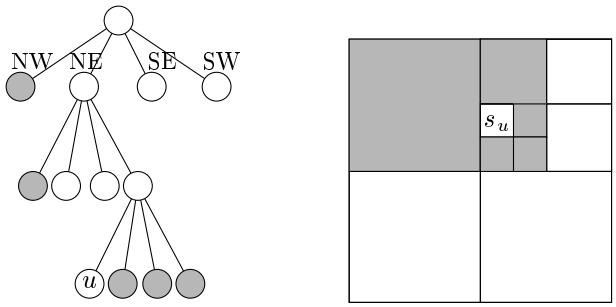


Figure 2: The neighborhood of a node u associated with square s_u is shown in grey

whether s_v is intersected by more than $\lfloor \kappa/2 \rfloor$ objects of $\mathcal{O}$. If the answer is negative, then we do not expand v; it becomes a leaf of $\mathcal{T}$, and we add the four corners of s_v to the guarding set $\mathcal{G}$. If the answer is positive, then we continue expanding the tree by creating four new nodes corresponding to the four quadrants of s_v and attaching them as the children of v.

Claim 4.1. *The set $\mathcal{G}$, computed by $\mathcal{A}_1$, is a κ-guarding set for $\mathcal{O}$ against squares.*

Proof. Let c be a square and assume that $c \cap \mathcal{G} = \emptyset$. If x is a corner of a square s associated with a leaf of $\mathcal{T}$, then x does not lie in the interior of c (since $x \in \mathcal{G}$). Therefore, c can be covered by at most two squares associated with leaves of $\mathcal{T}$. Since each of these at most two squares is intersected by at most $\lfloor \kappa/2 \rfloor$ objects of $\mathcal{O}$, we conclude that c is intersected by at most κ objects of $\mathcal{O}$. (Note that if c is not fully contained in s_0, then $c \cap s_0$ is covered by a single square associated with a leaf of $\mathcal{T}$, so, in this case, c is intersected by at most $\lfloor \kappa/2 \rfloor$ objects of $\mathcal{O}$.) ∎

Algorithm II: The second algorithm, $\mathcal{A}_2$, differs from $\mathcal{A}_1$ in (i) the stopping criterion and (ii) the rule by which points are added to $\mathcal{G}$. We stop expanding a node v associated with a square s_v, if s_v is intersected by at most $\lfloor \kappa/6 \rfloor$ objects of $\mathcal{O}$. The guarding points are the corners of the squares associated with the *internal* nodes of $\mathcal{T}$ (rather than the corners of the squares associated with the leaves, as in $\mathcal{A}_1$).

Claim 4.2. *The set $\mathcal{G}$, computed by $\mathcal{A}_2$, is a κ-guarding set for $\mathcal{O}$ against squares.*

Proof. Let c be a square and assume that $c \cap \mathcal{G} = \emptyset$. If s_v is a square associated with a leaf v of $\mathcal{T}$, then it is impossible that s_v is fully contained in (the interior of) c (because the corners of s_p, the square associated with the parent of v, are in $\mathcal{G}$, and s_v shares a corner with s_p). Moreover, it is easy to verify that the number of vertices of the subdivision of s_0 (formed by the squares associated with the leaves of $\mathcal{T}$) that lie in the interior of c cannot exceed 2. Therefore, since c is a square, it can be covered by at most six squares associated with leaves of $\mathcal{T}$. Each of these squares is intersected by at most $\lfloor \kappa/6 \rfloor$ objects of $\mathcal{O}$, hence c is intersected by at most κ objects of $\mathcal{O}$. ∎

Algorithm III: The third algorithm, $\mathcal{A}_3$, is completely different; it is based on the notion of *neighborhood*.

Definition 4.3. *A node v associated with square s_v is a* neighbor *of a node u associated with square s_u, if s_v is at least as large as s_u, and either an edge of s_u is contained in an edge of s_v, or s_v shares a vertex with s_u.*

Definition 4.4. *The* neighborhood *of a node u in $\mathcal{T}$ is the set of all its neighbors (see Figure 2).*

In the third algorithm, $\mathcal{A}_3$, the quad tree is not constructed in the standard way; it is constructed as follows. Initially, as in the standard way, $\mathcal{T}$ consists of a single node representing the bounding

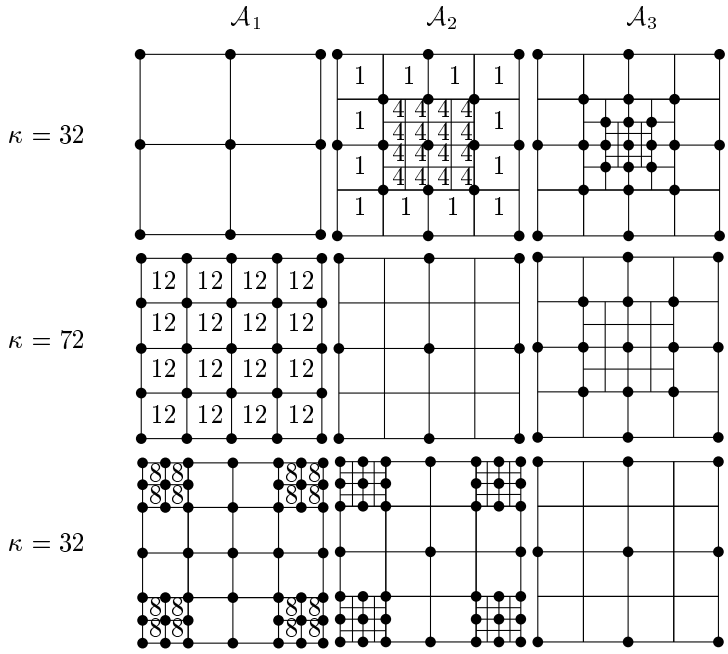


Figure 3: A 'winning' scene for each of the algorithms

square s_0 and $\mathcal{G}$ is empty. Now, for each node u in level i, associated with square s_u, we check whether the union of s_u and the squares associated with the nodes in the neighborhood of u is intersected by more than κ objects of $\mathcal{O}$. If so, we add the four corners of s_u to $\mathcal{G}$, and expand u by creating four new nodes corresponding to the four quadrants of s_u and attaching them as the children of u. After applying this check to all nodes in level i (in arbitrary order), we move to level $i+1$ and apply this check to all nodes in level $i+1$, and so on. We stop when a level is reached for which none of the nodes needs to be expanded. Note that all neighbors of a node u of level i are leaves of the current tree (i.e., the tree consisting of levels 0 to i), and their number is at most 8.

Claim 4.5. *The set $\mathcal{G}$, computed by $\mathcal{A}_3$, is a κ-guarding set for $\mathcal{O}$ against squares.*

Proof. Let c be a square and assume that $c \cap \mathcal{G} = \emptyset$. We know (see proof of Claim 4.2) that since the guarding points are the corners of the squares associated with the internal nodes of $\mathcal{T}$, c can be covered by at most six squares associated with leaves of $\mathcal{T}$. But, the neighborhood of one w (out of two) of these six leaves covers c. Since w is a leaf, its neighborhood is intersected by at most κ objects of $\mathcal{O}$ and therefore c is intersected by at most κ objects. ∎

Let us discuss the advantages and disadvantages of the three algorithms above. Figure 3 shows that for each of the algorithms there exists a scene for which it is better than the other two, in the sense that it produces a smaller guarding set. The left column corresponds to $\mathcal{A}_1$, the middle column to $\mathcal{A}_2$, and the right column to $\mathcal{A}_3$.

In the first scene that we consider, each of the quadrants of s_0 contains exactly $\kappa/2$ objects. In the top line of Figure 3, the three subdivisions obtained for such a scene (assuming $\kappa = 32$) are shown. The numbers in the middle subdivision are the numbers of objects intersecting the cells of the subdivision. (Their sum is greater than $n = 64$, since an object can intersect more than one cell of the subdivision.) In the left subdivision each of the quadrants contains exactly 16 objects. The appropriate numbers for the right subdivision are easily obtained from those for the middle

one, assuming each of the tiny cells is intersected by a single object. The size of the guarding set computed by $\mathcal{A}_1$ is 9, while the sizes for $\mathcal{A}_2$ and $\mathcal{A}_3$ are 17 and 25, respectively.

The second scene we consider consists of $16\kappa/6$ small objects that are distributed uniformally in s_0. Assuming $\kappa = 72$, the three subdivisions that are obtained for such a scene are shown in the middle line. The size of the guarding set computed by $\mathcal{A}_2$ is 9, while the sizes for $\mathcal{A}_1$ and $\mathcal{A}_3$ are 25 and 17, respectively.

The third scene consists of 4κ small objects that are concentrated near the corners of s_0. The bottom line corresponds to the three subdivisions that are obtained for such a scene, assuming $\kappa = 32$. The size of the guarding set computed by $\mathcal{A}_3$ is 9, while the sizes for $\mathcal{A}_1$ and $\mathcal{A}_2$ are 45 and 41, respectively.

Scenes such as triangulations in which there exist points that lie on the boundaries of several objects might be problematic, especially for the first two algorithms. We call such scenes *degenerate scenes*. For a degenerate scene, it is possible that one or more of these special points must be present in any κ-guarding set that is computed for the scene. (Otherwise we could place a small enough square around such a point that is both empty and is intersected by more than κ objects.) On the other hand, since the guarding points generated by our algorithms are corners of squares associated with nodes of a quad tree, it is possible that these special points will never arise as potential guarding points.

The advantage of $\mathcal{A}_3$, with respect to the above problem, becomes more evident when dealing with 3-dimensional scenes. The stopping criterion of $\mathcal{A}_1$ and $\mathcal{A}_2$ deteriorates, when moving to 3-space, while the stopping criterion of $\mathcal{A}_3$ does not change (although it becomes more difficult to find the set of neighbors of a node).

We omit from this version (theoretical) observations concerning the size of the guarding set that is computed by algorithm $\mathcal{A}_1$ (alternatively, $\mathcal{A}_2$). In the full version of this paper, we also discuss the effect of incorporating into our algorithms a so-called *shrinking step*, similar to the last step in de Berg's algorithm (described at the end of Section 2).

5. Experimental evaluation

We have implemented the three algorithms described in the preceding section, using the CGAL software library of geometric data structures and algorithms [5], and have performed various experiments in order to learn about their suitability in practice.

Our primary goal was to evaluate the algorithms, according to the sizes of the guarding sets that they produce. Recall that the size of the guarding set is closely related to the size of the BSP and data structures that are subsequently constructed for the input scene, assuming de Berg's algorithms [1, 2] are being used.

We also applied our algorithms to uncluttered scenes with clutter factor κ, and checked (i) whether the κ-guarding sets that are obtained tend to be smaller than $4n$ (recall that, by definition, the set consisting of all $4n$ bounding-box vertices is a κ-guarding set for such scenes), and (ii) assuming the answer is positive, what is the smallest value, $\kappa_0 \leq \kappa$, for which the scene still has a κ_0-guarding set.

We considered two types of scenes: polyhedral terrains and randomly generated collections of triangles. Polyhedral terrains are often used to represent pieces of the earth's surface in Geographic Information Systems. Most of the polyhedral terrain algorithms work with the terrain's projection on the xy-plane. Thus our terrain test scenes were generated from DEM files of certain areas in Canada and the U.S. as follows. A DEM file specifies the elevation of a set of sample points in the underlying area, where the sample points form a regular grid. Using the so-called VIP method [6] the m most important points were extracted for various values of m. The terrain test scene was then generated by computing the Delaunay triangulation of the extracted sample points.

The random scenes were generated by repeating the following step until the desired number of triangles was reached: Generate a random triangle within a fixed square, and, if it does not intersect

Death Valley						**San Bernardino**					
n	clutter factor	κ	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	n	clutter factor	κ	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$
202	25	72	62	147	56	202	16	60	49	119	40
		25	307	x	232			20	278	x	172
		24	307	x	251			16	x	x	271
		22	x	x	291			15	x	x	271
		20	x	x	328			13	x	x	364
		18	x	x	391			11	x	x	529
		17	x	x	412			10	x	x	677
		16	x	x	433	402	18	100	40	92	42
		14	x	x	492			66	75	198	74
		13	x	x	564			22	462	x	286
		12	x	x	685			18	x	x	399
		11	x	x	x			16	x	x	505
402	29	66	100	284	105			14	x	x	636
		29	427	x	307						
		22	610	x	439						
		20	x	x	485						

Table 1: Two Death Valley scenes and two San Bernardino scenes

Random scenes					
n	clutter factor	κ	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$
52	9	20	34	116	25
		9	184	1317	93
		8	184	1317	109
		7	315	1317	139
		6	315	1317	183
		5	669	x	265
100	9	12	159	525	110
		10	217	x	171
		9	307	x	195
		8	307	x	227
		7	601	x	264
		6	601	x	349
		5	1347	x	518
150	16	20	126	425	89
		16	173	1148	124
		12	315	1148	195
		10	401	x	240
		8	597	x	398
		6	1115	x	662

Table 2: Three random scenes

any of the triangles in the current collection of triangles, add it to this collection.

The results of some of our tests are presented in the following two tables. The left part of Table 1 corresponds to two "Death Valley" scenes, consisting of 202 and 402 triangles, respectively. The clutter factors of these scenes are 25 and 29, respectively. The right part of Table 1 corresponds to two "San Bernardino" scenes, and Table 2 to three random scenes. An 'x' entry in the column of algorithm $\mathcal{A}_i$ means that for the appropriate value of κ, $\mathcal{A}_i$ failed to produce a guarding set, before some halting condition was fulfilled (indicating that $\mathcal{A}_i$ would probably never terminate without the halting condition).

For each of the test scenes, we apply the three algorithms for various values of κ, beginning with rather high values and ending around the smallest value κ_0 for which one of the algorithms still succeeds in producing a guarding set. As expected (see discussion at the end of the previous section), the first two algorithms begin to fail once the values are less than $2\kappa_0$ and $6\kappa_0$, respectively.

Consider for example the first Death Valley scene (see Figure 4 left). Since the number of objects in this scene is 202 and the clutter factor is 25, we can obtain a 25-guarding set of size at most $4n = 808$, by taking all bounding-box vertices (see Figure 4 right). However, both $\mathcal{A}_1$ and $\mathcal{A}_3$ produce much smaller 25-guarding sets (see Table 1). For this scene, the value κ_0 is about 12, and algorithm $\mathcal{A}_3$ produces a 12-guarding set of size 685, which is still less than $4n$. Since κ_0 is about 12, it is not surprising that algorithm $\mathcal{A}_2$ fails for values below 24, and that algorithm $\mathcal{A}_3$ fails for even higher values. Figures 5–6 show both the guarding set and the partition computed by algorithm $\mathcal{A}_3$ for κ-values 25 and 16.

In general, the sizes of the guarding sets produced by $\mathcal{A}_2$ are much larger than the corresponding sizes of $\mathcal{A}_1$ and $\mathcal{A}_3$, and the sets produced by $\mathcal{A}_3$ are usually smaller than those produced by $\mathcal{A}_1$. In conclusion, algorithm $\mathcal{A}_3$ seems to perform quite well in practice.

All tests were performed on a SPARC Ultra-Enterprise machine. The running time for small scenes never exceeds a few seconds (e.g., $\mathcal{A}_1$ computed a 9-guarding set for the second random scene in 0.7 seconds), and for large scenes a few tens of seconds (e.g, $\mathcal{A}_3$ computed a 21-guarding set for a Death Valley scene consisting of more than 2000 objects in 50.77 seconds).

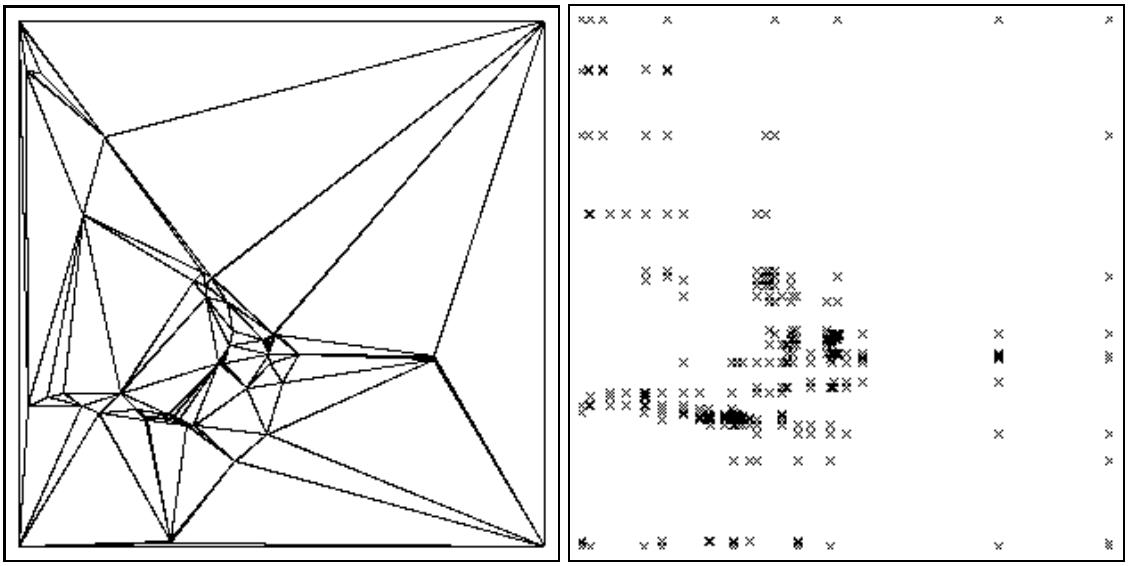


Figure 4: Death Valley (first scene) and the 25-guarding set consisting of all bounding-box vertices

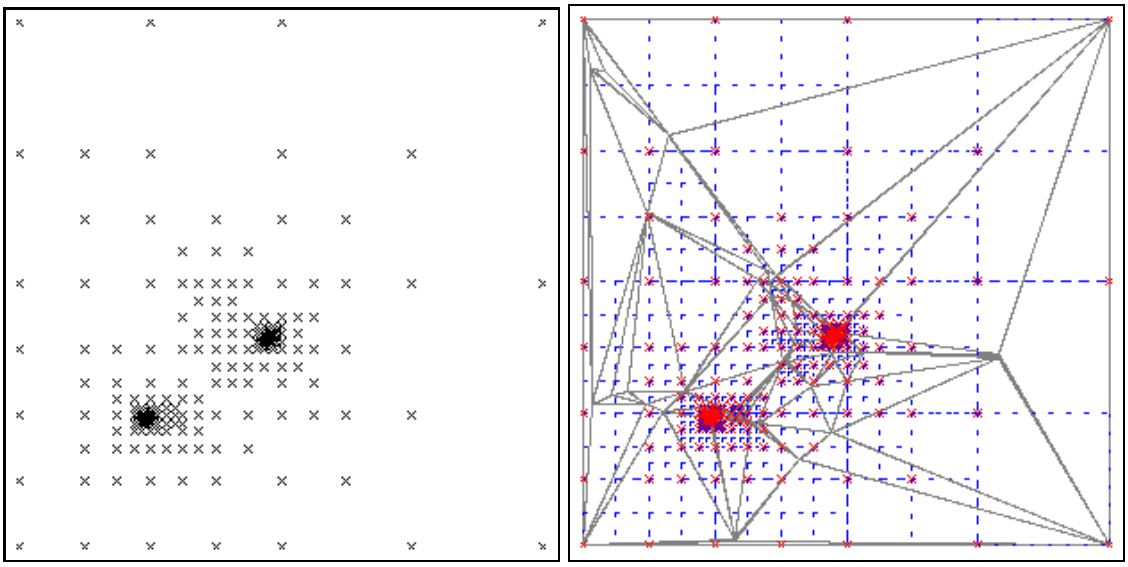


Figure 5: Left: A 25-guarding set computed by $\mathcal{A}_3$. Right: The scene together with the partition computed by $\mathcal{A}_3$ for $\kappa = 25$

References

[1] M. de Berg. Linear size binary space partitions for fat objects. In *Proc. 3rd Annu. European Sympos. Algorithms*, volume 979 of *Lecture Notes Comput. Sci.*, pages 252–263, Springer-Verlag, 1995.

[2] M. de Berg. Linear size binary space partitions for uncluttered scenes. Technical report UU-CS-1998-12, Dept. Comput. Sci., Utrecht Univ., 1998.

[3] M. de Berg, M. J. Katz, M. Overmars, A. F. van der Stappen, and J. Vleugels. Models and motion planning. In *Proc. 6th Scand. Workshop Algorithm Theory*, *Lecture Notes Comput. Sci.*, Springer-Verlag, 1998. To appear.

[4] M. de Berg, M. J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 294–303, 1997.

[5] CGAL Computational Geometry Algorithms Library. `www.cs.uu.nl/CGAL`.

[6] Z. Chen and J. A. Guevara. System selection of very important points (VIP) from digital terrain models for constructing triangular irregular networks. In *Proc. 8th Internat. Sympos. Comput.-Assist. Cartog. (Auto-Carto)*, pages 50–56, 1988.

[7] J. S. B. Mitchell, D. M. Mount, and S. Suri. Query-sensitive ray shooting. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 359–368, 1994.

[8] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. *J. Algorithms*, 21:629–656, 1996.

[9] O. Schwarzkopf and J. Vleugels. Range searching in low-density environments. *Inform. Process. Lett.*, 60:121–127, 1996.

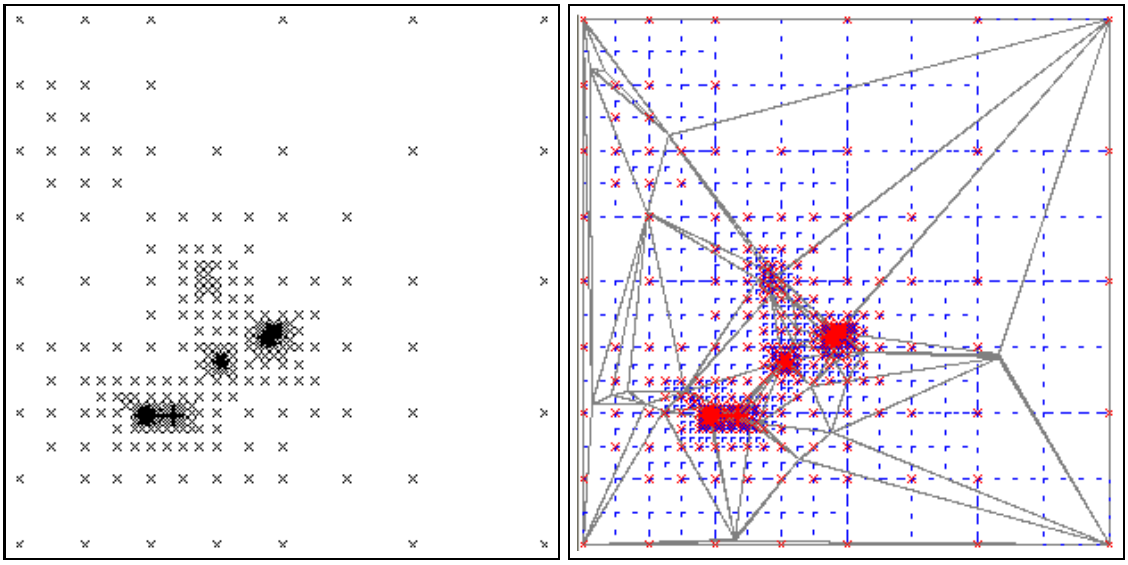


Figure 6: Left: A 16-guarding set computed by $\mathcal{A}_3$. Right: The scene together with the partition computed by $\mathcal{A}_3$ for $\kappa = 16$

[10] A. F. van der Stappen and M. H. Overmars. Motion planning amidst fat obstacles. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 31–40, 1994.

[11] A. F. van der Stappen, M. Overmars, M. de Berg, and J. Vleugels. Motion planning in environments with low obstacle density. Technical report UU-CS-1997-19, Dept. Comput. Sci., Utrecht Univ., 1997.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 121–132

Computing maximum-cardinality matchings in sparse general graphs

John D. Kececioglu[1]
A. Justin Pecqueur
Department of Computer Science, The University of Georgia
Athens, Georgia 30602-7404, USA
e-mail: {kece,andre}@cs.uga.edu

ABSTRACT

We give an experimental study of a new $O(mn\,\alpha(m,n))$-time implementation of Edmonds' algorithm for a maximum-cardinality matching in a sparse general graph of n vertices and m edges. The implementation incorporates several optimizations resulting from a depth-first order to search for augmenting paths, and we study the iteraction between four heuristics, each with the potential to significantly speed up the code in practice, through experiments with all sixteen possible variants. The experiments indicate that the simplest heuristic, an early-termination test for the depth-first search, results in the greatest performance gain, and yields an implementation that on graphs with large degree actually finds an optimal solution in less time than a standard greedy heuristic. The resulting code appears to be the fastest among those publicly available on the classes of random, k-regular, union-of-k-cycle, and Euclidean k-nearest-neighbor graphs for tests with up to 100,000 vertices and 500,000 edges with average degree from 1 to 10, achieving a maximum speedup of 50 over the two LEDA codes, and 4 and 350 over two of the DIMACS implementation challenge codes, while never taking longer than these implementations.

1. Introduction

One of the classic problems of combinatorial optimization is maximum-cardinality matching in general graphs [15]. A *matching* of an undirected graph $G = (V, E)$ is a subset of the edges $M \subseteq E$ such that no two edges in M touch a common vertex. A *maximum-cardinality* matching is a matching with the maximum number of edges.

Among the fundamental polynomial-time results in combinatorial optimization is Edmonds' algorithm for maximum-cardinalty matching in general graphs [8]. This paper gives an experimental study of a new $O(mn\,\alpha(m,n))$-time implementation of Edmonds' algorithm on large sparse graphs with n vertices and m edges. Our motivation comes from the problem in computational biology of large-scale DNA sequence assembly [14], which can be approximated in the presence of error using maximum-weight matchings in sparse nonbipartite graphs [13]. For this problem, our ultimate goal is an implementation of the $O(mn \log n)$-time *weighted*-matching algorithm of Galil, Micali and Gabow [12], but to understand this intricate algorithm at the level of detail necessary to produce a good implementation, we found ourselves forced to first understand the special case of *cardinality*-matching, and hence the present initial work of implementing an efficient, sparse, cardinality-matching algorithm.

There have been several studies of cardinality-matching algorithms, notably from the first DIMACS algorithm implementation challenge [6, 17]. Crocker [6] and Mattingly and Ritchey [17] give

[1]Research supported in part by a National Science Foundation CAREER Award, Grant DBI-9722339.

implementations of the $O(m\sqrt{n})$-time algorithm of Micali and Vazirani [20] and study its performance experimentally. Rothberg [25] gives an implementation of Gabow's $O(n^3)$-time version [10] of Edmonds' algorithm. LEDA, a library of combinatorial and geometric data structures and algorithms designed by Mehlhorn and Näher [18], provides an $O(mn\,\alpha(m,n))$-time implementation [19] of Edmonds' algorithm. Möhring and Müller-Hannemann [22] and Magun [16] experimentally study several $O(m+n)$-time heuristics for cardinality matching.

In the next section we sketch Edmonds' algorithm. Section 3 follows with a discussion of our implementation, highlighting the optimizations and heuristics that we examined. Section 4 presents results from experiments on random and structured graphs with our implementation and those of Crocker [6], LEDA [19], and Rothberg [25], which suggest that the new code is among the fastest available for large sparse graphs. Section 5 summarizes our results.

2. Edmonds' algorithm

We now briefly sketch Edmonds' algorithm [8]. As the algorithm is involved, we defer to readers familiar with its basic implementation to meet space guidelines. Good expositions may be found in Tarjan [27], Ahuja, Magnanti and Orlin [1], and Moret and Shapiro [21].

A vertex v is said to be *unmatched* with respect to matching M if no edge of M touches v. An *alternating path* in G with respect to a matching M is a path whose edges alternate between being in and out of M. An *augmenting path* is an alternating path that begins and ends at an unmatched vertex. A *blossom* is an alternating path that forms a cycle of odd length; note that on any such cycle there must be a vertex incident to two unmatched edges on the cycle; this vertex is called the *base* of the blossom.

The essential step of the algorithm is to find an augmenting path with respect to a current matching M, which may initially be the empty matching. If M has an augmenting path, its cardinality can be increased by one, while if M has no augmenting path, it is optimal [15]. A *phase* of the algorithm consists of searching for an augmenting path, and terminates on finding such a path or determining that there is none.

An augmenting path may be found by exploring along search trees rooted at unmatched vertices. As edges at successive levels of these trees alternate between being in and out of M, they are called *alternating trees*. Vertices at even depths from the roots are called *even*, and vertices at odd depths are called *odd*. During the search a blossom may be discovered, at which point all vertices on the cycle are shrunk into a single *supervertex* that is identified with the base of the blossom. The key observation of Edmonds is that the shrunken graph has an augmenting path if and only if the original graph has one [8].

With suitable data structures, a phase can be implemented to run in $O(m\,\alpha(m,n))$ time, which Tarjan [27] credits to Gabow, where $\alpha(m,n)$, a very slowly-growing inverse of Ackermann's function, is the amortized time per operation for the disjoint-set data structure [27]. As there are at most $n/2$ phases, this gives an $O(mn\,\alpha(m,n))$-time algorithm. Due to the form of the set unions performed by the algorithm, it is possible in theory to reduce the factor of $\alpha(m,n)$ in the running time to $O(1)$ using the disjoint-set result of Gabow and Tarjan [11]; the resulting $O(mn)$ time algorithm is unlikely to be better in practice, however, since for all conceivable inputs, already $\alpha(m,n) \leq 4$.

3. Implementation

The data structures in our $O(mn\,\alpha(m,n))$-time implementation of Edmonds' algorithm are as follows. Edges in the undirected graph G are assigned an arbitrary orientation, so that each undirected edge is represented by one directed edge that may be traversed in either direction. We access the neighborhood of a vertex by maintaining with each vertex a list of in-edges and out-edges. When detecting a blossom, the graph is not actually shrunk, but the partition of vertices in the original

graph into blossoms is maintained via the disjoint-set data structure; we use the disjoint-set path-halving variant of Tarjan and van Leeuwen [27]. The recursive structure of blossoms is recorded using the bridge representation described by Tarjan [27].

Perhaps the most significant choice for an implementation is the order in which to explore edges of the graph when searching for an augmenting path. We chose to explore them in *depth-first search* order, so that we grow alternating trees one at a time until we discover an augmenting path or that there is none starting from the current root. This choice permits several optimizations in the code, some of which are not discussed in standard presentations of Edmonds' algorithm [27, 1], and which we detail next.

3.1. Optimizations

The implementation incorporates the following three ideas. As each is guaranteed to speed up the code, we call them *optimizations*. Later we examine four more ideas which may or may not speed up the code, and which we call *heuristics*.

3.1.1. One-pass shrinking

A key advantage of examining edges in depth-first search order is that on encountering an edge $e = (v, w)$ between two even-labeled vertices, it is not hard to show that e always forms a blossom, and furthermore that one of v and w must be an ancestor of the other in the alternating tree. Thus the algorithm can avoid performing the standard interleaved walk to (1) determine whether e forms a blossom or an augmenting path and (2) find the nearest common ancestor of v and w in the alternating tree if e forms a blossom.

To identify the ancestor we assign an *age* to each vertex from a global counter that is incremented as vertices are reached by the depth-first search. The nearest common ancestor of v and w is the younger of the two, so we can shrink the discovered blossom while walking directly to the ancestor in one pass.

3.1.2. Avoiding expansion and relabeling of unsuccessful trees

On returning from a search of an alternating tree that does not lead to an augmenting path, we leave all blossoms in the unsuccessful tree shrunken, and the labels of all vertices in the tree in their current state: Since the depth-first search is exhaustive, no future augmentations will ever pass through the tree. Tarjan [27] credits this observation to Edmonds; in the context of bipartite matching, Chang and McCormick [4] and Cherkassky, Goldberg, Martin, Setubal and Stolfi [5] study this heuristic, and others, experimentally.

3.1.3. Finding the next search root quickly

On completing the search of an alternating tree, the next root for a search must be found. Instead of scanning the vertices of the graph to find an unreached unmatched vertex for the next search root, we maintain one list across all phases of the algorithm of unreached unmatched vertices. When an unmatched vertex is first reached by a search, we unlink it from this list; the vertex will never be placed back on the list, since if it is reached on an unsuccessful search it will always remain reached and unmatched, while if it is reached on a successful search it will be relabeled as unreached but will always remain matched. To find the next search root, we pop the next vertex from this list.

3.2. Heuristics

We now describe four heuristics that have the potential to significantly speed up the algorithm in practice.

3.2.1. Initializing with a greedy matching

While Edmonds' algorithm is usually described as starting from the empty matching, it can be started from any matching. The conventional wisdom is that starting from a near-optimal matching should speed up the algorithm in practice since this reduces the number of augmentations. (In truth, though, the situation is unclear, since after performing as many augmentations as there are edges in the initial matching, the algorithm started from the empty matching proceeds for the final, difficult-to-find augmentations on a contracted graph, which could be preferable to proceeding from the initial matching on an uncontracted graph.)

We consider starting from an initial matching obtained by the following standard heuristic [21]. Form a maximal matching by repeatedly selecting a vertex v that has minimum degree in the subgraph $\widetilde{G}$ induced by the currently unmatched vertices, and select an edge (v, w) to an unmatched vertex w that has minimum degree in $\widetilde{G}$ over all vertices incident to v. This greedy procedure can be implemented to run in $O(m + n)$ time using a discrete, bucketed heap of unmatched vertices prioritized by degree in $\widetilde{G}$. Since the resulting matching is maximal, it is guaranteed to have at least half the edges of a maximum matching, and as shown by Shapira [26], the size of the smallest matching found by this heuristic over all graphs on n vertices and m edges is the size of the smallest maximum matching on a graph with n vertices and m edges.

3.2.2. Stopping successful searches early

In the depth-first search, unexplored edges are pushed onto a search stack and later popped as they are explored. Edges are pushed when first encountering a vertex that gets labeled even, or when shrinking a blossom into an even supervertex. In both situations, when pushing an edge e we can test whether its other end touches an unreached unmatched vertex. If so, edge e completes an augmenting path, and pushing other unexplored edges onto the stack will simply bury e; if instead we leave e exposed on top of the stack, the next iteration of the search will pop e and immediately discover the augmenting path. This simple *stopping test* can halt a potentially lengthy successful search early. In the context of bipartite matching, Chang and McCormick [4] appear to have also used this heuristic, which they call "look-ahead," and attribute to Duff [7].

3.2.3. Delayed shrinking of blossoms

We also consider a suggestion of Applegate and Cook [2], originally made in the context of weighted matchings. Since the algorithm performs work when shrinking and later expanding blossoms, it may be worth postponing the formation of blossoms for as long as possible. Before pushing unexplored edges onto the search stack, we can test whether or not an edge forms a blossom. Edges that do not form blossoms are given precedence and placed on the stack above edges that create blossoms. We implement this by maintaining a list of cycle-forming and a list of non-cycle-forming edges when scanning the neighborhood of a vertex; after completing the scan, these two lists are concatenated onto the search stack in the appropriate order.

3.2.4. Lazy expansion of blossoms

As observed by Tarjan [27], on finding an augmenting path P in a successful search of an alternating tree T, the only blossoms we need to immediately expand and relabel are those on path P. Blossoms in T not on path P can remain shrunken; when a later search encounters a vertex in a blossom B of T that is not on P, B can at that moment be expanded lazily and its vertices treated as having been labeled unreached.

We implement this idea as follows. On an unsuccessful search, we delete all vertices in the alternating tree from the graph. At the start of a new search, we record the age that will be assigned to the next vertex that is reached, which we call the current *epoch*. When a vertex is encountered during a search, we first examine its age. If its age is from a prior epoch, we first lazily expand its blossom, consider its members as having been labeled unreached, and proceed as before.

4. Experimental results

We now study the performance of implementations resulting from different combinations of the heuristics of Section 3.2, together with several publicly available codes from other authors, through experiments on random and structured graphs.

4.1. Implementations

In our experiments we tested the following implementations.

- An $O(mn\,\alpha(m,n))$-time implementation, written in C by the first author, of the general approach described by Tarjan [27] with the optimizations described in Section 3. To decide which of the heuristics of Section 3 to incorporate, sixteen variants of this basic implementation were written, comprising all 2^4 combinations of the four heuristics. After testing these variants as described below, the variant with the most robust combination of heuristics overall was selected (namely, start from the greedy initial matching and use the stopping test). In the experiments, this implementation is called `Kececioglu`, and may be accessed at `http://www.cs.uga.edu/~kece/Research/software.html`.

  The implementation is part of an object-oriented library of fundamental string and graph algorithms being developed by the first author, called DALI for "a **d**iscrete **a**lgorithms **li**brary." DALI, which will be freely released, currently contains general implementations of several commonly-used data structures, including lists, multidimensional arrays, search trees, hash tables, mergeable heaps, disjoint sets, and undirected and directed graphs, as well as efficient algorithms for shortest paths, minimum spanning trees, maximum flow, minimum unrestricted cut, maximum weight branchings, and nearest common ancestors; under development are implementations of suffix trees, splitable heaps, and maximum-weight matchings. DALI's design emphasizes code reusability, portability, and efficiency; the implementation, written in C, is designed to be lightweight, with low operation overhead and small object-code size, while presenting a uniform, consistent interface.

  The maximum-cardinality matchings code, with comments, comprises about 1200 lines of C, and is built on top of a general list library of roughly 650 lines, a disjoint-set library of roughly 350 lines, and a directed graph library of roughly 1150 lines.

  While the libraries are not designed with space efficiency in mind, to give an idea of the space consumption DALI uses 3 words per list, 3 words per list element, 3 words per disjoint-set element, 11 words per graph vertex, and 15 words per graph edge. The cardinality-matchings code uses at most an additional 17 words per vertex, and at most an additional 3 words per edge, for a total of 112 bytes per vertex and 72 bytes per edge.

  To a limited extent DALI performs memory management by maintaining for each dynamically allocated datatype a *pool* of free objects; when a new object is requested and the corresponding pool is empty, a whole block of objects of that type is allocated with one call to `malloc`, and all the objects in the block are added to the pool; this reduces the number of calls to `malloc` and `free`.

- *Stefan Näher's* $O(mn\,\alpha(m,n))$-time implementation [19] of Tarjan's approach [27], written in C++ as part of the LEDA library of efficient data structures and algorithms developed by Mehlhorn and Näher [18]. The implementation comes with two heuristics for constructing an initial matching: the first finds a maximal matching by examining vertices in arbitrary order, and the second, attributed to Markus Paul, examines vertices in order of decreasing degree in the input graph and attempts to find all augmenting paths of length at most three. In the experiments the two resulting codes are called `LEDA 1` and `LEDA 2`.

- *Steven Crocker's* $O(m\sqrt{n})$-time implementation [6], written in Pascal, of Micali and Vazirani's algorithm [20]. The code augments from the empty initial matching. In the experiments this implementation is called `Crocker`.

- *Ed Rothberg's* $O(n^3)$-time implementation [25], written in C, of Gabow's version [10] of Edmonds' algorithm. The code augments from a maximal initial matching found by examining vertices in arbitrary order. In the experiments this implementation is called `Rothberg`.

4.2. Graph generators

These codes were compared by the second author [23] across five classes of graphs.

- *Random graphs* on n vertices with m edges. To test the codes on large-scale inputs it is important to generate a random graph in $O(m+n)$ space. To do this we generated a random subset of size m from $\{1,\ldots,\binom{n}{2}\}$ using Floyd's algorithm [3], translating the chosen integers into unordered pairs of vertices by a bijection. Since for large n, $\binom{n}{2}$ can easily exceed the machine representation, we also implemented an arbitrary integer arithmetic package. The resulting generator allowed us to generate large sparse random graphs in $O(n+m\log n)$ time in $O(m+n)$ space.
- *Union-of-k-cycle graphs* on n vertices with at least m edges. For a given n, m, and k, form a graph on n vertices by repeated choosing a random subset of k vertices, connecting them with a random cycle, eliminating parallel edges, and taking the union of such cycles until the graph contains at least m edges. (The number of edges in the resulting graph is between m and $m+k-1$ inclusive.) Since blossoms are odd-length cycles and create work for augmenting path algorithms, we considered $k = 3, 5, 7, 9, 11$, and 21 in our experiments; the results we report are for $k = 3$, as this produced the hardest instances for the codes.
- *Near-regular graphs* of degree at most k on n vertices. For a given n and k, form a graph on n vertices as follows. Maintain an array of length n representing a list of vertices and their unused degree-capacity. Initialize all vertices in the array to degree-capacity k. Then repeatedly pick the leftmost vertex v from the array, delete it, and generate a random subset of size $\min\{k', n'\}$ over the remaining vertices, where k' is the unused capacity of v and n' is the number of remaining vertices. Add an edge from v to each vertex in the subset, decrement the capacities of these vertices, and delete from the list any vertex with capacity zero.

  In contrast to the standard regular-graph generator, the graphs generated by this procedure do not contain self-loops or parallel edges. On the other hand, they are not guaranteed to be k-regular, though there are at most k vertices in the generated graph with degree less than k. The generator can be implemented to run in $O(nk\log k)$ time using $O(n)$ working space.
- *Euclidean nearest-neighbor graphs* of degree k on n points in the plane. For a given k, we formed a graph by choosing a point set from Reinelt's TSPLIB library [24] of Euclidean traveling salesman problems, and connecting each point to its k nearest neighbors.
- *Gabow's graph* [10] on $6n$ vertices. This dense graph is a worst-case input constructed by Gabow for his $O(n^3)$-time version Edmonds' algorithm augmenting from an empty matching and examining vertices in numerical order. The graph is the union of the complete graph K_{4n} with a matching connecting the odd-numbered vertices of K_{4n} to the empty graph N_{2n}.

4.3. Experiments

We now present results from experiments on these five classes of graphs. All experiments were run on a dedicated Sun Ultra 1 workstation with 256 Mb of RAM and a 200 MHz processor. In the experiments on random graphs, each data point represents an average over ten graphs, where care was taken to perform comparisons across codes on the same ten inputs. Times reported are user times in seconds obtained by the UNIX `time` command under the Bourne shell. The time measured is the amount of time taken to read in the graph, compute a matching, and output its cardinality. C codes were compiled under `gcc` version 2.7.2.1 with the `-O3` option, and C++ codes were compiled under `g++` version 2.7.2.1 with the `-O` option.

To meet space guidelines, we only display results from experiments on the largest graphs; plots of all experiments are available at `http://www.cs.uga.edu/~kece/Research/papers/KP98.ps.Z`.

4.3.1. Combining heuristics

In the first set of experiments, shown in Figure 1, we tested the sixteen variants of our implementation on random graphs with 5,000 to 50,000 vertices, and average degree 1 to 20. Figure 1 shows results with $n = 50,000$. The top four plots divide the sixteen variants into four large groups of four variants each, according to whether they start from a greedy initial matching (`G`) or use the stopping test for a successful search (`S`). From each of the four groups we selected a variant with the best running time and these four winners are displayed in the bottommost plot, except that we do not include the winner from the initial group not containing the `G` and `S` heuristics, as this group's times are much slower than all other groups; instead we include two representatives, `G` and `GD`, from the second group.

The most striking feature of Figure 1 is that incorporating just the `G` or `S` heuristic alone gives a speedup of a factor of 30 to 35 on the largest graphs. The improvement with the `G` variant supports the conventional wisdom that to speed up an exact matching procedure one should start from an initial approximate matching. The next most striking feature of Figure 1 is that on graphs with large degree, *it is faster to start from an empty matching and use the stopping test than to start from a greedy matching* (which can be seen by comparing the `S` and `G` curves in the bottom plot). It came as a surprise that the greedy heuristic actually slows down the `S` variant (which can be seen by comparing the `S` and `GS` curves in the bottom plot), especially since this heuristic runs in $O(m+n)$ time. To understand this better we compared the `S` variant, which finds an exact matching, to the greedy heuristic alone, which finds an approximate matching, and found that the exact `S` code was in fact faster: the greedy heuristic must examine every edge in the graph to compute the degree of each vertex, while with the stopping test the `S` code could avoid looking at every edge on graphs with higher degree. Nevertheless, of the sixteen variants, we chose the `GS` variant as the overall winner for further comparison (even though it sometimes loses to the `S` variant) as the `GS` code was the most robust across a wide range of vertex degrees. In the experiments that follow, this `GS` code is called `Kececioglu`.

4.3.2. Comparing codes

Figures 2 and 3 compare this `GS` code to `LEDA`, `Rothberg`, and `Crocker` across four classes of random and nonrandom graphs with number of vertices n fixed and number of edges m varying. We performed experiments on the random classes with n varying from 1,000 to 100,000, but only display results for the largest graphs; performance on the smaller graphs is similar.

The plots on the left display all codes together, while the plots on the right display only the two fastest codes for easier visual comparison. On the random graphs, the fastest code is `Kececioglu`, followed by `Crocker`. Its speedup compared to other codes generally increases with graph size (except compared to `LEDA 2`, which improves on larger graphs), with a maximum speedup of 50 versus `LEDA 2`, 350 versus `Rothberg`, and 4 versus `Crocker`. The near-regular graphs appear to be somewhat easier than the corresponding random graphs for the same n and m. On the union-of-3-cycle graphs, it is interesting that the times for `Rothberg` and `Crocker` increase with graph size, while the times for `LEDA` and `Kececioglu` (which both implement the approach of Tarjan [27]) after a critical size tend to remain constant or decrease.

Table 1 gives variances in running times on random graphs. As suggested by the prior plots, the variance of `Rothberg` is quite high, while the variance of `Crocker` appears to be generally the lowest, with `Kececioglu` roughly comparable.

Figure 3 compares the codes on Euclidean nearest-neighbor graphs derived from TSPLIB problems [24] on roughly 12,000 and 34,000 points. These graphs appear somewhat harder than the corresponding random graphs with the same m and n, but the ranking of codes is essentially unchanged. (On the nearest-neighbor graphs, `Crocker` crashed on problem `pla33810.tsp` with 9 neighbors; for

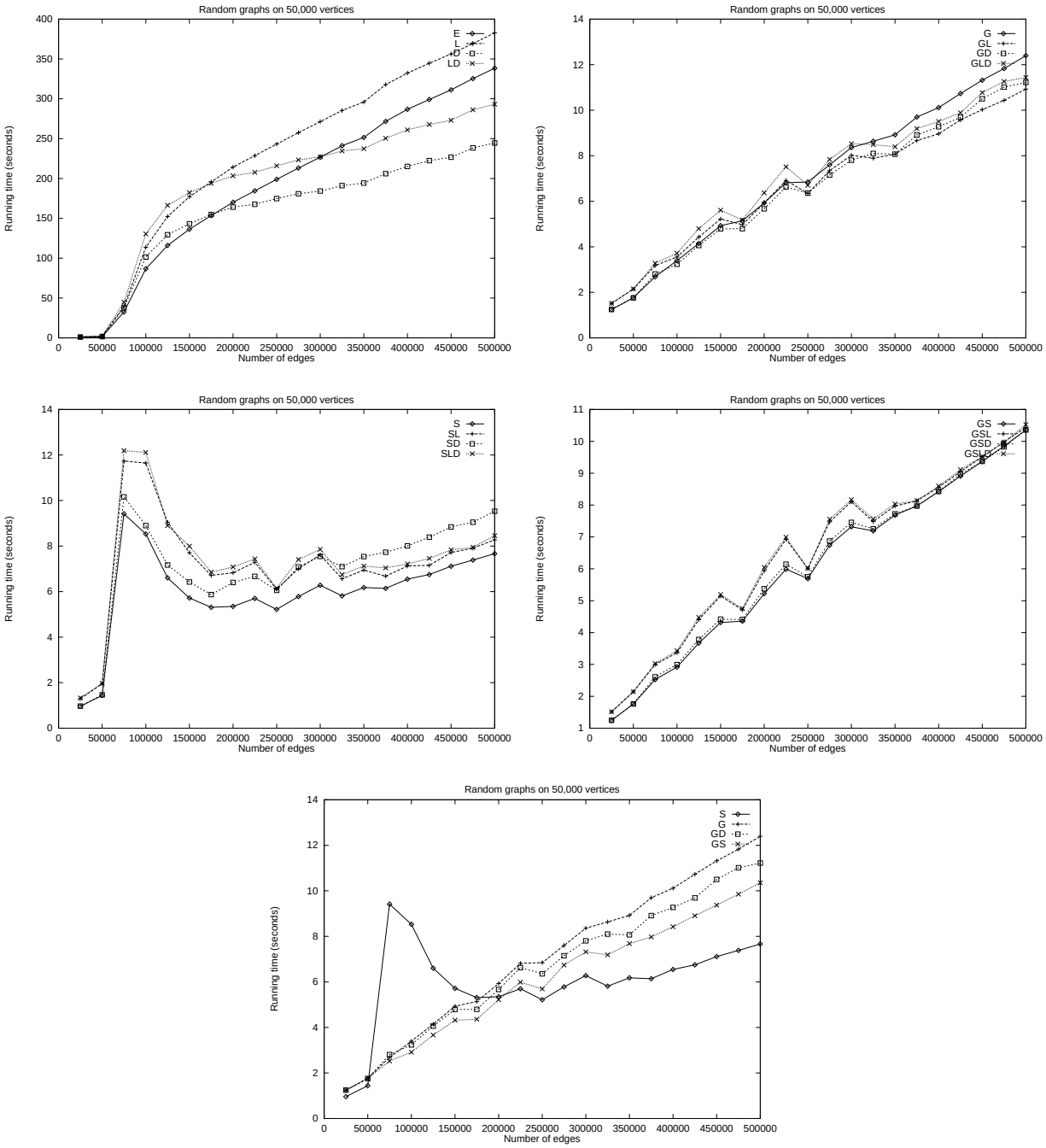


Figure 1: Running times for all combinations of heuristics on random graphs on 50,000 vertices. `E` is for no heuristics, `L` is for lazy expansion, `D` is for delayed shrinking, `G` is for greedy initial matching, and `S` is for the stopping test. The string labeling a curve denotes the combination of heuristics used by the variant.

this input, Figure 3 reports the time to failure.) We also performed experiments on the Gabow worst-case graphs, which are shown in the fuller version of this paper referenced earlier; the ranking of codes is similar.

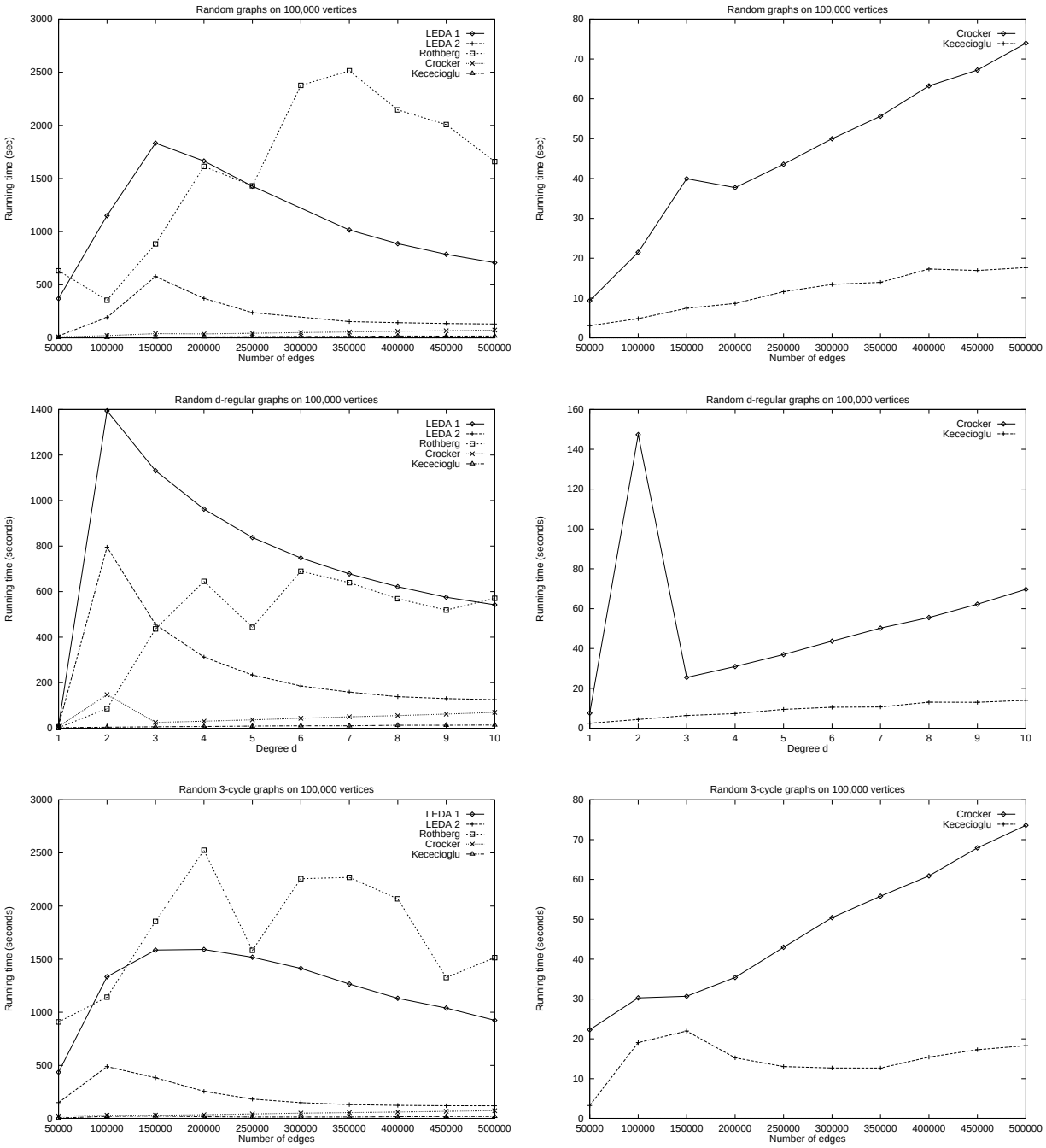


Figure 2: Running times for all codes on random, near-regular, and union-of-3-cycle graphs on 100,000 vertices with number of edges varying.

Figure 4 plots results for experiments on random and union-of-k-cycle graphs with number of edges m fixed and number of vertices m varying. Again the general ranking of codes observed above tends to continue, with `Kececioglu` and `Crocker` remaining the fastest.

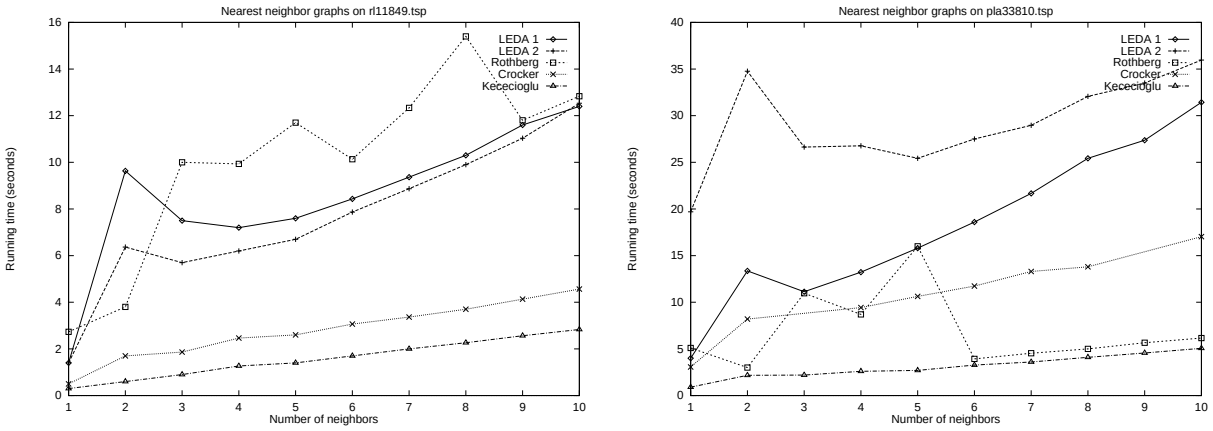


Figure 3: Running times for all codes on nearest-neighbor graphs generated from TSPLIB Euclidean traveling salesman problems.

Table 1: Mean and standard deviation of running time on random graphs of 100,000 vertices. Times are in seconds, and are computed for 10 trials.

	LEDA 1		LEDA 2		Rothberg		Crocker		Kececioglu	
edges	mean	dev	mean	dev	mean	dev	mean	dev	mean	dev
50,000	436.21	10.92	149.90	7.88	909.07	1.50	22.27	2.35	3.27	0.10
100,000	1334.85	16.11	489.09	12.27	1142.09	313.78	30.29	1.16	19.06	2.39
250,000	1518.15	16.70	183.02	5.87	1583.15	1783.58	42.98	2.08	13.05	2.88
500,000	924.66	10.88	119.83	3.32	1513.85	2142.98	73.59	4.15	18.30	6.14

5. Conclusion

We have presented computational experience with a new $O(mn\,\alpha(m,n))$-time implementation of Edmonds' algorithm for maximum-cardinality matching in sparse general graphs, studying several heuristics for speeding up the code in practice. The new code with the greedy matching and stopping test heuristics appears to be among the fastest currently available on large sparse graphs for several classes of random and nonrandom inputs, achieving a maximum speedup of 4 to 350 compared to other published codes. Interestingly, the greatest performance gains were obtained not by the more sophisticated heuristics but by incorporating a simple early-termination test for successful augmenting-path searches. On graphs with large degree, the resulting exact-matching code started from an empty matching was even faster than a linear-time approximate-matching heuristic, and hence was actually slowed down when started from the corresponding near-optimal matching; on graphs with small degree, however, the stopping test by itself gave uneven behavior, and the most robust variant across all degrees was a combination of the greedy matching and stopping test heuristics.

References

[1] R.K. Ahuja, T.L. Magnanti and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1993.

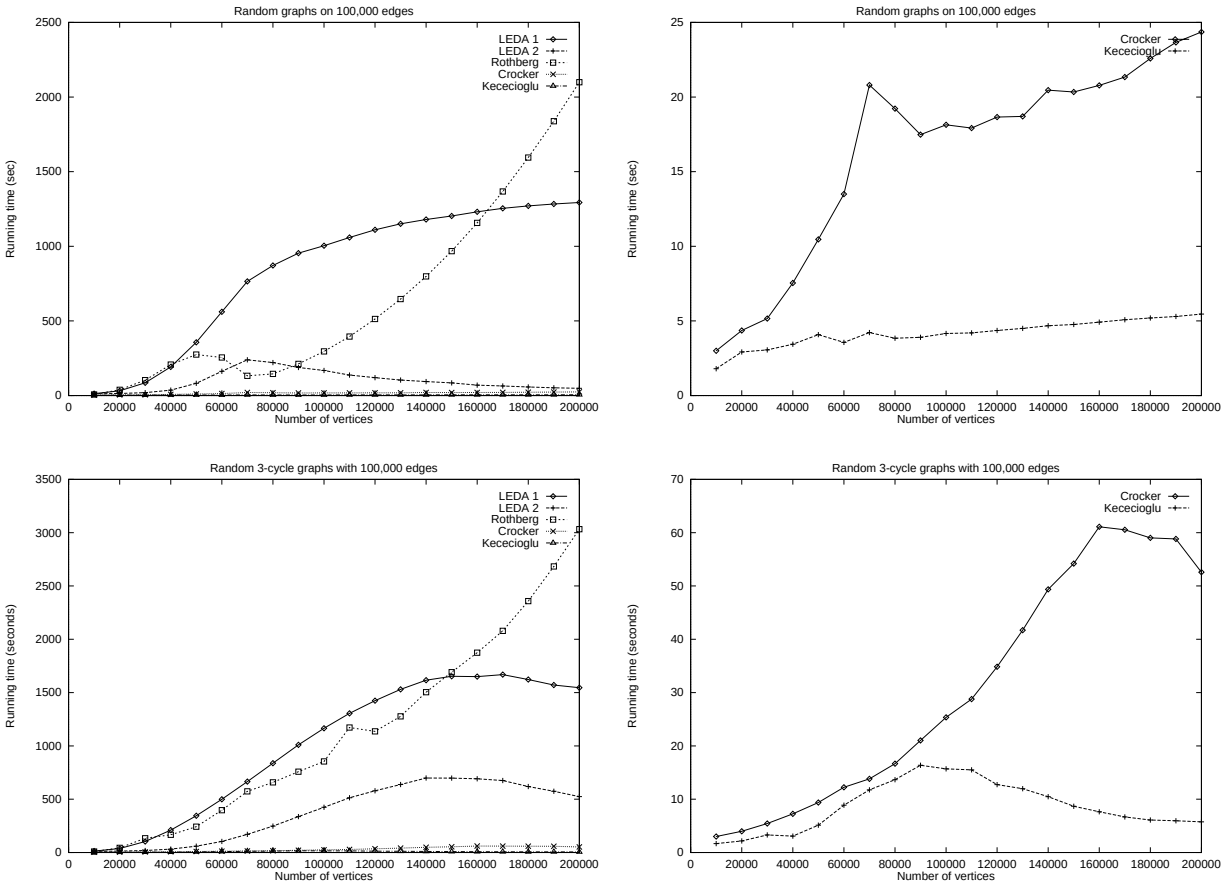


Figure 4: Running times for all codes on random and union-of-3-cycle graphs with 100,000 edges and number of vertices varying.

[2] D. Applegate and W. Cook, *Solving large-scale matching problems*, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 12, 557–576, 1993.

[3] J. Bentley and R. Floyd, *Programming pearls: A sample of brilliance*, Communications of the ACM, 754–757, September 1987.

[4] S.F. Chang and S.T. McCormick, *A faster implementation of a bipartite cardinality matching algorithm*, Technical Report 90-MSC-005, Faculty of Commerce and Business Administration, University of British Columbia, January 1990.

[5] B.V. Cherkassky, A.V. Goldberg, P. Martin, S.C. Setubal and J. Stolfi, *Augment or push? A computational study of bipartite matching and unit capacity flow algorithms*, Proceedings of the 1st *Workshop on Algorithm Engineering*, 1–10, 1997. `http://www.dsi.unive.it/~wae97/proceedings`

[6] S.T. Crocker, *An experimental comparison of two maximum cardinality matching programs*, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 12, 519–537, 1993. `ftp://dimacs.rutgers.edu/pub/netflow/matching/cardinality/solver-2`

[7] I.S. Duff, *On algorithms for obtaining a maximum transversal*, A.E.R.E. Harwell Report CSS.49, Oxfordshire, England, 1977.

[8] J. Edmonds, *Paths, trees, and flowers*, Canadian Journal of Mathematics 17, 449–467, 1965.

[9] J. Edmonds, *Maximum matching and a polyhedron with* $0,1$*-vertices*, Journal of Research of the National Bureau of Standards 69B, 125–130, 1965.

[10] H. Gabow, *An efficient implementation of Edmonds' algorithm for maximum matchings on graphs*, Journal of the ACM 23, 221–234, 1976.

[11] H.N. Gabow and R.E. Tarjan, *A linear-time algorithm for a special case of disjoint set union*, Journal of Computer and System Sciences 30:2, 209–221, 1985.

[12] Z. Galil, S. Micali and H.N. Gabow, *An* $O(EV \log V)$ *algorithm for finding a maximal weighted matching in general graphs*, SIAM Journal on Computing 15, 120–130, 1986.

[13] J. Kececioglu, *Exact and Approximation Algorithms for DNA Sequence Reconstruction*, PhD dissertation, Technical Report 91-26, Department of Computer Science, The University of Arizona, December 1991.

[14] J.D. Kececioglu and E.W. Myers, *Combinatorial algorithms for DNA sequence assembly*, Algorithmica 13:1/2, 7–51, 1995.

[15] L. Lovász and M.D. Plummer, *Matching Theory*, Annals of Discrete Mathematics 29, North-Holland, 1986.

[16] J. Magun, *Greedy matching algorithms: An experimental study*, Proceedings of the 1st *Workshop on Algorithm Engineering*, 22–31, 1997. `http://www.dsi.unive.it/~wae97/proceedings`

[17] R.B. Mattingly and N.P. Ritchey, *Implementing an* $O(\sqrt{N}M)$ *cardinality matching algorithm*, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 12, 539–556, 1993. `ftp://dimacs.rutgers.edu/pub/netflow/matching/cardinality/solver-3`

[18] K. Mehlhorn and S. Näher, *LEDA: A platform for combinatorial and geometric computing*, Communications of the ACM 38:1, 96–102, 1995.

[19] K. Mehlhorn, S. Näher and S. Uhrig, *LEDA Release 3.4 module* `_mc_matching.cc`, Computer software, 1996. `http://www.mpi-sb.mpg.de/LEDA/leda.html`

[20] S. Micali and V.V. Vazirani, *An* $O(\sqrt{|V|} \cdot |E|)$ *algorithm for finding a maximum matching in general graphs*, Procedings of the 21st IEEE *Symposium on the Foundations of Computer Science*, 17–27, 1980.

[21] B.M.E. Moret and H.D. Shapiro, *Algorithms from P to NP*, Benjamin-Cummings, 1991.

[22] R.H. Möhring and M. Müller-Hannemann, *Cardinality matching: Heuristic search for augmenting paths*, Technical Report 439/1995, Fachbereich Mathematik, Technische Universität Berlin, 1995. `ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-439-1995.ps.Z`

[23] A.J. Pecqueur, *An Experimental Study of Edmonds' Algorithm for Maximum-Cardinality Matching in Sparse General Graphs*, M.S. thesis, Department of Computer Science, The University of Georgia, May 1998.

[24] G. Reinelt, *TSPLIB: A traveling salesman problem library*, ORSA Journal on Computing 3, 376–384, 1991.

[25] E. Rothberg, *Implementation of Gabow's* $O(n^3)$ *version of Edmonds' algorithm for unweighted nonbipartite matching*, Computer software, 1985. `ftp://dimacs.rutgers.edu/pub/netflow/matching/cardinality/solver-1`

[26] A. Shapira, *An exact performance bound for an* $O(m+n)$ *time greedy matching procedure*, Electronic Journal of Combinatorics 4:1, R25, 1997. `http://www.combinatorics.org`

[27] R.E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 133–142

A Network Based Approach for Realtime Walkthrough of Massive Models*

Matthias Fischer, Tamás Lukovszki, and Martin Ziegler
Heinz Nixdorf Institute and Department of Computer Science, University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
e-mail: {mafi, talu, ziegler}@uni-paderborn.de

ABSTRACT

New dynamic search data structures developed recently guarantee constant execution time per search and update, i.e., they fulfil the real-time requirements necessary for interactive walkthrough in large geometric scenes. Yet, superiority or even applicability of these new methods in practice was still an open question.

Their prototypical implementation presented in this work uses common libraries on standard workstations and thus represents a first strut to bridge this gap. Indeed our experimental results give an indication on the actual performance of these theoretical ideas on real machines and possible bottlenecks in future developments. By special algorithmic enhancements, we can even avoid the otherwise essential preprocessing step.

1. Introduction

Scientific visualization as well as computer tomography in medical diagnosis, computer aided design (CAD), and architectural construction are examples of applications which display large geometric data (*virtual scenes*) by interactive user control. If this control supports arbitrary changes of the virtual camera's position and orientation, this process is called *walkthrough*. In *dynamic* walkthrough, the user can insert and delete objects in the scene (modification).

Today three-dimensional (*massive*) objects are most commonly represented by their boundary, approximated and decomposed into surface polygons enclosing the object. These polygons (usually cut further into triangles) are stored as their vertices' and edges' coordinates together with a normal vector, color information, and texture data.

From this model representation, the computer generates a picture by performing the steps of projection, hidden-surface removal, and shading [6]. This time consuming process is called *rendering* and supported by hardware (e.g., z-buffer algorithm). The cost for rendering can be estimated by $\mathcal{O}(n + a)$ [9] and depends on two parameters: the number n of polygons and the sum a over all pixels and all polygons needed for drawing these polygons (without considering their visibility). For simplification, we will regard n as dominant.

In order to get a movie-like smooth sequence of images, as well as responsive navigation, a fixed *frame rate* of at least $20fps$ (frames per second) has to be computed. This strict real-time condition raises severe problems, since even high end graphic systems cannot guarantee such rates for very large scenes ($n \approx 1,000,000$).

*Partially supported by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), and DFG Grant Me872/7-1.

The walkthrough problem: Let $C(M)$ denote the complexity of scene M in the above cost measure (number of polygons), and let C_0 be the maximum complexity for which some specific rendering machine can still guarantee $20fps$. Then the problem is to compute a model M_r with a complexity $C(M_r) \leq C_0$ for every interval of time t. For $20fps$, time intervals are as short as $50ms$.

This problem is difficult to solve because from the visitor's position, model M_r should resemble scene M as much as possible. Additional severity arises for very large scenes which do not fit into to main memory (swapping) where access to secondary storage media can easily spoil the real-time requirements, if not performed carefully.

General approach: Most graphic systems pursue a similar approach to this problem (they differ in the kind of approximation and their rendering strategies):

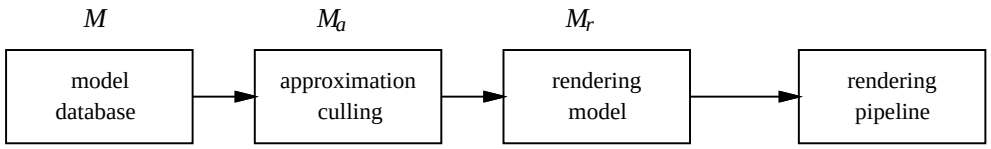


Figure 1: General approach

They compute a model M_a (See Fig. 1) including data structures for culling and polygonal model approximations of the scene M. Hence the complexity of M_a is larger than that of M. But during interactive walkthrough, the system needs only a part M_r of the scene M_a for an approximately correct view; e.g., large and very distant objects will be rendered using their approximations contained in M_a. Both approximations and culling information can be computed in a preprocessing phase (e.g., level of details models [7], visibility graph [12, 8]) or dynamically during walkthrough (e.g., repeated use of previously rendered objects via texture mapping [11] or visibility computations [3]).

The search problem: In this general approach, computation of the rendering poses the problem of deciding which objects are to be approximated (or to be rendered with full quality). For each frame, the system will have to search M_a for the sub-scene M_r presented to the visitor, choose – or even compute – approximations for all objects in M_r, and then render these approximations together with the rest of M_r.

This search problem is crucial for both visual impression and immediate responses to user interaction. In particular it should be really fast. Otherwise the costs for searching lets say some cluster of objects might consume most of the rendering time we intended to save by displaying approximations instead of the cluster itself.

In order to achieve high quality pictures, the processor should spend most of its time with rendering, not with searching.

[4] presents a very fast data structure for this goal. Other common approaches use a hierarchical structure based on ideas from Clark [2]. As an example consider the sequence of sub-scenes "world - country - town - house" organized in a tree. In existing systems, this concept is implemented with different data structure like BSP Trees (Binary Space Partitions) or Octrees.

Goals of this paper: This work is a major step towards a prototypic implementation of the ideas described in [5]. In particular, the authors' theoretical investigation on abstractly modeled dynamic walkthrough animation with several distributed visitors appears to yield a practically applicable system. In conjunction with methods from [4], this results in a network-based non-hierarchical data structure for solving the search problem in a scene.

Section 2 presents a brief description of this abstract animation system and its data structures. The current state of implementation is described in Section 3, and in Section 4 we describe an

evaluation of its performance. These experimental results are of importance to identify possible bottlenecks in future developments. In Section 5, we suggest an extension of the system considered in [5] to get rid of preprocessing by exploiting the advantage of locality.

2. A short survey of our system

Our model: Presume scene S consists of an arbitrary number of simple objects (e.g., balls) identified by their centers. The balls can be arbitrarily distributed over the scene, but they must not overlap.

Several visitors are sitting at their graphics workstations (rendering machines). They – or more precisely: their counterparts in this virtual world – can walk to arbitrary positions of the scene. For an approximately correct view, every visitor only needs the part $V_t(x) \subset S$ of the scene. The set $V_t(x) = \{b \in S : d(b,x) \leq t\}$ consists of all objects with Euclidean distance at most t from x. We call the set $V_t(x)$ the *t-environment of x*. The distance t is chosen so that the rendering complexity of the objects in $V_t(x)$ is at most C_0, i.e., the rendering machine can render the objects with a fixed frame rate (e.g., $20fps$). The scenes we have in mind are very large and have a great spatial extension so that most of the scene is stored on disks.

The scene can be modified by a modeler. Like the visitor he can walk to arbitrary positions of the scene. At any time he may insert or delete objects from the scene. These updates should occur in real-time and immediately affect the visitors' views.

Architecture: The very large scene description cannot be kept entirely in memory but must be fetched from secondary storage media. Since disk access is slow we introduce a two level access scheme for the rendering machine. Like a buffer the rendering machine will store a larger environment $V_T(y)$ for a position y and $T \gg t$ in the rendering machine's main memory so that the t-environment $V_t(x)$ can be extracted very fast.

To disburden the rendering processor, we assign this work to a third machine, the manager. This concept is introduced in [10], we adopt the approach here. The manager has access to the scene stored at disk. He computes the T-environment $V_T(y)$ for the rendering machine at fixed time intervals (steps). In every time interval $i+1$ the rendering machine will send the position x_i of its visitor to the manager who in turn computes $V_T(x_i)$ and sends this to the visitor.

The data sent should be a differential update of the form $V_T(x_i) \setminus V_T(x_{i-1})$ and $V_T(x_{i-1}) \setminus V_T(x_i)$ that allows easy computation of $V_T(x_i)$ out of $V_T(x_{i-1})$.

Let v be the maximal speed of the moving visitor and t_{int} the length of a time interval. If $T \leq 2vt_{\text{int}} + t$, then the visitor cannot in one step leave the t-environment $V_T(x)$, i.e., for an arbitrary successor position x' to x: $V_t(x') \subseteq V_T(x)$.

Data structures: For computing $V_T(x)$ or $V_t(x)$, we have to answer the following queries:

Given an arbitrary number m of balls. For position x in the scene (this position is known in data structure, too), SEARCH(x,T) reports all balls as a data structure $D(x)$ so that for each position y with $d(x,y) \leq T - t$ the t-environment $V_t(y)$ of y can be computed from $V_T(x)$ very fast. The last property is important since the rendering machine has to do this at least 20 times per second. More explicitly, we require SEARCH(x,T) to be of *output sensitive* running time in the sense that it has computational complexity $\mathcal{O}(1+k)$ where $k = |\text{SEARCH}(x,T)|$.

UPDATE(x,T,y) reports all balls in $V_T(x) \setminus V_T(y)$ and $V_T(y) \setminus V_T(x)$ such that – given $D(x)$ – the update $D(y)$ can be computed very fast.

INSERT(x) and DELETE(x) insert and delete a ball at the actual position x. Again, output sensitive running time is of high importance in order to respect the real-time requirements.

In [5], a data structure called γ-angle graph is presented that fulfils our requirements with the following additional properties: Let c be constant, m the number of balls, $l := |V_{T'}(x)|$, $l_1 :=$

$|V_{T'}(x) \backslash V_T(y)|$, and $l_2 := |V_{T'}(y) \backslash V_T(x)|$. Then

- the data structure needs space $\mathcal{O}(m)$
- SEARCH(x, T) can be done in time $\mathcal{O}(l + (\frac{T}{c})^2) = \mathcal{O}(T^2)$
- UPDATE(x, T, y) can be done in time $\mathcal{O}(l_1 + l_2 + \frac{T(r+c)}{c}) = \mathcal{O}(T \cdot (r + c))$, if $V_T(x)$ is given.
- DELETE(x) can be done in time $\mathcal{O}(c^2 \log(c))$ with high propability (w.h.p.)
- INSERT(x) can be done in time $\mathcal{O}(c^2)$ w.h.p.

For an exact description of the data structures, see [5]. A derandomized improvement with lower constants for both space and time requirements can be found in [4].

3. Implementation

Modeler/Visitor: Through the user interface the modeler/visitor can control three parts of the program: 'navigation', 'scene manipulation', 'measurements tools' (See Fig. 2).

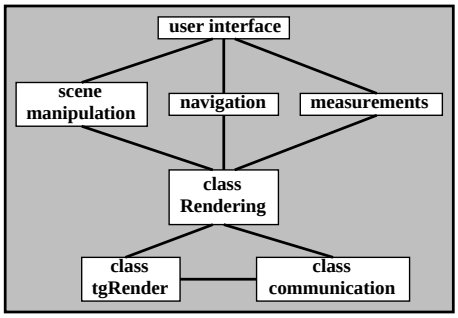


Figure 2: Implementation visitor

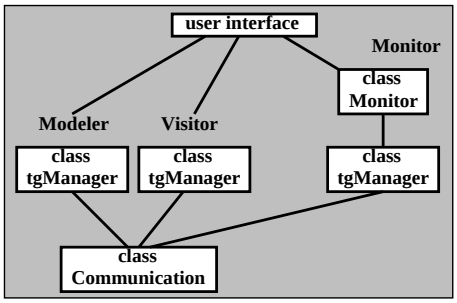


Figure 3: Implementation manager

The modeler/visitor controls navigation through the scene with the keyboard and mouse devices. She looks to the scene like a camera and can change the cameras' position and orientation arbitrarily. The modeler may insert and delete objects at every position.

In order to get reproducible experiments, we need some means to make the visitor move along the same path in the same scene several times. Therefore, we implemented a tool for recording, saving, and loading fixed camera positions and orientation. After loading these positions the computer will automatically move along them like the visitor did before.

Basis of the modeler/visitor program are the classes `Rendering`, `tgRender`, and `communication`. The data structure $V_t(x)$ for the γ-angle graph is controlled by the `tgRender` class, whereas class `communication` handles the communication with the manager and the `Rendering` class' task is to manage the polygonal scene description and to finally render the scene.

Manager: Like the modeler and visitor, the manager can be controlled via a graphical user interface. The basis for the manager is the class `tgManager`. The γ-angle graph is stored in this class as a static member. For every instance of a visitor/modeler that wants to walk through the scene, an object of the class `tgManager` is defined. In the example, we have a modeler and a visitor. Therefore, two objects are defined (cf. fig. 3).

For tests and evaluation of our implementation, a monitor is implemented which shows the γ-angle graph graph and the position and t-environment of every visitor/modeler (See Fig. 4).

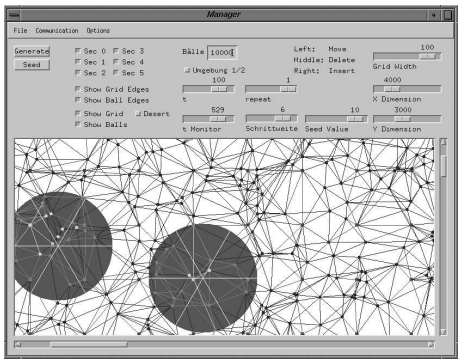


Figure 4: Screenshot Manager

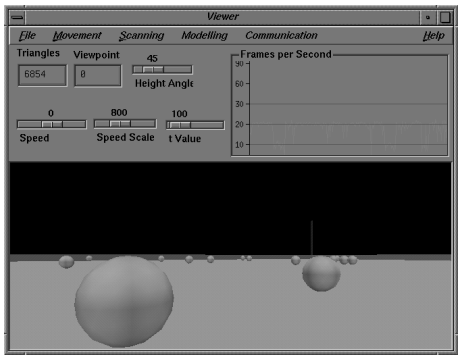


Figure 5: Screenshot Visitor/Modeler

For clipping the γ-angle graph in this window, we have a third instance of class `tgManager`. The t-environment of this object consist of the graph in the monitor window. The communication to the visitor and the manager is handled by an instance of the `communication` class.

Libraries and Communication Protocol: The implementation of the manager should run on a workstation without special graphics system (SUN Ultra Sparc, 200 MHz), therefore we used only standard libraries for Unix based systems. Our idea is to perform expensive computations on inexpensive computer systems. The special graphics system should be disburdened.

For the graphical user interface of the modeler, visitor, and manager, we use X11, XToolkit, and Motif. The implementation of the visitor and modeler runs on a special graphics workstation (SGI O2, 180Mhz). The rendering process of the scene is implemented with the standard libraries OpenGl and OpenInventor and for the graphical user interface Viewkit (See Fig. 6).

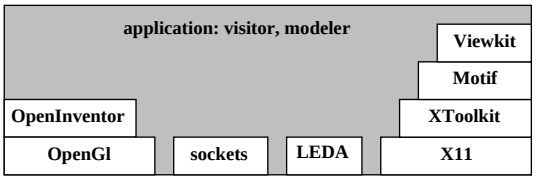


Figure 6: Libraries used

The communication between the modeler, visitor, and manager is based on the TCP/IP protocol (socket library). Therefore we can test our system on arbitrary systems, that are connected via Internet. Our aim is to test the system on different communication networks (e.g., ATM, Ethernet, etc.). A further advantage is the smaller communication overhead compared with other high level communication libraries (MPI, PVM, etc.). Some special data structures are implemented with LEDA [1].

4. Experimental Results

The objects of our scene are represented by unit size balls. A search data structure is responsible for reporting all balls within distance t. The scene will in general be very large, so it has to be stored on disk. The task of a further machine, the manager, is to access this disk. At fixed time intervals, the rendering machine receives updates from the manager to its locally stored part of the scene.

The goal of this work is to determine how fast the visitor can walk through the scene and how many balls can be rendered (number of balls). These parameters are crucial since they determine

practical applicability of our system. As it turns out, its performance is primarily limited by two factors: One bottleneck is the communication channel between manager and rendering machine. The second one arises from applying updates to the visitor's part of the scene. Therefore we explore the effect of these two constraints onto the speed of the visitor and the number of balls.

One surprising result of our research is that insertion of new objects to the scene graph reveals to be rather time consuming; much more expensive than deletion or moving. The problem occurs when, resulting from a visitor's movement, new balls of the t-environment $V_T(y)$ become part of the subscene handled by the rendering machine. Due to caching-like optimizations in the graphics library's internal data structures, the process of introducing new objects each time induces some kind of preprocessing or reinitialization. Emphasis lies on *insertion* of *new* objects: Deletion from the library data base is fast, and so is re-insertion of the same ball.

To the γ-angle graph , inserting and removing are symmetric and inexpensive operations. But the library's internal behavior is beyond our control. This has the following consequences for us: Updates of $V_T(y)$ of the rendering machine will be time consuming, but the update of $V_t(x)$ does not cause a great time demand. So we have to concentrate on the update of $V_T(y)$.

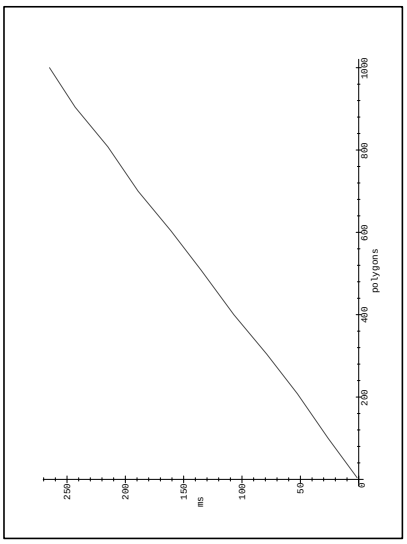


Figure 7: Running time for initialization in dependence of the ball complexity

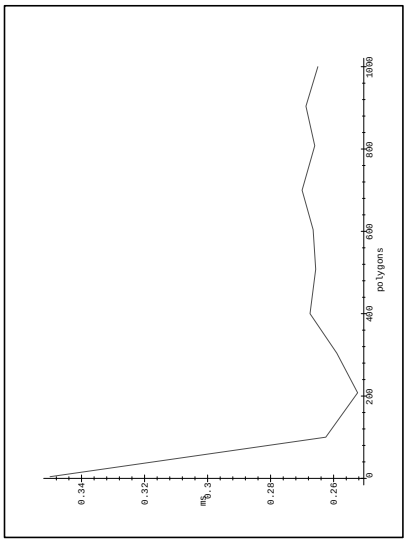


Figure 8: Running time for initialization per polygon in dependence of the ball complexity

The cost for the initialization of a ball depends of the number of triangles used for the representing its surface. Typically, in order to get a impression of a good approximation of a sphere, we need roughly 200 triangles. With the complexity of the ball we denote the number of triangles used for its representation. In Fig. 7 (figures are rotated), we have shown the initialization cost for balls of complexity from 4 to 1000 triangles. As we can see, the initialization time is nearly linear in the number of triangles. Since the curve did not cross the origin, the time cost per polygon of a ball is a little greater for very small balls. In Fig. 8, we can see this. We have drawn the initialization cost per triangle in dependence of balls with complexities from 4 to 1,000 triangles.

With this measured data we can compute how many balls we can initialize per second for balls of different complexities. In Table 1 we have shown the ratio balls/sec for balls of different complexity.

In order to get this values we need the rendering processor's full computational power (100%), i.e., it does not have any time for rendering.

triangles	4	100	208	304	400	508	604	700	808	904	1000
balls/sec	714.28	38.09	19.04	12.69	9.34	7.04	6.21	5.29	4.65	4.1	3.77

Table 1: Initialization ratio for balls of different complexity (triangles)

So at this point, we are confronted with the question how much time of the rendering processors computation power we will use for rendering computations and for initialization of the balls. This is a problem since the goal is not to disburden the rendering machine so that it can render the scene. A typically rendering capacity of our graphics workstation (SGI O2) can render up to 16,000 triangles (e.g., 80 balls each of 200 triangles), if the processor is not loaded with other work. In the following, we will describe that the practical measured values lead to satisfactory results. Furthermore we will show that there are two tradeoffs which are convenient for our model.

In Table 2, we have shown the maximum speed of the visitor for a scene consisting of balls having a complexity of 100 triangles.

percentage for initialization	size of t	maximum speed
10%	$100m$	1.32 $\frac{m}{s}$
10%	$500m$	6.61 $\frac{m}{s}$
20%	$100m$	2.97 $\frac{m}{s}$
20%	$500m$	14.86 $\frac{m}{s}$

Table 2: An example for scenes consisting of balls with a complexity of 100 triangles

If we allow 10% of the rendering machine's computation power for initialization, we get a maximum speed of 1.32 $\frac{m}{s}$ for a scene of $100m$ radius, and a maximum speed of 6.61 $\frac{m}{s}$ for a scene of $500m$ radius. In the other case, if we allow more time for initialization computation, e.g., 20% we get a maximum speed of 2.97 $\frac{m}{s}$ for a scene of $100m$ radius, and a maximum speed of 14.86 $\frac{m}{s}$ for a scene of $500m$ radius.

Here we have two tradeoffs: One between speed and size of the scene and the other between speed and the percentage for initialization computations. If we enlarge our scene, we can get a higher speed of the visitor. Future versions of our system will take advantage of this tradeoff, since in a small scene with a high density of balls the visitor walks slowly to see every part of the scene. Otherwise, in larger scenes with a lower density of balls, the visitor walks faster in order to reach the next ball.

We can get a higher speed if we enlarge the percentage for initialization computations. In this case, the remaining scene will have a less density, and so the visitor tries to reach the next object with high speed. We will exploit this in our implementation so that the rendering machine tries to initialize more balls if the visitor walks slowly. The absolute values of the maximum speed seem usable for a practical application.

For the communication bottleneck, we get similar satisfying results. We will describe them in our final report.

5. An extension of our architecture

At this point, our architecture will be extended in comparison to [5]. Our second problem induced by the management of large scenes is the expensive space requirement for the search of data structures of the objects. Because of locality, the visitor sees similar t-environments $V_t(x)$ of objects in consecutive steps of a few time intervals, i.e., she needs not a search data structure that manages all objects of the scene. We want to exploit this locality for saving memory of the manager and for saving time of the QUERY-operations of the rendering machine. Therefore, we use a two level search data structure of the manager. We distinguish a *coarse grained level* and a *fine grained level* (See Fig. 9).

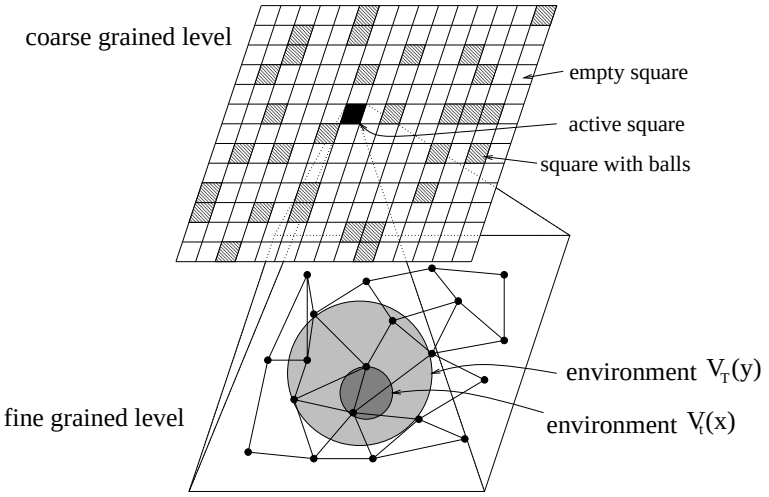


Figure 9: Two level access for the manager

At the coarse grained level, we divide the scene into *squares*. We denote squares with at least one ball as *full* and those without balls as *empty*. All balls of the same full square at the coarse level are stored on disk in a list. This list is represented by the center position of the full square and the unsorted list of elements. At each time, there is exactly one square that is *active* (See Fig. 9). This square contains the visitor's position. For this square, all balls from the disk are loaded into the main memory of the manager. These balls are contained in the so called *fine grained level*.

For each of the two levels, we have an (arbitrary) search data structure D_{COARSE} and D_{FINE}. Such a search data structure should have the operations QUERY and BUILD. BUILD builds the data structure for a given input from scratch and QUERY is query-operation (e.g next neighbor or range query) on the data structure. The input for D_{COARSE} consists of the positions of the full squares (we do not store the mesh explicitly) and the input for D_{FINE} of all balls of the active square. In our system, QUERY is one of the operations SEARCH, UPDATE, DELETE, and INSERT. The manager computes for balls of the active square the t-environment $V_T(y)$ as described above with the D_{FINE} data structure.

A walk of the visitor through the scene results in the following operations for this two level hierarchy: At every time we have a data structure D_{FINE} for the balls of the active square. The manager computes from this data structure the set $V_T(y)$ for the rendering machine. When the visitor leaves the active square, the manager has to search the next neighboring full square in the D_{COARSE} data structure (with QUERY operation of D_{COARSE}). If the directly neighboring square is an empty square there is nothing to do. Otherwise, the manager loads all balls of the new active square into the main memory and computes with the BUILD operation of D_{FINE} the data structure for the fine grained level from scratch. The data structure D_{FINE} and all balls of the previously active

square are removed from memory. Now the manager can compute the t-environments $V_T(y)$ with the QUERY operations of D_{FINE} and so on. The dimension of the mesh should be chosen in a way that it takes a lot of time for the visitor to cross a square. So the time for computing D_{FINE} from scratch and loading the balls will be expensive. Therefore it is recommended to hold permanently the data structures D_{FINE} for the at most 8 neighboring full squares of the active square. More than the 8 neighbors are not necessary since we assume the squares very large so it will go by a lot of time for moving across the active square.

We have a tradeoff between the dimension of the mesh and the number of balls of a square. The question is how is the optimum size of the mesh? Let m be the number of balls, s the distance of two consecutive moves of the visitor, d the dimension of the mesh ($d \times d$ mesh), and $c \cdot s$ be the size of the scene. Let $T^{\text{FINE}}_{\text{QUERY}}$ be the time for a query of D_{FINE} and let $T^{\text{COARSE}}_{\text{QUERY}}$ be the time for a query of D_{COARSE} (for BUILDanalogue). For a walk of the visitor of length $k \cdot s$, we get a total runtime T_{move} of

$$T_{move}(k, m, d, c) = kT^{\text{FINE}}_{\text{QUERY}}\left(\frac{m}{d^2}\right) + \frac{k}{c/d}\left(T^{\text{COARSE}}_{\text{BUILD}}\left(\frac{m}{d^2}\right) + T^{\text{COARSE}}_{\text{QUERY}}\left(d^2\right)\right).$$

For the following example, we assume the runtime for D_{COARSE} and D_{FINE} to be equal to $T_{\text{QUERY}} = q \cdot m\log(m)$ and $T_{\text{BUILD}} = p \cdot \log(m)$, and the balls are randomly distributed over the scene. Then we get for T_{move}

$$T_{move}(k,\ m,\ d,\ c,\ p,\ q) = k\ \left(q\log_2(\frac{m}{d^2}) + \frac{d}{c}\left(p\,\frac{m}{d^2}\log_2(\frac{m}{d^2}) + q\log_2(d^2)\right)\right).$$

The minimum of T_{move} depending on d is the solution the following equation for d

$$0 = -\frac{k\,(2\,q\,c\,d + m\,p\ln(\frac{m}{d^2}) - q\ln(d^2)\,d^2 + 2\,m\,p - 2\,q\,d^2)}{d^2\,c\ln(2)}.$$

If we solve this equation for, e.g., $m = 10^6$ balls, $k = 1,000$ steps of the visitor, $c = 1,000$ (area of the scene), and $p = q = 1$ (constants for runtime), then we will get a minimum of T_{move} for $d = 543$. Important is the question where T_{move} attains its minimum. If it does at the extreme points need not this two level structure. As we can see in some examples, the minimum depends strongly on constants p and q.

If a visitor walks more than twice through the entire scene, then our permanent process of computing and removing the fine grained level data structure will be more time consuming than computing the data structure for all objects of the scene once. But we have in mind that our scene has a large spatial extension, so we can save memory for the manager since he has a coarse data structure for the entire scene. A further advantage is that the data structure of the rendering machine for computing the t-environment $V_t(x)$ will have an $\mathcal{O}(m/d^2)$ input size instead of $\mathcal{O}(m)$. For our strict real time requirements, this constant factor is important for data structures with $\mathcal{O}(\log(n))$ QUERY-time (e.g., trees) since computing of the rendering machine is more critical as for the manager machine as we discussed above.

6. Ongoing Work

At the next step we will test search data structures in practice and compare them with other standard data structures. In a further step, we will implement the two level access for the manager and give an evaluation.

Acknowledgements

We would like to thank Friedhelm Meyer auf der Heide, Willy-Bernhard Strothmann, and Rolf Wanka for helpfull comments and suggestions.

References

[1] Christoph Burnikel, Jochen Knemann, Kurt Mehlhorn, Stefan Nher, and Stefan Schirra. Geometric computation in LEDA. In *Proceedings of the 11th Annual Symposium on Computational Geometry (SCG 95)*, pages C18–C19, 1995.

[2] James H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19(10):547 – 554, October 1976.

[3] S. Coorg and S. Teller. Real-Time Occlusion Culling for Models with Large Occluders. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics 1997*, pages 83 – 90, April 1997.

[4] Matthias Fischer, Tamás Lukovszki, and Martin Ziegler. Geometric Searching in Walkthrough Animations with Weak Spanners in Real Time. In *Proceedings of the Sixth Annual European Symposium on Algorithms*, 1998.

[5] Matthias Fischer, Friedhelm Meyer auf der Heide, and Willy-Bernhard Strothmann. Dynamic Data Structures for Realtime Management of Large Geometric Scenes. In *Proceedings of the Fifth Annual European Symposium on Algorithms*, pages 157–170, 1997.

[6] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison Wesley, 1995.

[7] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualisation of Complex Virtual Environments. In James T. Kajiya, editor, *Proceedings of the SIGGRAPH '93*, volume 27, pages 247 – 254, 1993.

[8] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of Large Amounts of Data in Interactive Building Walkthroughs. In *Proceedings of the SIGGRAPH '91*, pages 11 – 20, 1992.

[9] P. S. Heckbert and M. Garland. Multiresolution Rendering Modeling for Fast Rendering. *Proceedings of the Graphics Interface '94*, pages 43–50, May 1994.

[10] Jonathan Mark Sewell. *Managing Complex Models for Computer Graphics.* PhD thesis, University of Cambridge, Queens' College, March 1996.

[11] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *Proceedings of the SIGGRAPH '96*, pages 75 – 82, August 1996.

[12] Seth J. Teller and Carlo H. Sequin. Visibility Preprocessing For Interactive Walkthroughs. In *Proceedings of the SIGGRAPH '90*, pages 61 – 69, 1991.

Proceedings WAE'98, Saarbrücken, Germany, August 20-22, 1998
Ed. Kurt Mehlhorn, pp. 143–154

An Implementation of the Binary Blocking Flow Algorithm

Torben Hagerup
Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt
Robert-Mayer-Straße 11-15
D–60054 Frankfurt am Main, Germany
e-mail: hagerup@informatik.uni-frankfurt.de

Peter Sanders
Max-Planck-Institut für Informatik, Im Stadtwald
D–66123 Saarbrücken, Germany
e-mail: sanders@mpi-sb.mpg.de

and

Jesper Larsson Träff
Technische Universität München, Lehrstuhl für Effiziente Algorithmen
D–80290 München, Germany
e-mail: traeff@informatik.tu-muenchen.de

ABSTRACT

Goldberg and Rao recently devised a new *binary blocking flow algorithm* for computing maximum flows in networks with integer capacities. We describe an implementation of variants of the binary blocking flow algorithm geared towards practical efficiency and report on preliminary experimental results obtained with our implementation.

1. Introduction

Despite intensive research for more than three decades, problems related to flows in networks still motivate cutting-edge algorithmic research. In a recent development, Goldberg and Rao [4] combined elements of the algorithms of Dinitz [2] and Even and Tarjan [3] with new arguments to obtain a simple algorithm, the *binary blocking flow (BBF) algorithm*, for computing maximum flows in networks with integer capacities. On networks with n vertices, m edges, and integer capacities bounded by U, the algorithm runs in $O(m\Lambda \log(n^2/m) \log U)$ time, where, here and in the following, $\Lambda = \min\{m^{1/2}, n^{2/3}\}$.

The running time mentioned above places the BBF algorithm among the theoretically fastest maximum-flow algorithms, and our work was motivated by curiosity as to whether the new algorithm is fast in practice as well as in theory. In order to investigate this question, we implemented the algorithm in C++ using the LEDA library [8, 9], introducing a number of modifications intended to make the algorithm run faster in practice, while preserving the theoretical bound on the running time. This paper describes our implementation and reports on preliminary experimental findings. Our results are as yet inconclusive, but seem to indicate that variants of the BBF algorithm can indeed solve problem instances drawn from some frequently considered families with a much smaller number of blocking-flow computations than its ancestor, Dinitz' algorithm. Together with an efficient blocking-flow routine it sometimes even outperforms highly developed preflow-push algorithms. However, much remains to be done to make the algorithm more robust.

2. The binary blocking flow algorithm

In this section we describe and analyze the BBF algorithm. We start with a generic formulation that encompasses the original formulation of Goldberg and Rao as a special case. The generic algorithm is analyzed in Section 2.2, and Section 2.3 describes a number of possible instantiations of the generic algorithm. We assume familiarity with some of the basic definitions pertaining to flows in networks.

2.1. A generic formulation

The task is to compute a maximum s-t-flow in a network $\mathcal{N} = (G = (V, E), c, s, t)$, where $c : E \to \mathbb{N}$ maps each edge to its capacity. We take $n = |V|$, $m = |E|$, and $U = \max_{e \in E} c(e)$ and assume without loss of generality that E is *symmetric*: Whenever E contains an edge (u, v), it also contains the *reverse* edge (v, u). The *residual capacity* of each edge $e \in E$ with respect to a flow f in $\mathcal{N}$ is defined, as usual, as $r_f(e) = c(e) - f(e)$ (f is assumed antisymmetric; i.e., $f(u, v) = -f(v, u)$ for all $(u, v) \in E$), and the *residual network* of $\mathcal{N}$ with respect to f is $\mathcal{R}_f = ((V, E_f), r_f, s, t)$, where $E_f = \{e \in E \mid r_f(e) > 0\}$. The value of a flow f will be denoted by $|f|$, and the value of a maximum flow in $\mathcal{R}_f$ will be called the *residual value* of f. The capacity of a path in a network is the minimum capacity of an edge on the path.

Figure 1 shows the top-level structure of the generic BBF algorithm. Similar to Dinitz' algorithm, it operates in *rounds*, each of which computes a blocking flow (g') in an auxiliary acyclic network ($\mathcal{C}'$), called the *acyclic core*, and updates a flow (f) in $\mathcal{N}$ accordingly. The differences lie in the construction of $\mathcal{C}'$, carried out by the routine 'acyclicCore', and in the meaning of "accordingly", which is embodied in the routine 'transferFlow'. The auxiliary network used in Dinitz' algorithm consists of all s-t-paths with a minimal number of edges in the current residual network, and the blocking flow is simply added to the current flow edge by edge. The construction of the acyclic core, on the other hand, depends on a real parameter Δ, which is essentially chosen proportional to an upper bound $\widehat{F}$ on the current residual value, computed by the routine 'estimateFlow'. The constant of proportionality is α/Λ, where α is a positive constant, and the value of Δ is kept fixed until the estimate of the residual value has dropped to β times its original value, where β is another positive constant. The rounds executed with a common value of Δ constitute a *phase*.

```
Function maxFlow(N = (G = (V, E), c, s, t)): Flow
    f := 0                                              (* current flow *)
    while estimateFlow(R) ≥ 1 do                        (* a phase *)
        F̂ := estimateFlow(R)                            (* flow bound *)
        Δ := ⌈αF̂/Λ⌉
        while estimateFlow(R) ≥ βF̂ do                   (* a round *)
            C' := acyclicCore(R, Δ)
            g' := blockingFlow(C')
            f := f + transferFlow(g', C', N)
    return f
```

Figure 1: Top-level structure of the BBF algorithm.

The acyclic core $\mathcal{C}'$ is derived from a (full) *core* of $\mathcal{R}_f$, a concept that we define next. For conciseness, we suppress a dependence on the current value of Δ. First, let $L : \mathbb{N} \to \{0, 1\}$ be the function with $L(x) = 1$ for $x < 3\Delta$ and $L(x) = 0$ for $x \geq 3\Delta$. We define the *length* of each residual edge $e \in E_f$ as $\ell_f(e) = L(r_f(e))$ and the *height* of each $v \in V$, $d_f(v)$, as the length of a shortest path from v to t in the residual graph $\mathcal{R}_f$ with this length function (∞ if there is no such path). We say that an edge $(u, v) \in E$ is *downward* if $d_f(v) < d_f(u)$, *upward* if $d_f(v) > d_f(u)$, and *horizontal* if $d_f(v) = d_f(u)$. A (full) *core* of $\mathcal{R}_f$ (with respect to Δ) is a subnetwork of $\mathcal{R}_f$ that contains all

vertices in V, all downward edges, no upward edges, and a subset of the horizontal edges in $\mathcal{R}_f$ that includes at least every edge of length 0 and every edge e with $r_f(e) > 2\Delta$ whose reverse edge $\overleftarrow{e}$ belongs to the core; moreover, no edge on a cycle in the core is of capacity less than 2Δ. Goldberg and Rao do not introduce the notion of a core, but their algorithm can be viewed in our setting as using a minimal core, one whose set of horizontal edges includes only the residual edges of length 0 and the reverses with residual capacity at least 2Δ of such edges.

The generic BBF algorithm leaves some flexibility in exactly which core to use. Even more flexibility is present in the derivation of the acyclic core from the full core. Goldberg and Rao construct the acyclic core $\mathcal{C}'$ by contracting each strongly connected component (SCC) of the full core to a single vertex. The algorithmically most interesting generalization in our approach is that we replace each strongly connected component K of the full core by an acyclic network $\mathcal{C}'(K)$, called the *acyclic component* (AC) corresponding to K. For each vertex v in K we designate (not necessarily distinct) nodes innode(v) and outnode(v) in $\mathcal{C}'(K)$, and the acyclic core $\mathcal{C}'$ consists of the union of all acyclic components together with all *inter-component* edges, all edges of the form (outnode(u), innode(v)), where (u, v) is an edge of the full core and u and v belong to different SCCs. Every inter-component edge (outnode(u), innode(v)) inherits its capacity from the edge (u, v), and we generally identify it with (u, v).

The generic algorithm requires the routines 'acyclicCore' and 'transferFlow' to satisfy the following properties:

Capacity property: For each SCC K of the core and all vertices u and v in K, there is a path in $\mathcal{C}'(K)$ from innode(u) to outnode(v) of capacity at least Δ.

Transfer property: For every blocking flow g' in $\mathcal{C}'$, the call 'transferFlow($g', \mathcal{C}', \mathcal{N}$)' returns a flow g in the corresponding full core $\mathcal{C}$ with $|g| \geq \min\{|g'|, \Delta\}$ and $g(e) \geq \min\{g'(e), \Delta\}$ for every inter-component edge e.

Anticycle property: For every blocking flow g' in $\mathcal{C}'$, if the call 'transferFlow($g', \mathcal{C}', \mathcal{N}$)' returns a flow g with $|g| < \Delta$, then $g(e) < 2\Delta$ for every edge e in $\mathcal{C}$, and $g(e) < \Delta$ for every inter-component edge e in $\mathcal{C}$.

2.2. Analysis

In the following series of lemmas, we always consider a round that starts with the flow f and ends with the flow $f + g$; i.e., g is the flow returned by the call of 'transferFlow' in the round under consideration. $\mathcal{C}$ and $\mathcal{C}'$ denote the (full) core and the acyclic core used by the round. Changes of Δ are considered to take place between rounds, not within rounds.

In an execution of Dinitz' algorithm, no node height ever decreases. As shown in the first part of the following lemma, this is also true of the BBF algorithm, as long as Δ is not changed.

Lemma 2.1. *For all $u \in V$, $d_{f+g}(u) \geq d_f(u)$. Furthermore, suppose that $d_{f+g}(u) = d_f(u) < \infty$ and let π be a shortest path from u to t in $\mathcal{R}_{f+g}$. Then for all nodes v on π, $d_{f+g}(v) = d_f(v)$.*

Proof. Since a core contains no upward edges, adding g to f cannot create a new edge that is downward with respect to f or shorten such an edge. If follows that $d_{f+g}(u) \geq d_f(u)$ for all $u \in V$.

Suppose now that $d_{f+g}(u) = d_f(u) < \infty$ and that π is a shortest path from u to t in $\mathcal{R}_{f+g}$. Assuming that π contains at least one node w with $d_{f+g}(w) < d_f(w)$, let w be the first such node on π. By assumption, $w \neq u$. Let v be the node on π immediately before w. Then $\ell_{f+g}(v, w) < \ell_f(v, w)$, which implies that (v, w) is not downward with respect to f and that (v, w) is upward with respect to $f + g$, contradicting the fact that no shortest path to t can contain an upward edge. ∎

Lemma 2.2. *If $|g| < \Delta$ then g is a blocking flow in $\mathcal{C}$.*

Proof. Assume that some s-t path π in $\mathcal{C}$ is not blocked by g. Replacing each edge on π within an SCC of $\mathcal{C}$ by a path of capacity at least Δ according to the capacity property, we obtain a walk π'

in $\mathcal{C}'$. At least one edge e on π' is saturated by the blocking flow g' in $\mathcal{C}'$ from which g is derived. If e is an inter-component edge, we must have $|g'| \geq g'(e) > g(e)$, but the transfer and anticycle properties show that this cannot be the case if $|g| < \Delta$. Therefore the capacity of e is at least Δ, and it follows that $|g'| \geq \Delta$ and that $|g| \geq \Delta$. ∎

The following theorem plays the same central role in our analysis as does Theorem 4.3 of [4] in the analysis of Goldberg and Rao. Our proof is more detailed and works for the generic BBF algorithm.

Theorem 2.3. *If $d_{f+g}(s) = d_f(s) < \infty$, then $|g| \geq \Delta$.*

Proof. Assume that $d_{f+g}(s) = d_f(s) < \infty$ and let π be a shortest path from s to t in $\mathcal{R}_{f+g}$. By Lemma 2.1, $d_f(u) = d_{f+g}(u)$ for all nodes u on π. Define a *noncore edge* as an edge in $\mathcal{R}_f$, but not in $\mathcal{C}$. Suppose that e is a noncore edge on π. Since e lies on a shortest path in $\mathcal{R}_{f+g}$, it cannot be upward. As a noncore edge, it is also not downward. Therefore, e is horizontal. Since e lies on a shortest path in $\mathcal{R}_{f+g}$ and is horizontal, we must have $\ell_{f+g}(e) = 0$. On the other hand, since e is noncore, $\ell_f(e) > 0$. This is possible only if the reverse $\overleftarrow{e}$ of e belongs to the core $\mathcal{C}$ and an increase in the flow over $\overleftarrow{e}$ in the round under consideration increases the residual capacity of e to at least 3Δ. Since e is noncore, $r_f(e) < 2\Delta$. Hence the flow over $\overleftarrow{e}$ increases by at least Δ.

If the reverse of some noncore edge on π is an inter-component edge, we can conclude from the anticycle property that $|g| \geq \Delta$. Assume the opposite. Then the endpoints of every noncore edge $e = (u, v)$ on π belong to the same SCC of $\mathcal{C}$, i.e., the core contains a path from u to v, called the *detour* of e, of capacity at least 2Δ. Consider the walk π', identical to π, except that every noncore edge is replaced by its detour. π' goes from s to t and belongs entirely to $\mathcal{C}$. Assume that $|g| < \Delta$. Then, by Lemma 2.2, some edge on π' is saturated by g. This edge cannot belong to π (π is a path in $\mathcal{R}_{f+g}$), and hence must be an edge e' on a detour. We have $g(e') = r_f(e') \geq 2\Delta$, and the anticycle property implies that $|g| \geq \Delta$. ∎

As observed by Goldberg and Rao, the fact that every edge in the residual graph joining two adjacent layers is of residual capacity less than 3Δ means that one can use the arguments of Even and Tarjan [3] to show that after $c\lceil\Lambda\rceil$ rounds, the residual value of f has dropped below $\beta\widehat{F}$. Here c is a positive constant that depends on α and β. In a very simple implementation of the BBF algorithm, one could therefore end a phase after $c\lceil\Lambda\rceil$ rounds and choose $\Delta := \lfloor\beta\Delta\rfloor$ for the next round. Since the residual value is certainly bounded by nU initially, a maximum flow can be computed in $O(\Lambda \log(nU))$ rounds, a bound that was reduced to $O(\Lambda \log U)$ rounds by Goldberg and Rao. Assuming that the acyclic core $\mathcal{C}'$ has $O(n)$ vertices and $O(m)$ edges and that the time bound of $O(m \log(n^2/m))$ of [5] for computing a blocking flow in $\mathcal{C}'$ dominates the remaining computation of a round—two very mild assumptions—the simple BBF algorithm achieves the same running time of $O(m\Lambda \log(n^2/m) \log U)$ as the algorithm described by Goldberg and Rao.

Practically more efficient instantiations of the generic BBF algorithm need to use more intricate routines 'estimateFlow' that return the minimum of several upper bounds on the residual flow value. The capacity of every s-t cut in the current residual network is such an upper bound. It can be seen that the time bound established above holds for any variant that takes into account at least the cut $(\{s\}, V\backslash\{s\})$ and the *canonical cuts* $(S_k, V\backslash S_k)$, for $k = 1, \ldots, d_f(s)$, where $S_k = \{v \in V \mid d_f(v) \geq k\}$. Note that the canonical cut values can be found (almost) as a byproduct of the computation of the distance function d_f. Our implementation additionally uses the cut $(V\backslash\{t\}, \{t\})$ and the upper bound $\widehat{F} - |g|$ on the residual flow value, where $\widehat{F}$ is the upper bound of the previous round and g is the flow added to f in that round.

We mention briefly that the worst-case analysis of Goldberg and Rao can be improved in terms of constant factors. Let us say that a variant of the binary blocking flow algorithm achieves a *leading factor* of C if the number of rounds necessary to compute a maximum flow is bounded by $C\Lambda \log U + O(\Lambda + \log U)$. Inspection of [4] shows that the authors prove their algorithm to achieve a leading factor of 7 for $m \leq n^{4/3}$ (the *sparse case*) and of 6 for $m > n^{4/3}$ (the *dense*

case). This is done by ending a phase when half of the remaining flow is known to have been routed and by choosing $\Delta = \max\{\lceil \widehat{F}/m^{1/2} \rceil, \lceil \widehat{F}/n^{2/3} \rceil\}$, i.e., $\alpha = 1$ and $\beta = 1/2$. ([4] specifies $\Delta = \min\{\lceil \widehat{F}/m^{1/2} \rceil, \lceil \widehat{F}/n^{2/3} \rceil\}$, which we assume to be a simple error). By choosing smaller values for α and β we achieve leading factors of approximately 2.98 in the sparse case and 2.51 in the dense case. It appears that both leading factors of Goldberg and Rao can be lowered to 5 by using the wheels-within-wheels construction of Knuth [7], but this will benefit our analysis as well.

2.3. Some instantiations

The original algorithm of Goldberg and Rao can be viewed as an instantiation of our generic BBF algorithm as follows: Each SCC of $\mathcal{C}$ is replaced by a single vertex. If the value of the blocking flow g' exceeds Δ, the function 'transferFlow' first returns $|g'| - \Delta$ units of flow to s. Transferring the remaining flow to $\mathcal{C}$ is a matter of routing "within" the SCCs. Since, by construction, each edge within an SCC is of capacity at least 2Δ and a total of at most Δ units of flow is to be routed, this can be done in the following two-step manner: The flow entering the SCC is routed to an arbitrarily chosen root via a spanning *intree*, i.e., a spanning tree of edges leading to the root. Then the flow is distributed as appropriate via a spanning *outtree* (which is not necessarily disjoint from the intree—hence the factor of 2).

The simplest variant of our approach incorporates the tree-routing idea described above into the construction of the acyclic core. For trivial components K consisting of a single node, $\mathcal{C}'(K)$ simply is a copy of K. Otherwise, each node $v \in K$ of an SCC is replaced by two copies v_{in} and v_{out} such that $\text{innode}(v) = v_{\text{in}}$ and $\text{outnode}(v) = v_{\text{out}}$. An arbitrary node $w \in K$ is chosen as a root and the nodes of the form $\text{innode}(u)$ are connected to w_{in} via an intree such that an edge $(u_{\text{in}}, v_{\text{in}})$ in this tree corresponds to an edge (u, v) in K. Analogously, the nodes of the form $\text{outnode}(u)$ are reached from w_{out} via an outtree. The intree and the outtree can be constructed in various ways. We use breadth-first traversal of K. The root nodes w_{in} and w_{out} are connected by an infinite-capacity edge. Edges in K corresponding to both an intree edge and an outtree edge have their residual capacity split evenly among these two edges. The remaining intree edges and outtree edges get the entire residual capacity from the corresponding edge in K. The function 'transferFlow' simply maps the flow on edges in $\mathcal{C}'$ to the corresponding edges in $\mathcal{C}$.

It is easy to see that the capacity, transfer, and anticycle properties hold for this way of constructing acyclic components: $\mathcal{C}'$ has linear size and 'makeAcyclic' and 'transferFlow' can be implemented to run in linear time. For any two nodes u and v in an SCC K there is a path of capacity at least Δ from $\text{innode}(u)$ over the intree and the outtree to $\text{outnode}(v)$. The transferred flow is legal since it cannot exceed the residual capacity of any edge in K and because any flow on a path in $\mathcal{C}'(K)$ is mapped to a corresponding path in K. The capacity of an edge from $\mathcal{C}$ is split between at most two edges in $\mathcal{C}'$, and even that only for edges in an SCC, i.e., on a cycle in $\mathcal{C}$.

The main advantage of our approach is that a round may increase the s-t-flow by considerably more than Δ. This does not yield a better worst-case bound, but our experiments show that it is crucial to achieving a good performance in a practical setting. We note a number of additional measures that may increase the flow found in blocking-flow computations.

a) We can introduce infinite-capacity edges between any pair $(v_{\text{in}}, v_{\text{out}})$.

b) An edge $(u, v) \in K$ that has no correspondence in the intree or the outtree can be used to introduce a corresponding edge $(u_{\text{in}}, v_{\text{out}})$ in $\mathcal{C}'(K)$, since no cycles are introduced.

c) Some edges $(u, v) \in K$ as in b) can also be translated into $(u_{\text{in}}, v_{\text{in}})$ or $(u_{\text{out}}, v_{\text{out}})$ edges without incurring cycles. For example, we have implemented a variant which assigns BFS-numbers $k(v)$ to nodes of $\mathcal{C}'(K)$ and allows edges with $k(u_{\text{in}}) < k(v_{\text{in}})$ in the intree as well as edges with $k(u_{\text{out}}) < k(v_{\text{out}})$ in the outtree.

d) The definition of a core is quite flexible and, for example, allows those horizontal length-1 edges that do not introduce cycles into $\mathcal{C}$. We have implemented a variant which allows edges that do not violate a topological order of the SCCs.

3. Experimental results

In this section we present a preliminary experimental evaluation of the binary blocking flow (BBF) algorithm. We will mainly compare it to an implementation of Dinitz' algorithm that uses the same algorithm for computing blocking flows. In most cases we have used a simple $O(mn)$-time DFS-based routine as in Dinitz' original presentation [2][1]. We have also experimented with an $O(n^2)$-time implementation of Tarjan's wave algorithm [10], which sometimes (though rarely) yields better results. In all cases, all nodes in the acyclic core from which t is not reachable are first eliminated. This is crucial to the performance of certain blocking-flow algorithms.

In addition, we make comparisons with two implementations of preflow-push algorithms, namely the LEDA `MAX_FLOW()` function and the highest-label preflow-push code `h_prf` by Cherkassky and Goldberg, which performed best in the study [1]. When not otherwise stated, we have used a "standard" variant of the BBF algorithm with $\alpha = 2$, $\beta = 1/2$, and the refinements a) and b) from Section 2.3. This choice may be somewhat arbitrary, yet we believe it to reflects a reasonable compromise between simplicity, sophistication, and overhead for refinements not needed by the worst-case analysis.

We have experimented with the same problem families as in [1]. For the families whose instances are generated randomly, the figures reported are averages over five problem instances. A more detailed description of the problem families can be found in [6, Appendix A]. To keep the computation times within limits, we most often used smaller instances than in [1], ensuring a certain overlap to facilitate direct comparison. Experiments were carried out on a 200 MHz Sun Ultra-2 using GNU C++ 2.8.1 with compiler setting `-O6` and LEDA 3.7.

Table 1 lists the running times for the four codes. These times give a rough idea of the strengths and weaknesses of the individual algorithms. We stress that a direct comparison between the performance of the four implementations is not very meaningful; one should take into account that `h_prf` is carefully coded in C and uses many well-tuned heuristics, whereas the other codes put more emphasis on ease of implementation and accept some library overhead by using LEDA. Since our prime motivation with this study is to gain first insights into the behavior of the BBF algorithm, compared to Dinitz' algorithm and preflow-push algorithms, we feel that the LEDA implementations suffice at this point; for the same reason we do not include running times for available C code for Dinitz' algorithm, which can be found, for example, in [1].

Table 2 is intended to give a more implementation-independent picture by providing counts of important operations that can be compared more directly. For the Dinitz and BBF algorithms a key parameter is the number of edges scanned while searching for augmenting paths in the blocking-flow computation, an operation commonly known as "Advance". This number roughly accounts for the time spent in computation of blocking flows and plays a similar role as the number of push operations in preflow-push algorithms. We also count the number of rounds performed, which is indicative of where the time is spent, since the Dinitz and BBF algorithms in each round construct acyclic networks of size $O(m)$. For the BBF algorithm, we also count the number of phases, as well as the total number of SCCs encountered. The effect of allowing more than Δ units of flow per round is estimated by assuming that a blocking flow of value $B > \Delta$ saves $\lfloor B/\Delta \rfloor$ rounds. Since blocking flows are not unique and, in particular, since the blocking flow for the next round depends on how the $B - \Delta$ units of excess flow are routed back to s in the original BBF algorithm of [4], this estimate is not always accurate. Nevertheless, we believe it to give a good idea of how many rounds are saved over the original algorithm and present these numbers in the column "Saved". In all experiments conducted, a significant number of rounds were saved in this sense. We take this as a vindication of our alternative to contraction of SCCs as originally proposed in [4].

We observe that for the `rmf` instances, the BBF algorithm needs only a single round. Although it performs about as many advance operations as all blocking-flow computations of Dinitz' algorithm put together, the BBF algorithm is much faster (10–40 times). This is because Dinitz' algorithm

[1] We would like to thank Thomas Erlebach for providing us with a LEDA implementation.

n	m	LEDA	`h_prf`	Dinitz	BBF
Genrmf-Long (`rmf-Long`)					
675	2810	0.01	0.01	1.16	0.48
1152	4956	0.03	0.01	2.47	0.43
2205	9716	0.06	0.03	8.24	0.77
4096	18368	0.14	0.05	22.79	1.90
9000	41300	0.38	0.14	92.34	6.02
15488	71687	0.66	0.26	200.31	15.00
30589	143364	1.62	0.58	695.96	40.30
Genrmf-Wide (`rmf-Wide`)					
507	2210	0.03	0.01	0.96	0.11
1024	4608	0.07	0.03	2.85	0.24
2205	10164	0.21	0.09	9.45	0.62
3920	18256	0.56	0.20	25.78	1.28
8664	40964	2.11	0.57	91.69	3.48
16807	80262	5.11	1.70	285.35	8.18
32768	157696	14.09	4.25	783.04	19.32
Acyclic-Dense					
64	2016	0.01	0.00	0.06	0.15
128	8128	0.06	0.01	0.23	0.47
256	32640	0.33	0.04	1.08	2.92
512	130816	2.00	0.45	5.02	16.43
`AK`					
518	775	0.01	0.01	2.36	0.21
1030	1543	0.03	0.04	9.37	0.46
2054	3079	0.13	0.13	37.14	1.12
4102	6151	0.50	0.47	151.02	3.13
8198	12295	2.04	1.77	709.29	9.91
16390	24583	8.39	6.74	3090.65	38.77
32774	49159	34.08	27.49	12942.90	151.59
Washington-RLG-Long (`RLG-Long`)					
4098	12224	0.18	0.04	6.16	63.13
8194	24512	0.39	0.10	22.53	229.95
16386	49088	0.81	0.21	81.76	719.52
Washington-RLG-Wide (`RLG-Wide`)					
8194	24448	0.49	0.14	19.15	171.29
16386	48896	1.08	0.29	44.48	482.81
Washington-Line-Moderate (`wlm`)					
514	3007.4	0.01	0.00	0.35	4.51
1026	8069.6	0.05	0.01	1.16	34.08
2050	22293.6	0.14	0.01	3.11	165.10
4098	65020	0.46	0.05	9.94	829.28

Table 1: Running times (in seconds) for the four implementations: LEDA preflow-push, Cherkassky-Goldberg highest-label preflow-push (`h_prf`), Dinitz, and the binary blocking flow algorithm (BBF) with $\Delta = 2\widehat{F}/\Lambda$ and $\beta = 1/2$.

spends time repeatedly scanning the entire graph when building layered networks. Although the BBF algorithm almost matches the LEDA preflow-push algorithm for the `rmf-Wide` instances, the overall running times are disappointing compared to `h_prf`. For the `rmf-Long` instances the blocking-flow computation takes so long that an implementation of an $O(m \log n)$-time blocking-flow algorithm using dynamic trees might be worth considering. The wave algorithm performs even worse here.

For the acyclic dense graphs, Dinitz' algorithm performs quite well; it needs only 3–4 rounds,

		h_prf	Dinitz		Binary blocking flow				
n	m	Push	Advance	Rnd	Comp	Advance	Ph	Rnd	Saved
Genrmf-Long (rmf-Long)									
675	2810	2685	15418	37	99	21663	2	4	29
1152	4956	5088	35007	50	51	45098	2	2	38
2205	9716	12286	109671	89	45	132939	2	1	54
4096	18368	21039	271421	122	64	357868	2	1	74
9000	41300	51619	922269	207	90	1271902	2	1	112
15488	71687	92775	2026817	249	177	2962492	2	1	146
30589	143364	196330	6125112	428	181	9560940	2	1	208
Genrmf-Wide (rmf-Wide)									
507	2210	6684	12735	46	3	5917	2	1	25
1024	4608	14399	34591	67	4	17436	2	1	36
2205	10164	40295	101495	99	5	54783	2	1	54
3920	18256	88670	252345	139	5	124296	2	1	73
8664	40964	220348	732811	199	6	389819	2	1	109
16807	80262	595050	1970327	294	7	1050095	2	1	152
32768	157696	1165655	4795368	387	8	2705728	2	1	214
Acyclic-Dense									
64	2016	356	365	3	4	342	3	4	16
128	8128	770	720	3	0	718	3	3	23
256	32640	1856	1801	3	1	1772	3	4	42
512	130816	5757	7626	4	2	7190	3	5	74
AK									
518	775	13823	42184	193	2539	17773	6	6	37
1030	1543	48369	166280	385	5083	68312	6	6	47
2054	3079	177819	660232	769	10179	267712	6	6	66
4102	6151	658507	2631176	1537	20387	1059749	6	6	108
8198	12295	2492647	10505224	3073	40821	4216690	6	6	131
16390	24583	9478275	41981960	6145	81716	16822070	6	6	220
32774	49159	36597865	167849992	12289	163545	67198678	6	6	263
Washington-RLG-Long (RLG-Long)									
4098	12224	33783	199145	26	62245	184646	12	116	205
8194	24512	74506	758686	47	254355	649599	12	197	289
16386	49088	138184	2857194	82	480664	2119305	14	278	430
Washington-RLG-Wide (RLG-Wide)									
8194	24448	102874	470967	40	166109	429112	13	152	239
16386	48896	191133	963366	45	400498	909273	14	203	344
Washington-Line-Moderate (wlm)									
514	3007.4	1310	9303	8	1570	5547	11	51	166
1026	8069.6	2432	29952	9	7145	20392	16	134	361
2050	22293.6	4499	70984	8	7490	36209	19	221	632
4098	65020	8745	196558	8	7572	92485	19	336	971

Table 2: Operation counts for the h_prf, Dinitz, and BBF algorithms. "Push" is the number of pushes performed by h_prf, and is roughly comparable to the number of "Advance" operations performed by the blocking-flow algorithm. Columns "Rnd" give the number of blocking-flow computations. "Comp" is the total number of SCCs identified by the BBF algorithm. "Ph" is the number of phases required. The column "Saved" estimates the number of rounds saved over the original Goldberg-Rao algorithm.

each of which is quite fast. The BBF algorithm behaves similarly, but suffers somewhat from the more complex construction of the acyclic core in each round. The h_prf implementation is an order of magnitude faster than the Dinitz implementation. Much smaller differences are reported in [1]

(less than a factor of 1.5 for small instances). Implementation details therefore seem to play a significant role, at least for this family of graphs.

The `AK` instances of [1] were specially designed to be difficult for the Dinitz and preflow-push algorithms. For the instances tried here, the BBF algorithm needs only six rounds. With our default blocking-flow algorithm, at least one of these computations needs quadratic time. The wave algorithm works very well and solves the instances in a single wave, i.e., in linear time.

The instances generated by the `washington` generator (`RLG-Long`, `RLG-Wide` and `wlm`) are very difficult for our BBF algorithm. In most rounds, the blocking flow is only slightly larger than Δ, so that the number of rounds comes close to the worst case predicted by the analysis. This is in fact much slower than even Dinitz' algorithm, which in turn is considerably slower than both of the preflow-push algorithms.

Table 3 summarizes our experiments with the refinements from Section 2.3. We give results (run-

Austere			Large AC (abc)			Large core (abd)			All refinements		
Time	Advance	Rnd	Time	Advance	Rnd	Time	Advance	Rnd	Time	Advance	Rnd
Genrmf-Long (`rmf-Long`) $n \in \{675, \ldots, 30589\}$											
0.37	54307	4	0.37	22431	4	0.16	17254	1	0.15	17057	1
0.47	138082	1	0.32	43780	1	0.31	42948	1	0.29	42116	1
1.26	446549	1	0.67	125951	1	0.77	132939	1	0.72	125951	1
3.75	1351338	1	1.66	330119	1	1.88	357868	1	1.69	330119	1
15.53	5249763	1	5.22	1171796	1	6.05	1271902	1	5.36	1171796	1
49.20	14015360	2	12.84	2761991	1	13.69	2948817	1	12.00	2743935	1
165.15	47303533	1	35.01	8651479	1	41.53	9560940	1	36.52	8651479	1
Genrmf-Wide (`rmf-Wide`) $n \in \{507, \ldots, 32768\}$											
0.09	7943	1	0.09	6204	1	0.11	5917	1	0.11	6204	1
0.21	30307	1	0.22	17754	1	0.23	17436	1	0.24	17754	1
0.60	119225	1	0.53	54260	1	0.59	54783	1	0.58	54260	1
1.32	279440	1	1.12	122971	1	1.23	124296	1	1.18	122971	1
4.31	993331	1	3.00	379599	1	3.37	389819	1	3.28	379599	1
13.41	3240827	1	7.21	1030765	1	8.00	1050095	1	7.67	1030765	1
38.27	9107654	1	16.77	2573170	1	18.74	2705728	1	17.58	2573170	1
Acyclic-Dense, $n \in \{64, 128, 256, 512\}$											
0.14	342	4	0.15	430	4	0.19	342	4	0.19	2292	4
0.46	718	3	1.29	738	6	0.61	718	3	1.64	4562	6
2.61	1772	4	2.62	1809	4	3.70	1772	4	3.13	15479	4
13.99	7190	5	24.03	7999	6	20.75	7189	5	33.09	350873	6
`AK`, $n \in \{518, \ldots, 32774\}$											
0.19	17773	6	0.20	17773	6	0.23	17771	6	0.23	17771	6
0.42	68312	6	0.42	68312	6	0.52	68310	6	0.50	68310	6
1.09	267712	6	1.07	267712	6	1.26	267710	6	1.31	267710	6
3.13	1059749	6	3.07	1059749	6	3.38	1059747	6	3.32	1059747	6
10.31	4216690	6	10.15	4216690	6	10.48	4216688	6	10.13	4216688	6
38.79	16822070	6	38.21	16822070	6	38.42	16822068	6	36.50	16822068	6
157.63	67198678	6	158.75	67198678	6	162.25	67198676	6	152.86	67198676	6
Washington-RLG-Long (`RLG-Long`), $n \in \{4098, 8194, 16386\}$											
57.14	188044	138	47.80	223810	93	77.42	252822	111	43.72	505970	61
175.85	671742	201	150.16	749208	138	248.47	867272	170	130.33	1779247	89
524.61	2139227	271	416.75	2553156	167	770.71	2287125	250	355.17	5728522	108
Washington-RLG-Wide (`RLG-Wide`), $n \in \{8194, 16386\}$											
135.78	441360	159	115.46	485229	112	181.22	601157	132	94.27	1100676	71
344.52	911538	189	258.49	971222	123	429.23	1171136	158	300.68	2849692	108
Washington-Line-Moderate (`wlm`), $n \in \{514, 1026, 2050, 4098\}$											
3.42	5573	50	1.19	16880	14	4.46	10200	40	0.70	16157	6
24.33	19841	135	5.73	25694	25	30.82	31507	100	4.99	49010	14
116.65	35935	217	13.33	54853	18	111.65	88762	126	12.05	110839	11
482.42	91895	328	22.25	169688	9	403.67	321261	157	11.61	377980	3

Table 3: Alternative AC and large-core constructions.

ning time, number of advance operations, and number of rounds) for four variations on this theme. An *austere* algorithm without any of the refinements; an algorithm with *large* acyclic components which encompasses refinements a), b) and c); a variant with enlarged core using refinements a), b) and d), and finally a variant using all the refinements described in Section 2.3 plus the following heuristic that is not covered by the analysis given here: We translate horizontal length-1 edges within SCCs into edges of the acyclic components in the same way as described in the refinements b) and c).

The austere variant potentially allows less flow through SCCs than the standard (and large) construction, and thus a larger number of rounds is to be expected. This effect is most manifest for the `rmf` instances and especially striking for the Washington-Line-Moderate (`wlm`)-class, where the "large" construction gives an improvement of a factor of more than 20 for the largest instance over the austere variant. For these instances, the standard construction is slightly better than the austere one. The gain from using large ACs comes (solely; the number of advance operations increases somewhat) from a dramatic reduction in the number of rounds, which, for the instances tried here, even decreases as the instance size increases.

The two variants with larger cores shown in the two rightmost columns seem to yield the most for the Washington-Line-Moderate (`wlm`)-family, although for the larger core construction alone, the effects are by no means as significant as for the large AC construction. Combining the two constructions brings the performance of the BBF algorithm very close to that of Dinitz' algorithm for the Washington-Line-Moderate (`wlm`)-instances. The `RLG`-families remain difficult for our variants of the BBF algorithm.

Dinitz' algorithm can be viewed as a degenerate version of the BBF algorithm with $\Delta = \infty$, so that all edges have length 1. It might therefore be conjectured that a larger Δ (that is, a larger factor α) is better for the `washington` instances. We conducted a few experiments concerning possibly better choices of Δ. Table 4 shows the effect of making Δ a factor of two smaller ($\Delta = \widehat{F}/\Lambda$) and a factor of two larger ($\Delta = 4\widehat{F}/\Lambda$) than the choice made for the other experiments. In addition, it shows measurements for the other extreme, $\Delta = 0$. A larger Δ improves the bleak situation for the `RLG` and `wlm` instances somewhat. This is not so surprising since it makes the BBF algorithm more Dinitz-like. A larger Δ has no adverse effect on the other instances. $\Delta = \widehat{F}/\Lambda$ correspondingly worsens the situation, and, more significantly, the `rmf-Long` instances now take a long time to solve. In fact, from the definition of these instances it seems likely that sufficiently "long" families of `rmf` instances are not solvable in a single round with $\Delta = \alpha\widehat{F}/\Lambda$ for any fixed α. The choice $\Delta = 0$, which has the effect of assigning all edges length 0 and is not allowed by the generic algorithm, has quite striking effects. Although it does not perform well for the acyclic-dense family or the `RLG` families, now not only the `rmf` instances but also `AK` instances and larger `wlm` instances are solved in a single round. The `AK` instance with 2^{15} nodes is solved in 3.87s if we use the wave blocking-flow algorithm. This is more than seven times faster than the `h_prf` code.

4. Discussion

Compared to the original algorithm of Goldberg and Rao [4], our variant offers a number of advantages. First, our algorithm potentially allows the flow from s to t to increase by substantially more than Δ in a single round, which appears from experiments to be crucial to a good performance. In contrast, Goldberg and Rao route flow in excess of Δ back to the source. Second, we do not need subroutines for routing flow within strongly connected components or for routing excess flow back to the source. Third, we can establish better worst-case bounds, by constant factors, on the number of rounds needed to compute a maximum flow. And fourth, our definition of an acyclic core is quite flexible. As a consequence, it is possible to experiment with a number of different heuristics for constructing the acyclic core without jeopardizing the worst-case bound on the running time.

Many further experiments as well as refinements of the algorithm and its analysis suggest them-

$\Delta = 0$			$\Delta = 1\widehat{F}/\Lambda$			$\Delta = 2\widehat{F}/\Lambda$			$\Delta = 4\widehat{F}/\Lambda$		
Time	Advance	Rnd	Time	Advance	Rnd	Time	Advance	Rnd	Time	Advance	Rnd
Genrmf-Long (`rmf-Long`) $n \in \{675, \ldots, 30589\}$											
0.16	17254	1	9.03	13762	108	0.48	21663	4	0.16	17254	1
0.31	42948	1	29.03	30979	186	0.43	45098	2	0.33	42948	1
0.79	132939	1	93.41	81177	286	0.77	132939	1	0.81	132939	1
1.88	357868	1	269.32	208976	413	1.90	357868	1	1.95	357868	1
6.03	1271902	1				6.02	1271902	1	6.12	1271902	1
13.30	2948817	1				15.00	2962492	1	13.38	2948817	1
40.98	9560940	1				40.30	9560940	1	40.89	9560940	1
Genrmf-Wide (`rmf-Wide`) $n \in \{507, \ldots, 32768\}$											
0.10	5917	1	0.11	5917	1	0.11	5917	1	0.12	5917	1
0.25	17436	1	0.25	17436	1	0.24	17436	1	0.25	17436	1
0.61	54783	1	0.63	54783	1	0.62	54783	1	0.63	54783	1
1.25	124296	1	1.30	124296	1	1.28	124296	1	1.29	124296	1
3.40	389819	1	3.47	389819	1	3.48	389819	1	3.47	389819	1
7.94	1050095	1	8.16	1050095	1	8.18	1050095	1	8.21	1050095	1
19.06	2705728	1	19.35	2705728	1	19.32	2705728	1	19.24	2705728	1
Acyclic-Dense, $n \in \{64, 128, 256, 512\}$											
11.12	6179	217	0.18	344	5	0.15	342	4	0.11	333	3
133.55	28766	664	0.49	718	3	0.47	718	3	0.48	720	3
			2.99	1772	4	2.92	1772	4	2.59	1767	4
			16.74	7190	5	16.43	7190	5	14.25	7178	5
`AK`, $n \in \{518, \ldots, 32774\}$											
0.07	17162	1	0.19	17665	5	0.21	17773	6	0.24	17839	7
0.20	67082	1	0.43	68088	5	0.46	68312	6	0.54	68497	7
0.59	265226	1	1.06	267244	5	1.12	267712	6	1.33	268126	7
1.97	1054730	1	2.93	1058781	5	3.13	1059749	6	3.21	1059620	6
7.43	4206602	1	9.62	4214725	5	9.91	4216690	6	10.07	4216516	6
34.88	16801802	1	40.27	16818087	5	38.77	16822070	6	39.87	16821830	6
145.49	67158026	1	151.52	67190647	5	151.59	67198678	6	150.70	67198351	6
Washington-RLG-Long (`RLG-Long`), $n \in \{4098, 8194, 16386\}$											
1028.85	267962	1732	143.67	172621	235	63.13	184646	116	44.95	187974	88
4423.69	1045270	3499	452.72	569092	340	229.95	649599	197	138.35	688417	131
						719.52	2119305	278			
Washington-RLG-Wide (`RLG-Wide`), $n \in \{8194, 16386\}$											
			384.79	418544	307	171.29	429112	152	90.52	445529	92
						482.81	909273	203			
Washington-Line-Moderate (`wlm`), $n \in \{514, 1026, 2050, 4098, 8194, 16386\}$											
0.60	16325	6	7.33	7178	74	4.51	5547	51	2.21	7378	26
1.24	57043	4	38.33	27314	140	34.08	20392	134	15.35	22362	61
1.43	151221	1	114.05	78619	148	165.10	36209	221	78.07	53690	103
4.54	457669	1	646.62	215525	302	829.28	92485	336	434.65	130904	201
12.76	1151407	1									
40.37	3564981	1									

Table 4: Different choices for the constant α in the formula for Δ for the binary blocking flow algorithm. Entries are left blank for instances that take too long to complete.

selves. In our experiments, we have not looked at how the parameter β should be chosen or how the maximum edge weight U influences the practical execution times. More experiments on how Δ should be chosen are needed since we have seen that this can have a significant effect. Can we even come up with a more adaptive way to choose Δ, e.g., as a function of $d_f(s)$ or of the value of the canonical cuts? Are there better alternatives for constructing a larger core? The experiments indicate that the possibility to route more flow through the strong components is always advantageous. Thus, it might be interesting to try out more possibilities for constructing the acyclic components.

One could also come closer to the original algorithm of Goldberg and Rao: After a blocking flow through the contracted graph is found, one would try to route as much flow as possible through the components. Only the remaining excess flow needs to be returned to s. We did not implement

this variant since it appears to be slightly more complicated than our algorithm and because the "capacity" of the vertex representing a component is invisible to the blocking-flow routine, a fact that can decrease the amount of flow that we actually get through. On the other hand, for instances with most nodes in large components, the contracted graph of the original algorithm is much smaller than the acyclic core constructed in our implementation. In such cases, significant savings in the blocking-flow computation can be expected. For example, for appropriate Δ the `rmf` instances collapse to a single path and routing the required flow within the components is easy, so that the overall running time of the algorithm would be linear.

References

[1] B. V. Cherkassky and A. V. Goldberg, On implementing the push-relabel method for the maximum flow problem, *Algorithmica* **19** (1997), pp. 390–410.

[2] E. A. Dinic, Algorithm for solution of a problem of maximum flow in networks with power estimation, *Soviet Math. Dokl.* **11** (1970), pp. 1277–1280.

[3] S. Even and R. E. Tarjan, Network flow and testing graph connectivity, *SIAM J. Comput.* **4** (1975), pp. 507–518.

[4] A. V. Goldberg and S. Rao, Beyond the flow decomposition barrier, Proc. 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1997), pp. 2–11.

[5] A. V. Goldberg and R. E. Tarjan, Finding minimum-cost circulations by successive approximation, *Math. Oper. Res.* **15** (1990), pp. 430–466.

[6] D. S. Johnson and C. C. McGeoch (eds.), *Network Flows and Matching, First DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, Amer. Math. Soc., 1993.

[7] D. E. Knuth, Wheels within wheels, *J. Combin. Theory (B)* **16** (1974), pp. 42–46.

[8] K. Mehlhorn and S. Näher, LEDA: A platform for combinatorial and geometric computing, *Communications of the ACM* **38**(1) (1995), pp. 96–102.

[9] K. Mehlhorn, S. Näher, and C. Uhrig, *The LEDA User Manual. Version 3.5*, Max-Planck-Institut für Informatik, Saarbrücken, 1997.

[10] R. E. Tarjan, A simple version of Karzanov's blocking flow algorithm, *Oper. Res. Lett.* **2** (1984), pp. 265–268.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 155–166

An Iterated Heuristic Algorithm for the Set Covering Problem

Elena Marchiori
Department of Computer Science, University of Leiden, P.O. Box 9512
2300 RA Leiden, The Netherlands
e-mail: elena@cs.leidenuniv.nl

and

Adri Steenbeek
CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
e-mail: adri@cwi.nl

ABSTRACT

The set covering problem is a well-known NP-hard combinatorial optimization problem with wide practical applications. This paper introduces a novel heuristic for the uni-cost set covering problem. An iterated approximation algorithm (ITEG) based on this heuristic is developed: in the first iteration a cover is constructed, and in the next iterations a new cover is built by starting with part of the best solution so far obtained. The final output of the algorithm is the best solution obtained in all the iterations. ITEG is empirically evaluated on a set of randomly generated problem instances, on instances originated from the Steiner triple systems, and on instances derived from two challenging combinatorial questions by Erdös. The performance of ITEG on these benchmark problems is very satisfactory, both in terms of solution quality (i.e., small covers) as well as in terms of running time.

1. Introduction

The set covering problem (SCP) is one of the oldest and most studied NP-hard problems (cf. [13]). The SCP consists of covering the rows of a m-row, n-column, zero-one matrix (a_{ij}) by a minimal subset of the columns, where a row i is covered by a column j if the entry a_{ij} is equal to 1. This problem can be formulated as a constrained optimization problem as follows:

$$\text{minimize} \quad \sum_{j=1}^{n} x_j \quad \text{subject to the constraints} \quad \begin{cases} x_j \in \{0,1\} & j = 1,\ldots,n, \\ \sum_{j=1}^{n} a_{ij} x_j \geq 1 & i = 1,\ldots,m. \end{cases}$$

The variable x_j indicates whether column j belongs to the solution ($x_j = 1$) or not ($x_j = 0$). The m constraint inequalities are used to express the requirement that each row be covered by at least one column. In the weighted version of this problem, the objective function is $\sum_{j=1}^{n} c_j x_j$ for some weights $c_j > 0$ specifying the cost of column j. Therefore the SCP is sometimes referred to as the unicost SCP.

The SCP has a wide number of applications, such as construction of optimal logical circuits, scheduling (e.g., aircrew scheduling), assembly line balancing, and information retrieval. Recent

results in computational complexity provide a theoretical quantification of the difficulty of this problem. In particular, Feige in [11] has proven that for any $\epsilon > 0$ no polynomial time algorithm can approximate SCP within $(1-\epsilon)\ln m$ unless NP has slightly superpolynomial time algorithms. Due to these negative results, many approximation algorithms for the SCP have been developed (e.g., [7, 6, 14]). In particular, in [14] Grossman and Wool perform a comparative experimental study of nine effective approximation algorithms proposed by researchers in the field of combinatorial optimization, like the simple and randomized greedy algorithms, a randomized rounding algorithm, and an algorithm based on neural networks. The performance of these algorithms is tested on a large set of benchmark problems, and the results show that the multi-start randomized greedy algorithm outperforms the other algorithms in almost all of the problem instances. The heuristic used in the randomized greedy algorithm is rather simple: a cover is constructed starting from the empty set of columns (that is, all x_j's are equal to zero). At each step of the algorithm the actual set of columns is extended by adding one column which is randomly chosen amongst those that cover the largest number of (not yet covered) rows (in other words, at each step of the algorithm the variable x_j that appears in the largest number of unsatisfied inequality constraints is set to one). This process is repeated until all rows are covered. Observe that the obtained cover may contain redundant columns, which can be removed from the set without destroying the property of being a cover. Therefore, the cover is refined by removing the redundant columns in a random order, one at a time because removing one redundant column may affect the redundancy of other columns. The resulting procedure can be summarized as follows:

```
Random_Greedy
BEGIN
   S <- { };
   WHILE ( S is not a cover ) DO
      Add best column to S breaking ties randomly;
   ENDWHILE
   remove redundant columns from S in a random order;
   RETURN S
END
```

In this paper, we enhance `Rand_Greedy (RG)` as follows: a suitable rule is used for breaking ties in the 'add' step; moreover, a 'removal' step is introduced in the `while` loop, which makes it possible to discard columns from the partial cover so far constructed, and which also takes care of removing redundant columns; finally, an optimization step is introduced after the `while` loop, which tries to improve the obtained cover by means of a suitable optimization procedure. The resulting heuristic is called `Enhanced Greedy (EG)`.

Often, the basic algorithm `Rand_Greedy` is run a number of times and the best solution found in all the runs is returned (*multistart approach*). An alternative to the multistart approach is the so-called *iterated approach* (cf., e.g., [1]), in which the starting point of the next run is obtained by modifying the local optimum of a previous run. For a greedy heuristic for the SCP, this approach amounts to starting each run with a partial cover instead of the empty set. It turns out that a 'good' partial cover can be obtained by restoring a part of the best cover found in all the previous iterations. By applying this latter approach to `EG`, we obtain a very powerful algorithm, called `Iterated Enhanced Greedy Algorithm (ITEG)`, which produces covers of very good quality in a competitive amount of time.

The rest of the paper is organized as follows. In the next section we introduce the `EG` heuristic. In section 3 the `ITEG` algorithm is introduced. The computational experiments are treated in Section 4, while Section 5 contains a discussion on the iterated approach applied to `RG` and `EG`. We conclude with some final remarks on the present investigation and on future work.

2. EG: Enhanced Greedy Heuristic

In order to enhance `Random_Greedy`, we use a more advanced selection criterion for choosing a best column to be added. Moreover, we allow also for the removal of columns which have been previously added to the partial cover so far constructed, even if they are not redundant. The motivation for this latter step is that as more columns are added, a previously added column may become 'almost' redundant, hence it is discarded in the expectation that better alternative columns exist. The idea of incorporating a step for removing columns is also used in other variants of the greedy algorithm (see e.g. alternating greedy in [14]). However, we use a novel criterion, which is similar to the one we employ for adding columns.
When the construction of a cover is completed by means of this add-remove strategy, we apply a suitable optimization procedure in order to improve the cover.

Our set covering algorithm is described in pseudo-code below. Lines starting with "//" are comments. `EG` constructs a solution (a cover), starting from a (possibly empty) set S of columns. Then columns are added to S until S covers all the rows.

```
// extend S until it is a  cover:
FUNCTION EG( var S )
BEGIN
   WHILE ( S is not a cover ) DO
      // select and add one column to S
      S <- S + select_add();
      // remove 0 or more columns from S
      WHILE ( remove_is_okay() ) DO
         S <- S - select_rmv();
      ENDWHILE
   ENDWHILE
   // S is a cover, without redundant columns
   // apply local optimization
   S <- optimize(S);
   return S;
END
```

Below we explain the key functions of this algorithm.
cover value: For every column c we define the *cover value* $cv(c)$ to be the number of rows that are covered by c, but that are not covered by any column in $S \setminus \{c\}$. Note that $cv(c)$ is defined both if $c \in S$ and if $c \notin S$, and that $cv(c)$ does not change if c is removed from, or added to S. A column c in S is *redundant* if $cv(c) = 0$. The first criterion for selecting a column in both *select_add* and *select_rmv* is its cover value.

select_add: This function returns the next column that is to be added to S.
Let $Candidates = \{c \in \overline{S} \mid cv(c) = k\}$, where $k = max\{cv(c) \mid c \in \overline{S}\}$, and $\overline{S}$ denotes the set of columns which are not in S. Every column c in *Candidates* is evaluated by means of a merit criterion specified by the function `w_add(c)` which computes the so-called *add_value* of c. Possible choices of `w_add` are discussed in the next subsection. Then, one column is randomly selected amongst those having highest add_value. However, with low probability *param.add_rand* (with typical value 0.05) the columns are not evaluated, and select_add randomly selects a column in *Candidates*; this turns out to be useful for escaping from local optima.
The size of the set *Candidates* is bounded by a constant *param.max_candidates* (typical value 400). In case the number of candidates exceeds this threshold, a subset of *Candidates* containing *param.max_candidates* elements is randomly chosen. This situation typically occurs if *select_add* is called when S is (almost) empty.

remove_is_okay: This function returns a boolean value. It determines whether columns should be removed from S. If S is empty it returns $false$; if S contains at least one redundant column then it returns $true$; otherwise, with low probability $param.p\_rmv$ (typical value 0.1) it returns $true$, otherwise $false$.

select_rmv: This function returns a column that is to be removed from S. The definition is very similar to that of *select_add*. Let $Candidates = \{c \in S \mid cv(c) = k\}$, where $k = min\{cv(c) \mid c \in S\}$. Every column c in *Candidates* is evaluated by means of a merit criterion specified by the function `w_rmv(c)` which computes the so-called remove_value of c. Possible choices of `w_rmv` are discussed in the next subsection. Then, one column is randomly selected amongst those having highest rmv_value. However, with low probability $param.rmv\_rand$ (with typical value 0.05) the columns are not evaluated, and select_rmv randomly selects a column in $Candidates$.

optimize: This function tries to improve a cover, by identifying and replacing 'inferior' columns, defined as follows. Given a cover S, we call a column $c \notin S$ superior if the addition of c to S renders at least two (other) columns of S redundant. Then a column c in S is called *inferior* if there exists a superior column which, if added to S, would make c redundant. First, the set *Inf* of inferior columns is identified; next, all elements of *Inf* are removed from S; finally, `add_select` is called repeatedly to select and add a column, until a cover is obtained. Note that the function `optimize` operates on a cover containing no redundant columns.

2.1. The Merit Functions w_add() and w_rmv().

The merit functions `w_add()` and `w_rmv()` specify the rule to be used for breaking ties in the selection of the best column amongst those having equal maximum (resp. minimum) cover value. If `w_add(c)` returns a high value, then it means that it is convenient to add c to S; similarly, a high value of `w_rmv(c)` means that c is a good column to be removed from S. Two alternative definitions of `w_add()` and of `w_rmv()` are intoduced:

• The first `w_add()` function is based on the idea that it is good to add a column c to S if c covers rows that are not yet covered by 'too many' other columns in S. If a row is already covered by several other columns in S, the extra covering due to c has no relevant effect; however, if a row is covered by only one column of S, say x, then adding c to S helps to make column x redundant.

For a row r let `xcover(r)` denote the number of columns in S that cover r. Note that every c in $Candidates$ covers the same number of rows for which $xcover(r) = 0$, namely $cv(c)$. For a column c let $R(c)$ denote the set of rows that are covered by c. Then the `w_add()` and the corresponding `w_rmv()` functions are defined below in pseudo-code:

```
FUNCTION w_add1(c) // column c is not in S
BEGIN
   w <- 0;
   FOR all rows r in R(c) DO
      w <- w + 1/( xcover(r)+1 )^2;
   ENDFOR
   return w;
END
```

```
FUNCTION w_rmv1(c)  // column c is in S
BEGIN
   w <- 0;
   FOR all rows r in R(c) DO
      w <- w + 1/( xcover(r) )^2;
   ENDFOR
   return -w;
END
```

• The second merit function is based on the idea of directly trying to make columns in S redundant or almost redundant. To this aim, it considers the effect of adding column c to the partial cover S with respect to the cover values of the columns in S. The corresponding `w_add()` and `w_rmv()` functions are defined as follows:

```
FUNCTION w_add2(c)  // column c is not in S
BEGIN
   S <- S + {c};
   w <- 0;
   FOR all columns u in S DO
      w <- w + 1/( cv(u) + 0.01 );
   ENDFOR
   S <- S - {c};
   return w;
END
```

```
FUNCTION w_rmv2(c) // column c is in S
BEGIN
   S <- S - {c};
   w <- 0;
   FOR all columns u in S DO
      w <- w + 1/( cv(u) + 0.01 );
   ENDFOR
   S <- S + {c};
   return -w;
END
```

3. ITEG: Iterated Enhanced Greedy

The performance of EG is reasonably satisfactory. Nevertheless, it can be improved by applying a suitable iterated technique. Roughly, in the first iteration EG constructs a cover starting from the empty set; in the following iterations, EG builds a cover starting from a subset of the best cover found in all the previous iterations. The final result is the best cover found in all iterations. The corresponding algorithm is illustrated below in pseudo-code, where |S| denotes the number of elements of the list (or set) S.

```
FUNCTION ITEG()
BEGIN
   Sbest <- { 1..ncol };  // best solution so far
   S <- { };    // current partial solution
   FOR 1 .. param.number_of_iterations DO
      choose_add_strategy();
      choose_remove_strategy();
      S <- EG(S);
      // if S better than Sbest, replace Sbest by S
      IF ( |S| <= |Sbest| ) THEN Sbest <- S; ENDIF;
      // make S a "random" subset of Sbest,
      S <- random_selection(Sbest);
      // ready for the next iteration
   ENDFOR
   RETURN Sbest
END
```

Every iteration starts with a partial solution S. First, an add/remove strategy is selected: one merit function for the column addition is randomly selected between w_add1() or w_add2(), and analogously for the column removal. Next, EG is applied starting with partial cover S, and with the previously chosen merit functions. The cover S produced by EG is used to update the best solution *Sbest* so far obtained: if S has smaller or equal size than *Sbest* than it becomes the new *Sbest*. Note that we replace *Sbest* with S even if S and *Sbest* have the same size, in order to exploit different covers. Finally, S is initialized for the next iteration, where we will start with a subset of *Sbest*; every column of *Sbest* is selected to be in S with probability p, the *restore-fraction*. Every iteration the value of p is randomly chosen from the interval [param.rcv_low , param.rcv_high], with typical value $[0.6, 0.8]$. The idea is that *Sbest*, being the best solution so far, possibly contains a subset of 'relevant' columns which can be crucial to build a minimum cover. Section 5 we investigate the effect of using different (fixed) values for the restore fraction.

Problem	Iter 1 Avg (StDv)	Iter 10 Avg (StDv)	Iter 100 Avg (StDv)	Iter 1000 Avg (StDv)	Best ITEG	Best Known
STS.135	106.9 (0.3)	105.5 (0.5)	104.9 (0.3)	104.0 (0.0)	104	103
STS.243	209.7 (1.2)	203.2 (0.4)	203.0 (0.0)	202.2 (1.5)	198	198
STS.27	19.0 (0.0)	18.2 (0.4)	18.0 (0.0)	18.0 (0.0)	18	18*
STS.45	32.0 (1.0)	31.0 (0.0)	31.0 (0.0)	30.7 (0.5)	30	30*
STS.81	65.0 (0.0)	62.2 (1.0)	61.2 (0.6)	61.0 (0.0)	61	61*
Seymour	434.0 (1.2)	429.6 (1.3)	425.5 (1.3)	423.7 (0.6)	423	423

Table 1: Results on Combinatorial Problems (STS and Seymour)

4. Computational Experiments

In this section, we evaluate empirically the performance of the EG heuristic algorithm on the set of test problems for the unicost SCP taken from the OR-library[1] maintained by J.E. Beasley. These problems provide a valuable source for testing the performance of algorithms for SCP, because they arise from various different applications. Almost all the problem instances we consider have also been used in [14] for comparing nine heuristic algorithms for SCP. More specifically, the following problem instances are considered:

• **Random Problems** These are 70 randomly generated problems from the OR-library (see Tables 4 and 5). The instances of families 4-6 are from [3], those of families A-E are from [4], and those of families NRE-NRH are from [5]. As in [14], the original instances have been modified by discarding the costs of the columns, since we are dealing with the unicost SCP. Thus the results obtained are not comparable with those reported in the above-mentioned papers (except for the E family, which is also originally produced for the unicost SCP).

• **Combinatorial Problems** We have considered 16 problem instances arising from four different combinatorial questions (see Tables 3, 1). The CYC and CLR sets are from [14], and are described in more detail there. They are available in the OR-library. The STS-set consists of 5 problems from a class of set systems known as *Steiner triple systems*, introduced in [12]: these instances are known to be rather difficult for any branch and bound algorithm (cf., [2, 15]), and they have been often used as benchmark problems to test the performance of heuristic algorithms (cf., e.g., [17, 16]). Finally, the Seymour problem instance [2] is provided by P. Seymour. It specifies an SCP arising from work related to the proof of the 4-Color Theorem.

Characteristic parameters of the above problem instances, like number of rows and columns, are described in Table 2.

The ITEG algorithm has been implemented in C++. Parameters, like those specifying the probabilities of addition and removal, or the size of the portion of a cover to be restored at each iteration of ITEG have been set to a suitable value for each class of problem instances. For instance, for the CYC instances, param.add_rand is set to 0.15, param.rmv_rand to 0.10, param.p_rmv to 0.25, param.max_candidates is set to 500, and the restore-fraction is every iteration randomly selected in the interval $[0.4, 0.7]$.

The algorithm was run on a multi-user Silicon Graphics IRIX Release 6.2 IP25 (194 MHZ MIPS R10000 processor, Main memory size: 512 Mbytes). The results of the experiments are based on 10 runs of ITEG on each problem instance, using a different initial seed for the random-generator in every run. The results are summarized in Tables 1-5. In each table, the first column contains the name of the problem instance; columns with label of the form Iter k Avg (StDv) indicate the average of the results of the 10 runs of ITEG obtained after k iterations (i.e., param.iterations=k), and the

[1] available via WWW at http://mscmga.ms.ic.ac.uk/jeb/orlib/scpinfo.html

[2] available electronically at ftp://ftp.caam.rice.edu/pub/people/bixby/miplib/miplib3/seymour

Problem	Rows (m)	Columns (n)	Density (%)	Avg CPU (sec/iter)
4	200	1000	2	0.01
5	200	2000	2	0.01
6	200	1000	5	0.06
A	300	3000	2	0.03
B	300	3000	5	0.06
C	400	4000	2	0.04
D	400	4000	5	0.11
E	50	500	20	0.01
NRE	500	5000	10	0.34
NRF	500	5000	20	0.66
NRG	1000	10000	2	0.26
NRH	1000	10000	5	0.61
CYC.6	240	192	2.1	0.00
CYC.7	672	448	0.9	0.01
CYC.8	1792	1024	0.4	0.05
CYC.9	4608	2304	0.2	0.19
CYC.10	11520	5120	0.08	1.10
CYC.11	28160	11264	0.04	5.00
CLR.10-4	511	210	12.3	0.01
CLR.11-4	1023	330	12.4	0.02
CLR.12-4	2047	495	12.5	0.05
CLR.13-4	4095	715	12.5	0.11
STS.27	117	27	0.1	0.001
STS.45	330	45	0.06	0.002
STS.81	1080	81	0.04	0.011
STS.135	3015	135	0.02	0.015
STS.243	9801	243	0.01	0.100
Seymour	4944	1372	0.5	0.018

Table 2: Details of Problem Instances and Average CPU Times

Problem	Iter 1 Avg (StDv)	Iter 10 Avg (StDv)	Iter 100 Avg (StDv)	Iter 1000 Avg (StDv)	Best ITEG	Best RG	Best GW
CYC.6	63.2 (0.9)	61.8 (0.7)	61.4 (0.8)	-	60	64	60*
CYC.7	155.8 (1.8)	151.2 (0.9)	148.0 (2.1)	-	144	160	144
CYC.8	371.0 (1.3)	360.6 (4.2)	351.8 (2.6)	-	348	385	352
CYC.9	862.5 (3.1)	839.0 (3.1)	827.6 (1.6)	-	825	907	816
CYC.10	1973.3 (6.1)	1893.6 (17.1)	1860.7 (2.9)	-	1858	2081	1916
CYC.11	4450.0 (11.7)	4262.8 (15.7)	4218.3 (9.1)	-	4202	4710	4268
CLR.10-4	28.6 (1.6)	25.8 (1.1)	25.1 (0.3)	25.0 (0.0)	25	28	28
CLR.11-4	28.8 (0.6)	26.9 (1.8)	24.0 (1.6)	23.0 (0.0)	23	27	27
CLR.12-4	27.2 (1.0)	24.7 (1.3)	23.0 (0.0)	23.0 (0.0)	23	27	27
CLR.13-4	30.9 (0.5)	29.9 (0.3)	28.2 (1.0)	25.3 (2.4)	23	31	29

Table 3: Results on Combinatorial Problems (CYC and CLR)

Problem	Iter 1 Avg (StDv)	Iter 10 Avg (StDv)	Iter 100 Avg (StDv)	Iter 1000 Avg (StDv)	Best ITEG	Best RG	Best GW
4.1	40.8 (0.8)	39.6 (0.8)	38.9(0.3)	38.0(0.0)	38	41	41
4.2	39.1 (0.9)	38.3 (1.0)	37.1(0.3)	37.0(0.0)	37	38	38
4.3	40.7 (0.6)	39.3 (0.6)	38.2(0.4)	38.0(0.0)	38	41	40
4.4	42 (0.9)	40.6 (0.6)	39.9(0.5)	39.1(0.3)	39	41	41
4.5	40.4 (0.5)	39.6 (0.5)	38.8(0.4)	38.0(0.0)	38	40	40
4.6	39.9 (0.7)	38.9 (0.7)	37.9(0.3)	37.8(0.4)	37	40	40
4.7	41.4 (0.6)	40.0 (0.4)	38.7(0.7)	38.4(0.5)	38	41	41
4.8	40.4 (0.6)	39.6 (0.6)	38.1(0.5)	37.7 (0.4)	37	40	40
4.9	40.7 (0.4)	39.7 (0.4)	39.0(0.4)	38.1 (0.3)	38	40	40
4.10	41.2 (0.7)	40.1 (0.5)	39.2 (0.4)	38.6 (0.5)	38	41	41
5.1	37.0 (0.7)	36.1 (0.7)	35.3 (0.4)	34.9 (0.3)	34	35	35
5.2	36.6 (0.5)	35.9 (0.5)	35.0 (0.0)	34.7 (0.4)	34	35	35
5.3	36.1 (0.7)	35.5 (0.8)	34.5 (0.5)	34.0 (0.0)	34	36	36
5.4	36.0 (0.6)	35.4 (0.5)	34.2 (0.4)	34.0 (0.0)	34	36	36
5.5	36.0 (1.0)	35.2 (0.6)	34.3 (0.4)	34.1 (0.3)	34	36	36
5.6	36.5 (0.5)	35.7 (0.4)	34.8 (0.4)	34.5 (0.5)	34	36	36
5.7	36.0 (0.9)	35.3 (0.4)	34.6 (0.5)	34.0 (0.0)	34	36	35
5.8	37.5 (0.8)	36.7 (0.6)	35.1 (0.3)	34.9 (0.3)	34	37	37
5.9	37.7 (0.4)	36.4 (0.6)	35.6 (0.5)	35.0 (0.0)	35	36	36
5.10	37.0 (0.6)	36.2 (0.6)	35.4 (0.6)	34.6 (0.5)	34	36	36
6.1	22.0 (0.0)	21.8 (0.4)	21.2 (0.4)	21.0 (0.0)	21	21	21
6.2	21.4 (0.5)	21.0 (0.0)	21.0 (0.0)	20.3 (0.4)	20	22	21
6.3	22.1 (0.3)	21.7 (0.4)	21.0 (0.0)	21.0 (0.0)	21	22	22
6.4	22.2 (0.4)	22.0 (0.4)	21.5 (0.5)	21.0 (0.0)	21	22	22
6.5	22.5 (0.5)	21.9 (0.5)	21.3 (0.4)	21.0 (0.0)	21	22	22
A1	41.6 (0.6)	40.5 (0.7)	39.5 (0.5)	39.1 (0.3)	39	40	40
A2	41.9 (0.3)	41.4 (0.5)	39.8 (0.6)	39.1 (0.3)	39	40	41
A3	41.2 (0.7)	40.3 (0.6)	39.6 (0.5)	39.0 (0.0)	39	40	40
A4	39.6 (0.8)	39.0 (0.6)	38.3 (0.4)	38.0 (0.0)	38	40	40
A5	40.7 (0.7)	40.1 (0.5)	39.0 (0.0)	38.7 (0.4)	38	40	40
B1	23.7 (0.6)	23.4 (0.5)	22.2 (0.4)	22.0 (0.0)	22	23	23
B2	23.1 (0.3)	22.9 (0.3)	22.2 (0.4)	22.0 (0.0)	22	22	22
B3	23.3 (0.4)	23.0 (0.0)	22.3 (0.4)	22.0 (0.0)	22	22	22
B4	23.7 (0.4)	23.3 (0.4)	22.7 (0.4)	22.0 (0.0)	22	23	23
B5	23.4 (0.5)	23.0 (0.0)	22.5 (0.5)	22.2 (0.4)	22	23	23
C1	45.7 (0.4)	44.9 (0.5)	44.0 (0.4)	43.5 (0.5)	43	45	45
C2	46.0 (0.7)	45.4 (0.6)	44.1 (0.3)	43.5 (0.5)	43	45	45
C3	45.7 (0.6)	45.2 (0.6)	44.1 (0.3)	43.6 (0.5)	43	45	45
C4	45.6 (0.6)	44.9 (0.9)	44.1 (0.7)	43.1 (0.3)	43	46	46
C5	45.8 (0.9)	45.0 (0.7)	44.0 (0.0)	43.5 (0.5)	43	45	45
D1	26.5 (0.5)	26.2 (0.4)	25.3 (0.4)	25.0 (0.0)	25	26	26
D2	26.2 (0.4)	26.0 (0.4)	25.5 (0.5)	25.0 (0.0)	25	26	25
D3	26.2 (0.4)	25.8 (0.4)	25.4 (0.5)	25.0 (0.0)	25	25	25
D4	26.3 (0.7)	26.1 (0.7)	25.5 (0.5)	25.0 (0.0)	25	26	26
D5	26.6 (0.5)	26.1 (0.3)	25.5 (0.5)	25.0 (0.0)	25	26	26
E1	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5	5	5
E2	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5	5	5
E3	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5	5	5
E4	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5	5	5
E5	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5.0 (0.0)	5	5	5

Table 4: Results on Random Problems

Problem	Iter 1 Avg	Iter 10 Avg	Iter 100 Avg	Best ITEG	Best RG	Best GW
NRE1	17.9 (0.3)	17.7 (0.5)	17.0 (0.0)	17	17	17
NRE2	17.2 (0.4)	17.1 (0.3)	17.0 (0.0)	17	17	17
NRE3	17.2 (0.4)	17.1 (0.3)	17.0 (0.0)	17	17	17
NRE4	17.2 (0.4)	17.1 (0.3)	17.0 (0.0)	17	17	17
NRE5	17.8 (0.4)	17.6 (0.5)	17.2 (0.4)	17	17	17
NRF1	11.0 (0.0)	10.8 (0.4)	10.3 (0.5)	10	10	10
NRF2	11.0 (0.0)	11.0 (0.0)	10.4 (0.5)	10	11	11
NRF3	11.0 (0.0)	11.0 (0.0)	10.6 (0.5)	10	11	11
NRF4	11.0 (0.0)	11.0 (0.0)	10.5 (0.5)	10	11	11
NRF5	11.0 (0.0)	10.9 (0.3)	10.7 (0.5)	10	11	11
NRG1	64.2 (0.6)	63.4 (0.7)	62.4 (0.5)	62	-	-
NRG2	64.0 (0.8)	63.4 (0.7)	62.5 (0.5)	62	-	-
NRG3	64.1 (0.7)	63.4 (0.7)	62.8 (0.6)	62	-	-
NRG4	64.4 (0.5)	63.7 (0.8)	63.2 (0.6)	62	-	-
NRG5	64.4 (0.5)	64.0 (0.4)	62.7 (0.5)	62	-	-
NRH1	35.5 (0.5)	35.2 (0.6)	34.8 (0.4)	34	-	-
NRH2	35.5 (0.5)	35.0 (0.4)	34.7 (0.5)	34	-	-
NRH3	35.5 (0.5)	35.2 (0.6)	34.8 (0.4)	34	-	-
NRH4	35.5 (0.5)	35.2 (0.6)	35.0 (0.4)	34	-	-
NRH5	35.5 (0.5)	35.3 (0.6)	34.6 (0.5)	34	-	-

Table 5: Results on Random Problems

corresponding standard deviation (written between brackets). We consider the value obtained after 1 iteration, after 10 iterations, and so on, until the maximal number of iterations considered, which is set to 1000, or to 100 for bigger instances like the CYC ones. This is useful for illustrating the values of the best solutions found during the iterations made by ITEG. A column labeled Best RG contains the best result found by Random_Greedy when run 100 times, and Best GW contains the best result found by all the nine algorithms considered in [14]. Finally, Best Known is the best known solution. Values are labeled with a '*' if they have been proven (by means of exact algorithms) to be global optima. Entries containing only the symbol '-' indicate that the relative problem instance is not considered in [14].

The quality of the results found by ITEG on these problem instances is very satisfactory.
On the random problems the best cover found by ITEG is always better or equal to the best cover found by all the nine algorithms considered in [14]. In 45 cases out of 60 instances it is strictly better.
On the CYC instances (Table 3) ITEG finds better solutions on 3 out of 6 instances, and equivalent best solutions on other two instances; however, on CYC.9 it does not perform very well if compared with the best result reported in [14]. The random Greedy algorithm RG finds solutions of rather poor quality on all the CYC instances. However, the simple Greedy algorithm, where ties are broken lexicographically instead of randomly, is one of the algorithms in [14] which is performing very well on the CYC instances. This suggests that for the CYC problems the lexicographical order as used in simple Greedy is favourable.
On all the CLR instances (Table 3) ITEG finds better solutions than all the algorithms in [14].
Finally, on the other six instances of combinatorial problems (Table 1) ITEG finds the best known solution for all but one instance, namely STS.135, where the best known solution is 103, while ITEG can only find 104. The best known solution for STS.135 has been found in [17]: the authors develop a heuristic algorithm for solving the Steiner triple covering problem based on GSAT [18] (a popular method for solving satisfiability problems). Unfortunately, a rather poor discussion of their

experiments is reported, which makes it difficult to judge the performance of their algorithm.

The effectiveness of the heuristic EG employed in ITEG can be evaluated by considering the column corresponding to the results for 1 iteration. It can be seen that the quality of the solutions found by EG is satisfactorily. However, as illustrated by the results of the experiments, the iterated application of EG starting with a portion of the best solution found so far, yields a substantial improvement of the quality of the solutions, especially when applied to 'hard' instances like those relative to the combinatorial optimization problems. In the next section, we shall study in more detail the effect of the iterated approach also on the Random_Greedy algorithm.

It is difficult to perform a fair comparison between ITEG and Random_Greedy based on the results given in [14] and here reported in the columns labeled RG, because those results are based on 100 independent runs of Random_Greedy (multistart approach). A rough indication of the relative performance of these algorithms can be obtained by comparing the column for 100 iterations and the column labeled RG for getting an impression about the quality of the solutions. Table 2 contains the average running time per iteration of ITEG on the considered problem instances. RG is about five times slower, since it always starts from the empty set.

5. Discussion

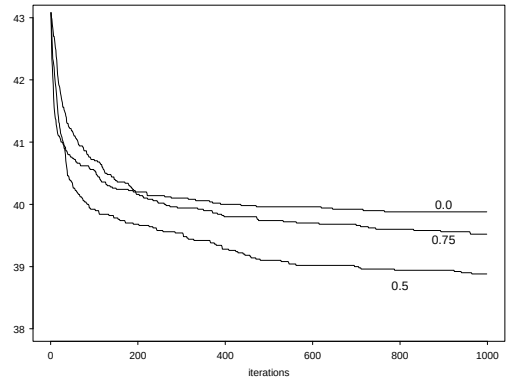
The choice of the restore-fraction in the iterated EG is relevant for the quality of the results. In general, the optimal value for this parameter depends on the specific problem instance considered. However, the experiments we have conducted on the considered problem instances seem to indicate that good values of the restore-fraction are between 0.40 and 0.75, where for smaller problems it is better to choose a rather low value for the restore-fraction.

In order to illustrate the effect of different values for the restore-fraction, we have plotted in Figure 4 the courses of ITEG and of the iterated Rand_Greedy (ITRG) on three specific problem instances, when the restore-fraction is fixed at 0.0 (that is multistart approach), 0.5 and 0.75. This is also useful for comparing the iterated Rand_Greedy with ITEG . On each case we had 50 runs with different random seeds. The x-axis represents the number of iterations (up to 1000), and the y-axis represents the average number (over 50 runs) of columns contained in the best solution computed at a given iteration of the algorithm. On all the problem instances we have that, for a fixed value of the restore-fraction, iterated EG outperforms ITRG Rand_Greedy. This is not surprising, because EG employs a more sophisticated heuristic. As a consequence, we have that ITRG is roughly three or four times faster than iterated EG.

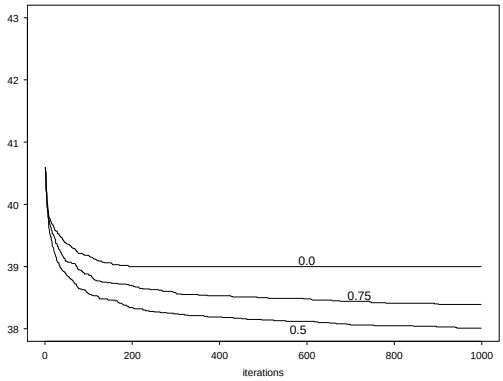
On the problem instance 4.1 of the random generated problems, we see that the multistart version of both EG and Rand_Greedy performs rather poorly, while the best results are obtained by considering a restore-fraction equal to 0.5.
A rather different behaviour is illustrated in the second pair of plots concerning the CLR problem instance CLR.12-4. Here Rand_Greedy gives better results when run using the multistart approach, while the EG gives better results when a high restore-fraction of 0.75 is used. This shows that the choice of a good restore-fraction depends also on the heuristic used.
Finally, on the Seymour instance we see that Rand_Greedy and EG follow a similar course, with a performance that dramatically improves when a restore-fraction of 0.0 is replaced with one of 0.5.

6. Conclusion

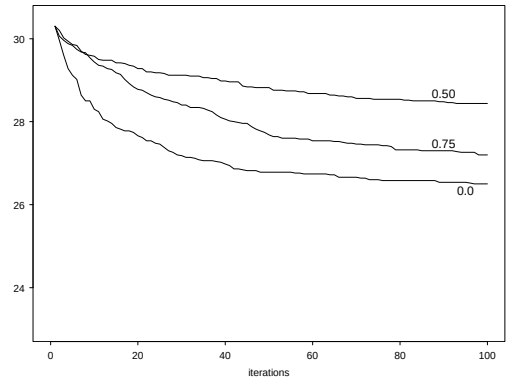
In this paper we have introduced a novel iterated heuristic ITEG for the set covering problem, and have studied its performance on a large set of benchmark problems. The results indicate that



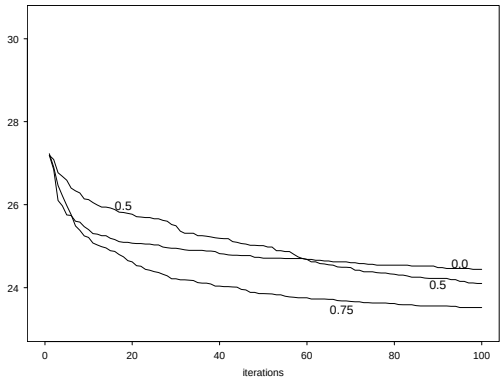
ITRG on instance 4.1



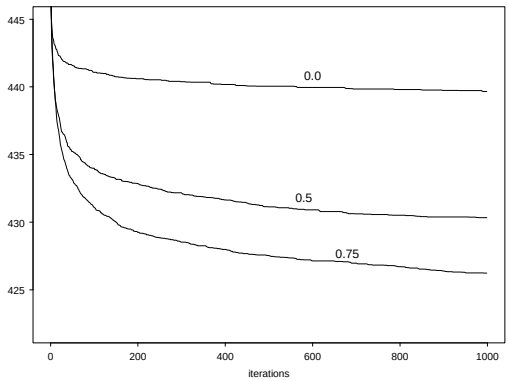
ITEG on instance 4.1



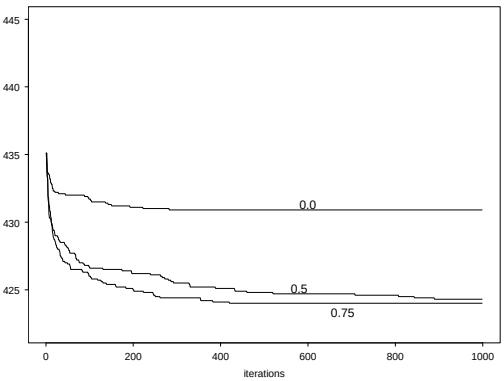
ITRG on instance CLR.12-4



ITEG algorithm on instance CLR.12-4



ITRG on instance Seymour



ITEG algorithm on instance Seymour

Figure 1: Behaviour of ITEG and ITRG with varying restore fractions

ITEG can produce covers of very good quality in competitive running time.

Future work concerns the study of a similar iterated heuristic for the weighted set covering problem. Various papers are dedicated to this problem (e.g., [6, 8]. In particular, the method introduced in [8] also uses the iterated approach. However, it differs from the one used in ITEG in two main aspects: it uses a specific rule for selecting the subset of the best solution, and it replace the best cover with a new one only if the latter is strictly smaller than the former. Instead, in ITEG the subset of the best solution is selected randomly, and a best cover is replaced by a new one also if they have the same size, which helps escaping from local optima. This is substantiated by the experiments we have conducted.

Acknowledgements We would like to thank Mark Goldberg, Carlo Mannino, and Avishai Wool for useful observations related to the subject of this paper.

References

[1] E. Aarts and J.K. Lenstra (eds.). *Local Search in Combinatorial Optimization*. Wiley, England, 1997.

[2] D. Avis. A note on some computationally difficult set covering problems. *Mathematical Programming*, 8:138–145, 1980.

[3] E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. *Mathematical Programming*, 12:37–60, 1980.

[4] J.E. Beasley. An algorithm for set covering problem. *European Journal of Operational Research*, 31:85–93, 1987.

[5] J.E. Beasley. A lagrangian heuristic for set covering problems. *Naval Research Logistics*, 37:151–164, 1990.

[6] J.E. Beasley and P.C. Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94:392–404, 1996.

[7] J.E. Beasley and K. Jornsten. Enhancing an algorithm for the set covering problem. *European Journal of Operational Research*, 58:293–300, 1992.

[8] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. In W.H. Cunningham, T.S. McCormick, and M. Queyranne, editors, *Proc. of the Fifth IPCO Integer Programming and Combinatorial Optimization Conference*. Springer-Verlag, 1996.

[9] P. Erdös. On a combinatorial problem i. *Nordisk Mat. Tidskrift*, 11:5–10, 1963.

[10] P. Erdös. On some of my favourite problems in graph theory and block design. *Le Mathematische*, 45:61–74, 1990.

[11] U. Feige. A threshold of ln n for approximating set covering. In *Proc. of the 28-th Annual ACM Symposium on the Theory of Computing*, pages 314–318. ACM, 1996.

[12] D.R. Fulkerson, G.L. Nemhauser, and L.E. Trotter. Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triple systems. *Mathematical Programming Study*, 2:72–81, 1974.

[13] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.

[14] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. *European Journal of Operational Research*, 101:81–92, 1997.

[15] Carlo Mannino and Antonio Sassano. Solving hard set covering problems. *Operations Research Letters*, 18:1–5, 1995.

[16] K. Nonobe and T. Ibaraki. A Tabu search approach to the CSP (Constraint Satisfaction Problem) as a general problem solver. *European Journal of Operational Research*, 1998. To appear.

[17] M.A. Odijk and H. van Maaren. Improved solutions for the steiner triple covering problem. Technical report, TU Delft University, 113 1996.

[18] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, 1992.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 167–178

A Computational Study of Routing Algorithms for Realistic Transportation Networks

Riko Jacob, Madhav V. Marathe and Kai Nagel
Los Alamos National Laboratory,
P.O. Box 1663, MS M997, Los Alamos, NM 87545.
e-mail: {jacob, marathe, kai}@lanl.gov

ABSTRACT

We carry out an experimental analysis of a number of shortest path (routing) algorithms investigated in the context of the TRANSIMS (TRansportation ANalysis and SIMulation System) project. The main focus of the paper is to study how various heuristic as well as exact solutions and associated data structures affected the computational performance of the software developed especially for realistic transportation networks. For this purpose we have used Dallas Ft-Worth road network with high degree of resolution. The following general results are obtained.

1. We discuss and experimentally analyze various one-to-one shortest path algorithms. These include classical exact algorithms studied in the literature as well as heuristic solutions that are designed to take into account the geometric structure of the input instances.
2. We describe a number of extensions to the basic shortest path algorithm. These extensions were primarily motivated by practical problems arising in TRANSIMS and ITS (Intelligent Transportation Systems) related technologies. Extensions discussed include – (i) Time dependent networks, (ii) multi-modal networks, (iii) networks with public transportation and associated schedules.

Computational results are provided to empirically compare the efficiency of various algorithms. Our studies indicate that a modified Dijkstra's algorithm is computationally fast and an excellent candidate for use in various transportation planning applications as well as ITS related technologies.

1. Introduction

TRANSIMS is a multi-year project at the Los Alamos National Laboratory and is funded by the Department of Transportation and by the Environmental Protection Agency. The main purpose of TRANSIMS is to develop new methods for studying transportation planning questions. A typical example of a question that can be studied in this context would be to study the economic and social impact of building a new freeway in a large metropolitan area. Conceptually speaking, TRANSIMS decomposes the transport system into three time scales —- a long time scale associated with land use and demographic distribution as they pertain to characterization of travelers, an intermediate time scale associated with intermodal trip chain route planning (called the intermodal route planner) and a very short time scale associated with driving and other modal execution of trip plans in the transport system. At each time scale, a traveler activity is simulated. Due to lack of space, we refer the reader to [CS97, TR+95a] and the website http://www-transims.tsasa.lanl.gov/research_team/papers/ for more details about the TRANSIMS project.

The basic purpose of the intermodal route planner is to use the demographic and other relevant characteristics of a traveler to determine specific mode choices and travel routes for an individual traveler. The module uses the results of disaggregated household and commercial transportation activity module. As pointed out in [CS97, TR+95a], this module is intended to enhance and integrate the Modal Split and Trip Assignment phases of standard Urban Transportation Modeling System. The main goal of this paper is to describe the computational experiences in engineering various path finding algorithms specifically in the context of TRANSIMS. Most of the algorithms discussed here are not new; they have been discussed in the Operations Research and Computer Science community. Although extensive research has been done on theoretical and experimental evaluation of shortest path algorithms, most of the empirical research has focused on randomly generated networks, special classes of networks such as grids. In contrast, not much work has been done to study the computational behavior of shortest path and related routing algorithms on realistic traffic networks. The realistic networks differ with random networks as well as homogeneous (structured networks) in the following significant ways:

(i) Realistic networks typically have a very low average degree. In fact in our case the average degree of the network was around 2.6. Similar numbers have been reported in [ZN98]. In contrast random networks used in [Pa84] have in some cases average degree of up to 10.

(ii) Realistic networks are not very uniform. In fact, one typically sees one or two large clusters (downtown and neighboring areas) and then small clusters spread out throughout the entire area of interest.

(iii) For most empirical studies with random networks, the edge weights are chosen independently and uniformly at random from a given interval. In contrast, realistic networks typically have short links.

With the above reasons and specific application in mind, the main focus of this paper is to carry out experimental analysis of a number of shortest path algorithms on real transportation network and subject to practical constraints imposed by the overall system.

The rest of the report is organized as follows. Section 2 contains probelm statement and related discussion. In Section 3, we discuss the various algorithms evaluated in this paper. Section refsec:summary summarizes the results obtained. Section 5 describes our experimental setup. Section 6 describes the experimental results obtained. Section 7 contains a detailed discussion of our results. Finally, in Section 8 we give concluding remarks and directions for future research.

2. Problem specification and justification

The problems discussed above can be formally described as follows: let $G(V, E)$ be a (un)directed graph. Each edge $e \in E$ has one attribute — $w(e)$. $w(e)$ denotes the weight of the edge (or cost) e. Here, we assume that the weights are non-negative floating point numbers. Most of our positive results can in fact be extended to handle negative edge weights also (if there are no negative cycles).

Definition 2.1. One-to-One Shortest Path:
Given a directed weighted, graph G, a source destination pair (s, d) find a shortest (with respect to w) path p in G from s to d.

Note that our experiments are carried out for shortest path between a pair of nodes, as against to finding shortest path trees. Much of the literature on experimental analysis uses the latter measure to gauge the efficiency. Our choice for using one-to-one shortest path time as the measure is motivated by the following observations:

1. We wanted the route planer to work for roughly a million travelers. In highly detailed networks, most of these travelers have different starting points (for example, for Portland we

have 1.5 million travelers and 200 000 possible starting locations). Thus, for any given starting location, we could re-use the tree computation only for about ten other travelers.

2. We wanted our algorithms to be extensible to take additional elements into account. For example, each such traveler typically has a different starting time for his/her trip. Since we use our algorithms for time dependent networks (networks in which edge weights vary with time), the shortest path tree will be different for each traveler. Another example in this context is to find paths for travelers in network with multiple mode choices. In this context, we are given a directed labeled, weighted, graph G representing a transportation network with the labels on edges representing the various modal attributes (e.g. a label t might represent a rail line). The goal is typically to find shortest (simple) paths subject to certain labeling constraints on the set of feasible paths. In general, the criteria for path selection vary so much from traveler to traveler that it becomes doubtful that the additional overhead for the "re-use" of information will pay off.

3. The TRANSIMS framework allows us to use paths that are not necessarily optimal. This motivates investigation into the possible use of heuristic solutions for obtaining near optimal paths (e.g. the modified A^* algorithm). For most of these heuristics, the idea is to bias a more focused search towards the destination – thus naturally motivating the study of one-one shortest path algorithms.

4. Finally, the networks we anticipate to deal with contain more than 80 000 nodes and around 120 000 edges. For such networks storing shortest path trees amounts to huge memory overheads.

3. Choice of algorithms

Important objectives used to evaluate the performance of the algorithms include (i) time taken for computation on real networks, (ii) quality of solution obtained, (iii) ease of implementation and (iv) extensibility of the algorithm for solving other variants of the shortest path problem. A number interesting engineering questions were encountered in the process. We experimentally evaluated a number of variants of basic Dijkstra's algorithm. The basic algorithm was chosen due to the recommendations made in Cherkassky, Goldberg and Radzik [CGR96] and Zhan and Noon [ZN98]. The algorithms studied were:

- Dijkstra's algorithm with Binary Heaps [CGR96],
- A^* algorithm proposed in AI literature and analyzed by Sedgewick and Vitter [SV86],
- a modification of the A^* algorithm that we will describe below, and alluded to in [SV86].

We also considered a bidirectional version of Dijkstra's algorithm described in [Ma, LR89]. We briefly recall the A^* algorithm and the modification proposed. When the underlying network is Euclidean, it is possible to improve the average case performance of Dijkstra's algorithm. Typically, while solving problems on such graphs, the inherent geometric information is ignored by the classical path finding algorithms. The basic idea behind improving the performance of Dijkstra's algorithm is from [SV86, HNR68] and can be described as follows. In order to build a shortest path from s to t, we use the original distance estimate for the fringe vertex such as x, i.e. from s to x (as before) *plus* the Euclidean distance from x to t. Thus we use global information about the graph to guide our search for shortest path from s to t. The resulting algorithm typically runs much faster than Dijkstra's algorithm on typical graphs for the following intuitive reasons: (i) The shortest path tree grows in the direction of t and (ii) The search of the shortest path can be terminated as soon as t is added to the shortest path tree.

We can now modify this algorithm by giving an appropriate weight to the distance from x to t. By choosing an appropriate multiplicative factor, we can increase the contribution of the second component in calculating the label of a vertex. From a intuitive standpoint this corresponds to giving the destination a high potential, in effect biasing the search towards the destination. This modification will in general **not** yield shortest paths, nevertheless our experimental results suggest that the errors produced are typically quite small.

4. Summary of Results

We are now ready to summarize the main results and conclusions of this paper. As already stated the *main focus* of the paper is towards engineering well known shortest path algorithms in a practical setting. Another goal of this paper is also to provide reasons for and against certain implementations from a practical standpoint. We believe that our conclusions along with the earlier results in [ZN98, CGR96] provide practitioners a useful basis to select appropriate algorithms/implementations in the context of transportation networks. The general results/conclusions of this paper are summarized below.

1. We conclude that the simple Binary heap implementation of Dijkstra's algorithm is a good choice for finding optimal routes in real road transportation networks. Specifically, we found that certain types of data structure fine tuning did not significantly improve the performance of our implementation.

2. Our results suggest that heuristic solutions using the geometric structure of the graphs are attractive candidates for future research. Our experimental results motivated the formulation and implementation of an extremely fast heuristic extension of the basic A^* algorithm that seems to yield near optimal solutions.

3. We have extended this algorithm in two orthogonal and important directions; (i) time dependent networks and (ii) multi-modal networks. These extensions are significant from a practical standpoint since they are the most realistic representations of the underlying physical network. We perform suitable tests to calculate the slow down experienced as a result of these extensions.

4. Our study suggests that bidirectional variation of Dijkstra's algorithm is not suitable for transportation planning. Our conclusions are based on two factors: (i) the algorithm is not extensible to more general path problems and (ii) the running time of the algorithm is more than A^* algorithm.

5. Experimental Setup and Methodology

In this section we describe the computational results of our implementations. In order to anchor research in realistic problems, TRANSIMS uses example cases called *Case studies* (See [CS97] for complete details). This allows us to test the effectiveness of our algorithms on real life data. The case study just concluded focused on Dallas Fort-Worth (DFW) Metropolitan area and was done in conjunction with Municipal Planning Organization (MPO) (known as North Central Texas Council of Governments (NCTCOG)). We generated trips for the whole DFW area for a 24 hour period. The input for each traveler has the following format: (starting time, starting location, ending location).[1] There are 10.3 million trips over 24 hours. The number of nodes and links in the Dallas network is roughly 9863, 14750 respectively. The average degree of a node in the network was 2.6. We route all these trips through the so-called focused network. It has all freeway links, most major arterials,

[1] This is roughly correct, the reality is more complicated, [NB97, CS97].

etc. Inside this network, there is an area where *all* streets, including local streets, are contained in the data base. This is the study area. We initially routed all trips between 5am and 10am, but only the trips which did go through the study area were retained, resulting in approx. 300 000 trips. These 300 000 trips were re-planned over and over again in iteration with the micro-simulation(s). For more details, see, e.g., [NB97, CS97]. A 3% random sample of these trips were used for our computational experiments.

Preparing the network. The data received from DFW metro had a number of inadequacies from the point of view of performing the experimental analysis. These had to be corrected before carrying out the analysis. We mention a few important ones here. First, the network was found to have a number of disconnected components (small islands). We did not consider (o, d) pairs in different components. Second, a more serious problem from an algorithmic standpoint was the fact that for a number of links, the length was *less* than the actual Euclidean distance between the the two end points. In most cases, this was due to an artificial convention used by the DFW transportation planners (so-called centroid connectors always have length 10 m, whatever the Euclidean distance), but in some cases it pointed to data errors. In any case, this discrepancy disallows effective implementation of A^* type algorithms. For this reason we introduce the notion of the "normalized" network: For all "too short" links we set the reported length to be equal to the Euclidean distance.

We also carried out preliminary experimental analysis for the following network modifications that could be helpful in improving the efficiency of our algorithms. These include: (i) Removing nodes with degrees less than 3: (Includes collapsing paths and also leaf nodes) (ii) Modifying nodes of degree 3: (Replace it by a triangle)

Hardware and Software Support. The experiments were performed on a Sun UltraSparc CPU with 250 Mhz, running under Solaris 2.5. 2 gigabyte main memory were shared with 13 other CPUs; our own memory usage was always 150 MB or less. In general, we used the SUN Workshop CC compiler with optimization flag -fast. (We also performed an experiment on the influence of different optimization options without seeing significant differences.) The advantage of the multiprocessor machine was reproducibility of the results, as the operating system has no need to interrupt since requests by other processes were delegated to other CPUs.

Experimental Method We used the network described earlier. 10,000 arbitrary plans were picked from the case study. We used the timing mechanism provided by the operating system with granularity .01 seconds (1 tick). Experiments were performed only if the system load did not exceed the number of available processors, i.e. processors do not get shared. As long as this condition was not violated during the experiment, the running times were fairly consistent, usually within relative errors of 3%.

We used (a subset) of the following values measurable for a single or a specific number of computations to conclude the reported results

- (average) running time excluding i/o
- number of fringe/expanded nodes
- pictures of fringe/expanded nodes
- maximum heap size
- number and length of the path

Software Design We used the object oriented features as well as the templating mechanism of C++ to easily combine different implementations. We also used preprocessor directives and macros. Virtual methods were not used (even so it is tempting to create a purely virtual "network" base class) to avoid unnecessary function calls (by this enable inlining of functions). There are classes encapsulating the following elements of the computation:

- network (extensibility and different levels of detail lead to small, linear hierarchy)

- plans: (o, d) pairs and real paths, starting time
- heap
- labeling of the graph and using the heap
- storing the shortest path tree
- Dijkstra's algorithm

As expected, this approach leads to an overhead of function calls. Nevertheless, the compiler optimization can take care of this fairly well. (There is a factor of 2-3 difference in running time between debugging flag and full optimization.)

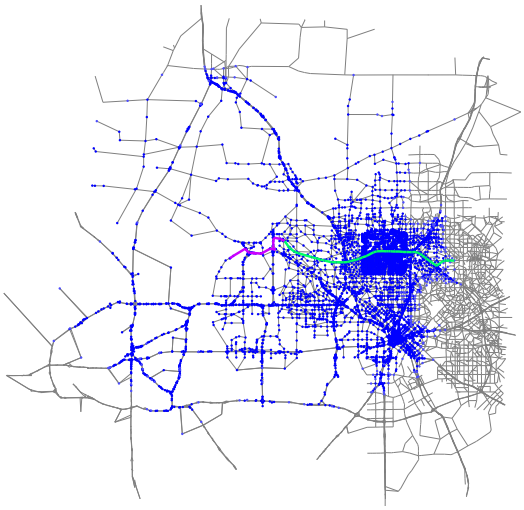
6. Experimental Results

Design Issues about Data Structures We begin with the design decisions regarding the data structures used.

A number of alternative data structures were considered to investigate if they results in substantial improvement in the running time of the algorithm. The alternatives tested included the following. (i) Arrays versus Heaps , (ii) Deferred Update, (iii) Hash Tables for Storing Graphs, (iv) Smart Label Reset (v) Heap variations, and (vi) struct of arrays vs. array of structs. We found, that indeed good programming practice, using common sense to avoid unnecessary computation and textbook knowledge on reasonable data structures are useful to get good running times. For the alternatives mentioned above, we did not find substantial improvement in the running time. More precisely, the differences we found were bigger than the unavoidable noise on a multi-user computing environment. Nevertheless, they were all below 10% relative difference.

Analysis of results. The plain Dijkstra, using static delays calculated from reported free flow speeds, produced roughly 100 plans per second. Figure 1 illustrates the improvement by the obatined by A^*. The numbers shown in the corner of the network snapshots tell an average (100 repetitions) running time for this particular O-D-pair, (destroying cache effects between subsequent runs) in system ticks. It also gives the number of nodes expanded and fringe nodes. Note the changed scale of the depictions due to the different nodes expanded. Overall we found that A^* is faster than basic Dijkstra's algorithm by roughly a factor of 2. Also, recall that for the original network Sedgewick and Vitter's heuristic was not applicable: it turned out that there exist some links that have reported length much smaller (factor 100) than the Euclidean distance of the endpoints. To be able to conduct any reasonable experiment, we modified ("normalized") the network as reported above: If necessary the reported length was changed to Euclidean distance, to ensure the correct inequality.

Modified A^* (Overdo Heuristic) Next consider the modified A^* algorithm – the heuristic is parameterized by the multiplicative factor used to weigh the Euclidean distance estimate to the desitnation. We call it the *overdo* parameter due to obvious reasons. As a result it is natural to discuss the time/quality trade-off of the heuristic as a function of the *overdo* parameter. Figure 2 summarizes the performance. In the figure the X-axis represents the overdo factor, being varied from 0 to 100 in steps of 1. The Y-axis is used for multiple attributes which we explain below. First, it is used to represent the average running time per plan. For this attribute, the scale is .02 seconds per unit. As depicted by the solid line, the average time taken without any overdo at all is 12.9 microseconds per plan. This represents the base measurement (without taking the geometric information into account). Next, for overdo value of 10 and 99 the running times are respectively 2.53 and .308 microseconds. On the other hand, the quality of the solution produced by the heuristic detiorates as the overdo factor is increased. We used two quantities to measure the error — (i) the maximum relative error incurred over 10000 plans and (ii) the number of plans with errors more than a given threshold error. The maximum relative error (plot marked with *) ranges from 0 for



ticks 2.40, #exp 6179, #fr 233

ticks 0.64, #exp 1446, #fr 316

Figure 1: Figure illustrating the number of expanded nodes while running (i) Dijkstra (ii) A^* algorithms. As the figures clearly show the A^* heuristic clearly is much more efficient in terms of the nodes it visits. In both the graphs, the path is outlined as a dark line. The fringe nodes and the expanded nodes are marked as dark spots. The underlying network is shown in light grey.

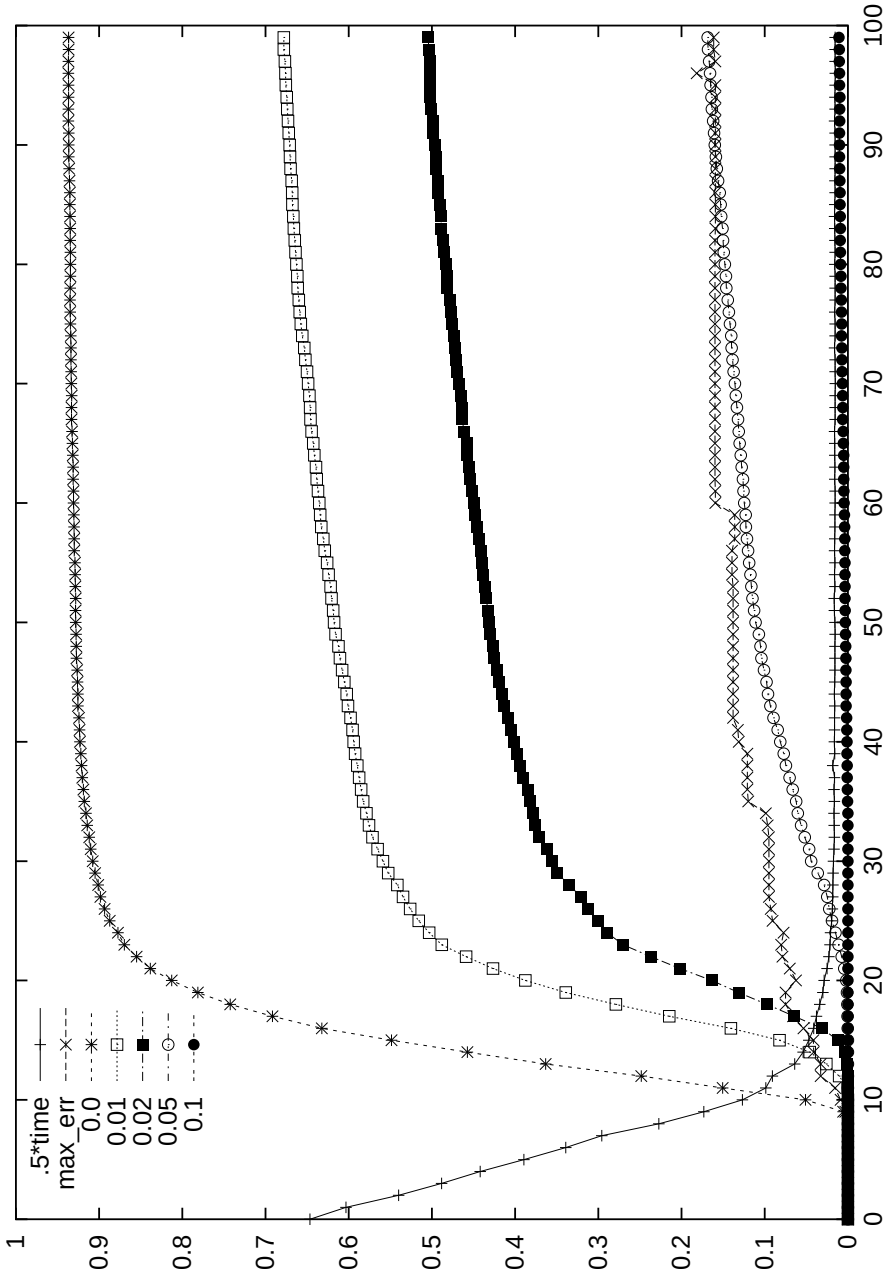


Figure 2: Figure illutrating the trade-off between the running time and quality of paths as a function of the overdo-parameter.

overdo factor 0 to 16% for overdo value 99. For the other error measure, we plot one curve for each threshold error of 0%, 1%, 2%, 5%, 10%. The following conclusions can be drawn from our results.

1. The running times improve significantly as the overdo factor is increased. Specifically the improvements are a factor 5 for overdo parameter 10 and almost a factor 40 for overdo parameter 99.

2. In contrast, the quality of solution worsens much more slowly. Specifically, the maximum error is no worse than 16% for the maximum overdo factor. Moreover, although the number of erroneous plans is quite high (almost all plans are erroneous for overdo factor of 99), most of them have small relative errors. To illustrate this, note that only around 15% of them have relative error of 5% or more.

3. The experiments and the graphs suggest an "optimal" value of overdo factor for which the running time is significantly improved while the solution quality is not too bad. Thus our experiments are a step in trying to find an empirical time/performance trade-off as a function of the overdo parameter.

4. We also found that the near-optimal paths produced were visually acceptable and represented a feasible alternative route guiding mechanism. This method finds alternative paths that are quite different than ones found by the k-shortest path algorithms and seem more natural. Intuitively, the k-shortest path algorithms, find paths very similar to the overall shortest path, except for a few local changes.

7. Discussion of Results

First, we note that the running times for the plain Dijkstra are reasonable as well as sufficient in the context of the TRANSIMS project. Quantitatively, this means the following: TRANSIMS is run in iterations between the micro-simulation, and the planner modules, of which the shortest path finding routine is one part. We have recently begun research for the next case study project for TRANSIMS. This case study is going to be done in Portland, Oregon and was chosen to demonstrate the validate our ideas for multi-modal time dependent networks with public transportation following a scheduled movement. Our initial study suggests that we now take .5 sec/trip as opposed to .01 sec/trip in the Dallas Ft-Worth case. All these extensions are important from the standpoint of finding algorithms for realistic transportation routing problems. We comment on this in some detail below. Multi-modal networks are an integral part of most MPO's. Finding optimal (or near-optimal) routes in this environment therefore constitutes a real problem. In the past, solutions for routing in such networks was handled in an adhoc fashion. In [BJM98], we have proposed models and corresponding algorithms to solve such problems. Next consider another important extension — namely to time dependent networks. In this case the edge length is assumed to be a function of time. We make an important modeling assumption, namely it does not pay a person to wait. This need not be true in general but is adequate for most purposes. This implies that the edge length function is monotonically non-increasing. Time dependent networks can also be used to models public transportation systems with fixed schedules. By using an appropriate extension of the basic Dijkstra's algorithm, one can calculate optimal paths in such networks. Our preliminary results on these topics in the context of TRANSIMS can be found in [JM98]. The Portland network we are intending to use has about 120 000 links and about 80 000 nodes. Simulating 24 hours of traffic on this network will take about 24 hours computing time on our 14 CPU machine. There will be about 1.5 million trips on this network. Routing all these trips should take $1.5 \cdot 10^6$ trips $\cdot 0.5$ sec/trip $\approx$ 9 days on a single CPU and thus less than 1 day on our 14 CPU machine. Since re-routing typically concerns only 10% of the population, we would need less than 3 hours of computing time for the re-routing part of one iteration, still significantly less than the micro-simulation needs.

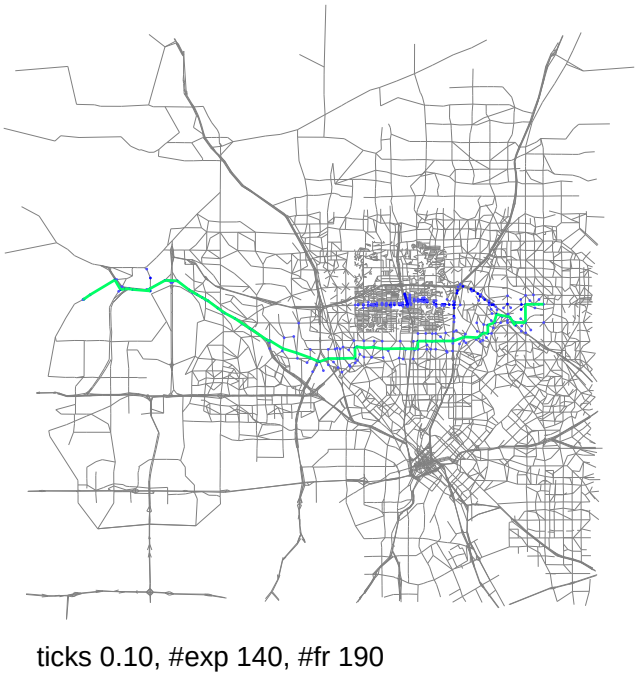


Figure 3: Figure illutrating two instances of Dijkstra's algorithms with a very high overdo parameter start at origin and destination respectively. One of them really creates the shown path, the beginning of the other path is visible as a "cloud" of expanded nodes

Our results and the constraints placed by the functionality requirement of the overall system imply that bidirectional version of Dijkstra's algorithm is not a viable alternative. Two reasons for this are: (i) The algorithm can not be extended in a direct way to path problems in a multi-modal and time dependent networks, and (ii) the running times of A^* is better than the bidirectional variant; the modified A^* is much more faster.

8. Conclusions

The computational results presented in the previous sections demonstrate that Dijkstra's algorithm for finding shortest paths is a viable candidate for compute route plans in a route planning stage of a TRANSIMS like system. In fact, even more interestingly, the results demonstrate that the algorithm that has optimized well compares well (or even sometimes better) than several heuristics proposed in the literature. Thus such an algorithm should be considered even for ITS type projects in which we need to find routes by an on-board vehicle navigation systems.

In the context of the TRANSIMS project, we are faced with the problem of routing many millions of trips in iteration with a micro-simulation. Most trips have entirely different characteristics, such as different starting locations, different starting times, and different preferences towards mode choice. This leads to the consideration of one-to-one shortest path algorithms, as opposed to algorithms that construct the complete shortest-path tree from a given starting (or destination) point. As is well known, the worst-case complexity of one-to-one shortest path algorithms is the same as of one-to-all shortest path algorithms. Yet, in terms of our practical problem, this is not applicable. First, a one-to-one algorithm can stop as soon as the destination is reached, saving computer time especially when trips are short (which often is the case in our setting). Second, since our networks are roughly Euclidean, one can use this fact for heuristics that reduce computation time even more.

One heuristic, the Sedgewick-Vitter or A* algorithm (denoted SV/A*), generates results that are provably optimal, but is a heuristic in the sense that the worst-case complexity does not get any better although practical computing times decrease. One can extend the approach of SV/A* towards a "true" heuristic where routes are no longer optimal but computation time goes down even more. The above approaches were evaluated in the context of the TRANSIMS Dallas-Fort Worth case study. The underlying road network was a "focussed network", with all streets including the local ones in a 25 square mile study area, with increasing number of streets left out when going away from the study area. For that case, SV/A* turns out to be about a factor of two faster than regular Dijkstra; the second heuristic could save, for example, another factor of 5 while generating results within 1% of the optimal solution.

Making the algorithms time-dependent in all cases slowed down the computation by a factor of at most two. Since we are using a one-to-one approach, adding extensions that for example include personal preferences (e.g. mode choice) are straightforward; preliminary tests let us expect slow-downs of not more than a factor 30. This significant slowdown was caused by a number of factors including: (i) increase in the network size by a factor of 4 caused by node/edge splitting and adding public tranportation, (ii) complicated time dependency functions representing scheduled buses and (iii) different type of delays inducing a qualtitatively different exploration of the network by the algorithm. Extrapolations of the results for the Portland case study show that, even with this slowdown the route planning part of TRANSIMS still uses significantly less computing time than the micro-simulation.

Finally, we note that under certain circumstances the one-to-one approach chosen in this paper may also be useful for ITS applications. This would be the case when customers whould require customized route suggestions, so that re-using a shortest path tree from another calculation may no longer be possible.

Acknowledgments: Research supported by the Department of Energy under Contract W-7405-ENG-36. We would like to thank the members of the TRANSIMS team in particular, Doug Anson, Chris Barrett, Richard Beckman, Roger Frye, Terence Kelly, Marcus Rickert, Myron Stein and Patrice Simon for providing the software infrastructure, pointers to related literature and numerous discussions on topics related to the subject. The second author wishes to thank Myron Stein for long discussions on related topics and for his earlier work that motivated this paper. We also thank Joseph Cheriyan, S.S. Ravi, Prabhakar Ragde, R. Ravi and Aravind Srinivasan for constructive comments and pointers to related literature.

References

[AMO93] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading MA., 1974.

[BJM98] C. Barrett, R. Jacob, M. Marathe, *Formal Language Constrained Path Problems* to be presented at the Scandanavian Workshop on Algorithmic Theory, (SWAT '98), Stockholm, Sweden, July 1998. Technical Report, Los Alamos National Laboratory, LA-UR 98-1739.

[TR+95a] C. Barrett, K. Birkbigler, L. Smith, V. Loose, R. Beckman, J. Davis, D. Roberts and M. Williams, *An Operational Description of TRANSIMS*, Technical Report, LA-UR-95-2393, Los Alamos National Laboratory, 1995.

[CS97] R. Beckman et. al. *TRANSIMS-Release 1.0 – The Dallas Fort Worth Case Study*, LA-UR-97-4502

[CGR96] B. Cherkassky, A. Goldberg and T. Radzik, *Shortest Path algorithms: Theory and Experimental Evaluation*, Mathematical Programming, Vol. 73, 1996, pp. 129–174.

[GGK84] F. Glover, R. Glover and D. Klingman, *Computational Study of am Improved Shortest Path Algorithm*, Networks, Vol. 14, 1985, pp. 65–73.

[EL82] R. Elliott and M. Lesk, "Route Finding in Street Maps by Computers and People," *Proceedings of the AAAI-82 National Conference on Artificial Intelligence,* Pittsburg, PA, August 1982, pp. 258-261.

[HNR68] P. Hart, N. Nilsson, and B. Raphel, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on System Science and Cybernetics,* (4), 2, July 1968, pp. 100-107.

[HM95] Highway Research Board, *Highway Capacity Manual*, Special Report 209, National Research Council, Washington, D.C. 1994.

[JM98] R. Jacob et. al. *Models and Algorithms for Routing Multi-Modal Trips in Time Dependent Networks,* in preparation, June 1998.

[LR89] M. Luby and P Ragde, "A Bidirectional Shortest Path Algoroithm with Good Average Case Behaviour," *Algorithmica*, 1989, Vol. 4, pp. 551-567.

[Ha92] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36-42 (1992).

[Ma] Y. Ma, "A Shortest Path Algorithm with Expected Running time $O(\sqrt{V}\log V)$," Master's Thesis, University of California, Berkeley.

[MCN91] J.F. Mondou, T.G. Crainic and S. Nguyen, *Shortest Path Algorithms: A Computational Study with C Programming Language*, Computers and Operations Research, Vol. 18, 1991, pp. 767–786.

[NB97] K. Nagel and C. Barrett, *Using Microsimulation Feedback for trip Adaptation for Realistic Traffic in Dallas,* International Journal of Modern Physics C, Vol. 8, No. 3, 1997, pp. 505-525.

[NB98] K. Nagel, *Experiences with Iterated Traffic Microsimulations in Dallas,* in D.E. Wolf and M. Schreckenberg, eds. Traffic and Granular flow II Springer Verlag 1998. Technical Report, Los Alamos National Laboratory, LA-UR 97-4776.

[Pa74] U. Pape, *Implementation and Efficiency of Moore Algorithm for the Shortest Root Problem*, Mathematical Programming, Vol. 7, 1974, pp. 212–222.

[Pa84] S. Pallottino, *Shortest Path Algorithms: Complexity, Interrelations and New Propositions*, Networks, Vol. 14, 1984, pp. 257–267.

[Po71] I. Pohl, "Bidirectional Searching," *Machine Intelligence*, No. 6, 1971, pp. 127-140.

[SV86] R. Sedgewick and J. Vitter "Shortest Paths in Euclidean Graphs," *Algorithmica*, 1986, Vol. 1, No. 1, pp. 31-48.

[SI+97] T. Shibuya, T. Ikeda, H. Imai, S. Nishimura, H. Shimoura and K. Tenmoku, "Finding Realistic Detour by AI Search Techniques," Transportation Research Board Meeting, Washington D.C. 1997.

[ZN98] F. B. Zhan and C. Noon, *Shrotest Path Algorithms: An Evaluation using Real Road Networks* Transportation Science, Vol. 32, No. 1, (1998), pp. 65–73.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 179–192

Hybrid Tree Reconstruction Methods

Daniel Huson
Program in Applied and Computational Mathematics
Princeton University, Princeton NJ 08544-1000
e-mail: huson@math.princeton.edu

Scott Nettles
Department of Computer and Information Science
University of Pennsylvania, Philadelphia PA 19104
e-mail: nettles@central.cis.upenn.edu

Kenneth Rice
Bioinformatics Department
SmithKline Beecham, King of Prussia PA, 19406
e-mail: ken_rice@sbphrd.com

Tandy Warnow
Department of Computer and Information Science
University of Pennsylvania, Philadelphia PA 19104
e-mail: tandy@central.cis.upenn.edu

and

Shibu Yooseph
DIMACS, Rutgers University, Piscataway, NJ 08854
e-mail: yooseph@saul.cis.upenn.edu

ABSTRACT

A major computational problem in Biology is the reconstruction of evolutionary trees for species sets, and accuracy is measured by comparing the *topologies* of the reconstructed tree and the model tree. One of the major debates in the field is whether large evolutionary trees can be even approximately accurately reconstructed from biomolecular sequences of realistically bounded lengths (up to about 2000 nucleotides) using standard techniques (polynomial time distance methods, and heuristics for NP-hard optimization problems). Using both analytical and experimental techniques, we show that on large trees, the two most popular methods in systematic biology, neighbor-joining and maximum parsimony heuristics, as well as two promising methods introduced by theoretical computer scientists, are all likely to have significant errors in the topology reconstruction of the model tree. We also present a new general technique for combining outputs of different methods (thus producing *hybrid* methods), and show experimentally how one such hybrid method has better performance than its constituent parts.

1. Introduction

Evolution of biomolecular sequences is modeled as a Markov process operating on a rooted binary tree. A biomolecular sequence at the root of the tree "evolves down" the tree, each edge of the tree introducing point mutations, thereby generating sequences at the leaves of the tree, each of the same length as the root sequence. The *phylogenetic tree reconstruction problem* is to take the sequences that occur at the leaves of the tree, and infer, as accurately as possible, the tree that generated the

sequences. In this paper, we focus on the *topology estimation problem*, which is the major objective of systematic biologists (that is, biologists whose research is the evolutionary history of different species sets).

The importance of accurate (or at least *boundedly inaccurate*) reconstruction of evolutionary trees to systematic biologists is reflected in the heated debates in the field about the relative accuracies of different phylogenetic reconstruction methods. This has been explored in the systematic biology literature through experimental performance studies which simulate biomolecular (DNA, RNA, or aminoacid) sequence evolution on model trees (see, for example, [12, 24, 18, 27, 25, 15, 16, 13, 28]). One of the most important limitations on performance turns out to be that real biomolecular sequences are not particularly long; those used for phylogenetic tree reconstruction purposes are typically bounded by 2000 nucleotides, often by much smaller numbers, and sequence lengths of 5000 nucleotides are generally considered to be *unusually long* [19]. The majority of these experimental studies have focused on small trees containing at most 20 leaves (many studies have addressed only four-leaf trees [14, 16, 15]), and very few have examined performance on large trees (having more than 50 taxa).

This paper has two major contributions. First, we provide an experimental performance study of four phylogenetic tree reconstruction methods: the two major phylogenetic tree reconstruction methods (a heuristic used to "solve" the NP-hard maximum parsimony problem, and the polynomial time "neighbor-joining" method), and two polynomial time methods introduced by the theoretical computer science community (the "Single Pivot" algorithm of Agarwala *et al.* [1] and the Buneman Tree [4, 3] method). We show, using both experimental and analytical techniques, that the polynomial time distance methods have poor accuracy when the model tree has high **divergence** (that is, when the model tree contains a path on which a random site changes many times). This suggests that datasets that have high divergence may need to be analyzed using computationally expensive techniques (such as maximum parsimony heuristics) rather than polynomial time techniques, in order for even *approximately* accurate reconstructions to be obtained. This observation, although preliminary, is new to the systematic biology literature; no other experimental study has explicitly studied the effect of varying divergence on degrees of accuracy of reconstructions of large trees.

The other contribution of the paper is the introduction of *Hybrid Tree Reconstruction Methods*. Our experimental results indicated that methods differed according to the *types* of topological errors they made. This discovery led us to propose a new approach to phylogeny reconstruction, in which outputs of different methods are combined (in a *hybrid* approach), so as to obtain the best of each of the methods. We experimentally explore the performance of a particular hybrid method, and show it obtains better results than its constituent parts (parsimony, neighbor-joining, and the Buneman Tree) over a significant portion of the "hard" part of the parameter space we explore.

2. Basics

Poisson Processes on Trees: Our performance study is based upon simulations of sets of DNA sequences under the Jukes-Cantor model of evolution [17] on different model trees.

Definition 1. *Let T be a fixed rooted tree with leaves labelled $1 \ldots n$. The* **Jukes-Cantor** *model of evolution describes how a site (position in a sequence of nucleotides) evolves down the tree T, and assumes that the sites evolve identically and independently (iid). The state at the root is drawn from a distribution (typically uniform), and each edge $e \in E(T)$ is associated with a* mutation probability *$p(e)$, which is the probability that a given site will have different states at the endpoints of the edge. Given that a change occurs on an edge, then the probability of obtaining state j at the child of the edge, given that the state of the parent of the edge is i, is given by the ij^{th} entry of a matrix M.*

In our study, we will assume that the substitution matrix for each edge is homogeneous, and the underlying tree is binary. Let λ_{ij} denote the expected number of mutations on the path P_{ij} between

leaves i and j in T, for a random site. We will call λ_{ij} the **expected evolutionary distance**, or **true distance** between i and j.

Definition 2. *We define the* **divergence** *of the tree T to be $\lambda_{max} = max_{ij}\{\lambda_{ij}\}$.*

The divergence can be unboundedly large, even if the number of leaves is held constant, since sites can change many times on an edge, even though only one change can possibly be observed for any site, between any pair of leaves.

Because the matrix λ is additive (it fits an edge-weighted tree exactly), given λ the tree T can be constructed in polynomial time using a number of different distance-based methods [32, 4]. The basic technique used in distance methods is as follows: *First*, an approximation d to the matrix λ is computed; *then*, d is mapped, using some distance method M, to a (nearby) additive matrix $M(d) = D$. If D and λ define the same unrooted leaf-labelled tree, then, the method is said to be **accurate**, even if they assign different weights to the edges of the tree.

While complete topological accuracy is the objective, partial accuracy is the rule. Systematic biologists quantify degrees of accuracy according to the shared *bipartitions* induced on the leaves of the model tree and the reconstructed tree.

Character encodings of trees: Given a tree T leaf-labelled by S and given an internal edge $e \in E(T)$, the *bipartition* induced by e is the bipartition on the leaves of T obtained by deleting e from T. We denote this bipartition by Π_e. It is clear that every S-labelled tree T is defined uniquely by the set $C(T) = \{\Pi_e : e \in E_{int}(T)\}$, where $E_{int}(T)$ denotes the internal edges of T (i.e. those edges not incident to leaves). This is called the **character encoding** of T. The character encoding of trees is useful in many ways. For example, we say that a tree T **refines** tree T' if T' can be obtained from T by contracting certain edges in T (and similarly, T' is a **contraction** of T). It follows that T refines T' if and only if $C(T') \subseteq C(T)$. We also say that a set C_0 of bipartitions of a set S is **compatible** if and only if there is a tree T such that $C(T) = C_0$.

Comparing trees: If T is the model tree and T' is an approximation to T (obtained using some method, perhaps), then the *errors* in the reconstructed tree T' can be classified into two types: **false positives**: edges $e \in E(T')$ such that $\Pi_e \notin C(T)$. **false negatives**: edges $e \in E(T)$ such that $\Pi_e \notin C(T')$. The **false positive rate** is then the number of false positives, divided by $n-3$, the number of internal edges in a binary tree on n leaves (evolutionary trees are typically presumed to be binary, even if attempts to reconstruct them have a hard time obtaining all edges). Similarly, the **false negative rate** is the number of false negatives divided by $n-3$.

Tree reconstruction methods: The two perhaps most frequently used methods in systematic biology are **heuristic parsimony** and **neighbor-joining**. Given a set S of sequences, and given a tree T leaf-labelled by $S \subseteq \mathbb{Z}^k$ and internally labelled by vectors in $\mathbb{Z}^k$, the parsimony cost of T is the sum of the Hamming distance between the endpoints of the edge (where the Hamming distance between x and y $H(x,y) = |\{i : x_i \neq y_i\}|$). Finding the most parsimonious tree (i.e. the tree of minimum cost) is the "maximum parsimony" problem, and is NP-hard [11]. Heuristics used for maximum parsimony are based upon hill-climbing through tree space, and return the *strict consensus* of all the best trees that are found during the search. (The strict consensus is the (unique) tree T_{strict} defined by $C(T_{strict}) = \cap_i C(T_i)$, where the best trees found are $T_1, T_2, \ldots, T_p$.) Heuristic parsimony is a computationally intensive method which has, on at least one real 500 taxon data set [21], taken several years of CPU time without finding an optimal tree.

Distance methods are very popular as well, and have the advantage over maximum parsimony of being very fast (almost all are $O(n^3)$ or $O(n^4)$, where n is the number of taxa). Furthermore, provided that properly corrected distances are given as input, most distance methods are provably **statistically consistent** throughout the parameter space of *i.i.d.* site evolution, meaning that they are guaranteed to recover the model tree topology (with arbitrarily high probability) given long enough sequences (see [31, 6] for the conditions that suffice to guarantee statistical consistency). By contrast, maximum parsimony has no such guarantee (it is not consistent on all trees under the general Markov model, and will, in fact, converge to the wrong tree given infinite length sequences

under some model conditions [8]). The most favored distance method is probably **neighbor-joining** [23], a simple polynomial time agglomerative clustering heuristic which performs surprisingly well, especially considering that it does not claim to solve or approximate any known optimization problem. The **Buneman Tree** method [4] and **Single Pivot** algorithm [1] are two other polynomial time distance methods which have been introduced by the theoretical computer science community, and which do solve or approximate optimization problems related to tree reconstruction. These three distance-based methods are each statistically consistent for the Jukes-Cantor model of site evolution.

An analysis of the sequence length needed for a completely accurate reconstruction (with high probability) of the topology of a tree under the general Markov model was given in [5]. In this section, we extend the analysis in [5] to derive upper bounds on the error rates of these methods.

We will say that an edge $e \in E(T)$ is reconstructed by a method M given input d if the tree $T' = M(d)$ contains an edge inducing the same bipartition as induced by e. Let $E_{int}(T)$ denote the internal edges of the tree T. We give an analysis of the sequence length that suffices for recover all long enough edges, under the Jukes-Cantor model. (The proof of the following theorem uses techniques similar to those for the corresponding theorem in [5], and is omitted.)

Theorem 1. *Let T be a Jukes-Cantor model tree, let λ_{ij} be the expected number of mutations in the path P_{ij} of a random site, and let $\lambda^* = max\{\lambda_{ij}\}$. Let $\epsilon > 0$ be given. If we use either AddTree (a variant of neighbor-joining), the Single Pivot algorithm, or the Buneman Tree method, then with probability at least $1 - \epsilon$ we will reconstruct all edges $e \in E_{int}(T)$ such that $p(e) \geq f$ if the sequence length exceeds*

$$c \log n e^{O(\lambda^*)}$$

where c is a constant that depends upon f, the method, and upon ϵ.

This theorem places an upper bound on the sequence length that suffices for these methods to reconstruct (with high probability) all edges above a given threshold of length, but does not imply correspondingly bad performance if shorter sequences are used. However, the theorem is an *upper bound*, and although discouragingly high, may be pessimistic-ally large (in other words, actual performance may be better than this upper bound would suggest). In the following section, we explore the performance of these methods by simulating sequence evolution on trees with varying degrees of divergence. Note also that the theorem does not imply any bound for neighbor-joining; the convergence rate of neighbor-joining is unfortunately still an open problem (see [2] for the result on AddTree, and a discussion about neighbor-joining.).

3. Performance Study

3.1. Methods and procedures:

Model trees and simulations: Our two basic model trees are both subtrees of the 500 *rbc*L tree from [21]; one has 35 taxa and the other has 93 taxa. The two trees were reconstructed using parsimony analysis [21], and the substitution rates on the edges in each tree were set to be the proportion of change on that edge, on the basis of a most parsimonious assignment of sequences to the internal nodes. We then scaled the rates both up and down to explore the effect of how different rates at different sites would affect the performance of these different methods. We used 11 different settings for the maximum mutation probability on any edge in each model tree (denoted in our study by $p(e)$), ranging from .005 to .64, and maintained the ratios between different edges. This technique also allowed us to generate sets of sequences with varying degrees of divergence (i.e. the maximum distance between pairs of leaves), while still having small enough data sets to do a significant number of experiments. In the high end of the range (i.e. when $p(e) = .64$, the maximum we tested), there is significant divergence and "homoplasy" (i.e. many sites change many times). The average number of times a site changes on the 35-taxon tree at that setting is 12, and the

average number of times a site changes on the 93-taxon tree at that setting is 30. Since there are almost three times as many edges in the 93-taxon tree as in the 35-taxon tree, these trees have approximately the same amount of homoplasy for their sizes. However, the 35-taxon tree has not quite as wide a range of mutation probabilities: the ratio between the "longest" and the "shortest" branches is about 17, while on the 93-taxon tree the ratio is about 30 (these ratios are calculated on the basis of the mutation probabilities, not upon corrected distances). Thus, for every maximum $p(e)$ setting, the 93 taxon tree contains *shorter* edges than the 35 taxon tree.

We varied the length of the sequences we generated, using 12 different lengths in the 200 through 3200 range, and including longer sequences (up to 12,800) on the hardest model trees. For each combination of tree, sequence length, and maximum mutation probability $p(e)$, we generated 100 sets of sequences, using different seeds for the random number generator. Each of these datasets was then given as input to each of the tree reconstruction methods we studied.

Tree reconstruction methods The tree reconstruction methods we compared in this study were: neighbor-joining [23], the Buneman Tree [4], heuristic search parsimony combined with the strict consensus (which we call the **HS-strict** tree), and the Single Pivot algorithm. The implementations of neighbor-joining was obtained from *Phylip* [9], and the HS-strict method was obtained from *PAUP* [29]; both of these are standard phylogenetic software packages. We implemented the Buneman Tree method and the Single Pivot algorithm ourselves, selecting a random pivot for the Single Pivot algorithm. While the parsimony search is generally improved significantly by allowing many random starting points for the heuristic searches, for the sake of the experimental study, we only permitted one random starting point for each experiment. In practice, real data sets are often analyzed with thousands of different random sequence addition orders, so as to explore more of the tree space; this restriction may result in poorer performance predictions for parsimony than might be achievable.

Experimental procedure: We used *ecat* [20] to simulate sequence evolution on each of the model trees we used. We then computed Jukes-Cantor distances [17] $d_{ij} = -3/4log(1-4/3H_{ij})$, where H_{ij} is the Hamming distance. For those pairs i, j in which $H_{ij} \geq 3/4$, we set d_{ij} to a very large value, significantly exceeding the other values in the matrix, so as to permit each of the distance methods to reconstruct trees instead of simply failing to return any tree at all (this is a standard adjustment to Jukes-Cantor distances used in these cases). We gave the same distance matrices to each of the three distance methods, and the original set of sequences to the heuristic for maximum parsimony analysis. We then compared the outputs of each method to the model tree, and computed false positive and false negative rates. Because the lengths of sequences in realistic datasets are typically bounded by 2000 or perhaps 3000 nucleotides (and are often only a few hundred), we will focus on performance results on short sequences, although we will report results for longer sequences as well.

3.2. Experimental results

Here we report on the results of our experiments on the four basic methods we studied. At the low end of this setting, except under exceedingly high sequence lengths (not explored here), many edges will have no changes on them – and hence be impossible to reconstruct for any method (except by guesswork). Thus, we will expect to see (and will see, in fact) high false negative rates for all methods when the mutation rates are very low. The important distinctions in performance will be obtained for mutation rates in the moderate to high range.

Each point in each figure represents the mean of 100 samples. This number of samples gives us a 90% confidence of having an *absolute error* of no more than 3%, across all of our parameter space. We provide error bars in the first figure, indicating the 25th and 75th percentiles. These representations of the distribution are omitted from later figures as they make it difficult at times to see the pattern and are not particularly informative.

In the figures below, **HS-strict** (or simply **HS**) refers to Heuristic Search Parsimony combined with the Strict Consensus, and **NJ** refers to the neighbor-joining method.

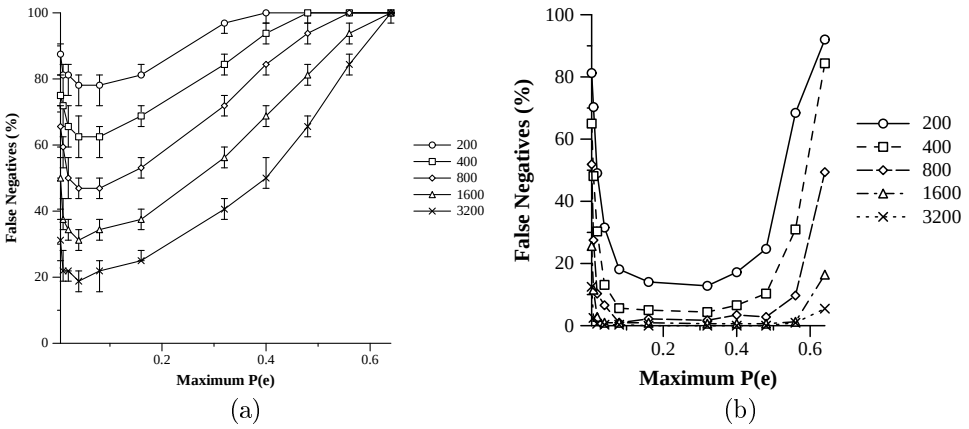


Figure 1: (a) Max p(e) vs. FN, by sequence length, for Buneman on the 35-taxon tree, (b) Max p(e) vs. FN/FP, by sequence length, for NJ on the 35-taxon tree

35-taxon tree In Figures 1-4, we examine the false negative rates for the four methods on the 35-taxon tree, as we vary the maximum divergence (by modifying the setting for $p(e)$, the largest mutation rate on any edge in the tree) and the sequence length.

The false negative curves for the four methods have a characteristic shape – initial decrease in the false negative rate as edges start to have enough "hits" to be reconstructible, followed by a period in which the method performs well, and then an increase in the false negative rate. It is worth noting that shapes of the three curves for the distance methods are consistent with Theorem 1, and that while the curve for parsimony has a comparable shape, we have no analytical result which would predict this. A comparison between the four methods in terms of the false negative rate for the high divergence settings is also worth noting. The Buneman Tree has the worst false negative rate of the four methods at all sequence lengths, and the Single Pivot algorithm fits between the Buneman and neighbor-joining method at all sequence lengths. The comparison between parsimony and neighbor-joining is more interesting. For short sequences (that is, for sequences of length 200, 400, and 800) and high divergence, parsimony has a lower false negative rate than neighbor-joining, but the relative performance shifts for longer sequences.

We now compare the false positive rates for these four methods. See Figures 2 and 3 for the neighbor joining and Single Pivot algorithms, whose false positive rates and false negative rates are essentially identical (since both almost always produce binary trees), and Figures 3(a) and 3(b) for heuristic parsimony and the Buneman Tree method. The Buneman Tree method has the best false positive rate of all four methods on this tree, obtaining actually no false positives at all sequence lengths; thus, the Buneman Tree method produces a *contraction* of the model tree. Following close behind the Buneman Tree is Heuristic Parsimony, which obtains close to 0 false positives for a wide range of the different $p(e)$ settings at most sequence lengths. The other two methods, neighbor-joining and Single Pivot, have generally higher false positive rates, and of these two, Single Pivot's is worse (just as the Single Pivot false negative rate is higher than that of neighbor-joining).

Comparing more carefully between the neighbor-joining method and heuristic parsimony's false positive rates is interesting; for short sequences (below 1600), parsimony has a lower false positive rate than neighbor-joining, but the relative performance changes at longer sequences.

93-taxon tree Figures 4-5 give the same types of results as Figures 1-3, but on the 93-taxon tree. We have omitted the figures for the Buneman Tree performance, because they are comparable to the performance on the 35 taxon tree: as before, the false positive rate is almost always zero, although

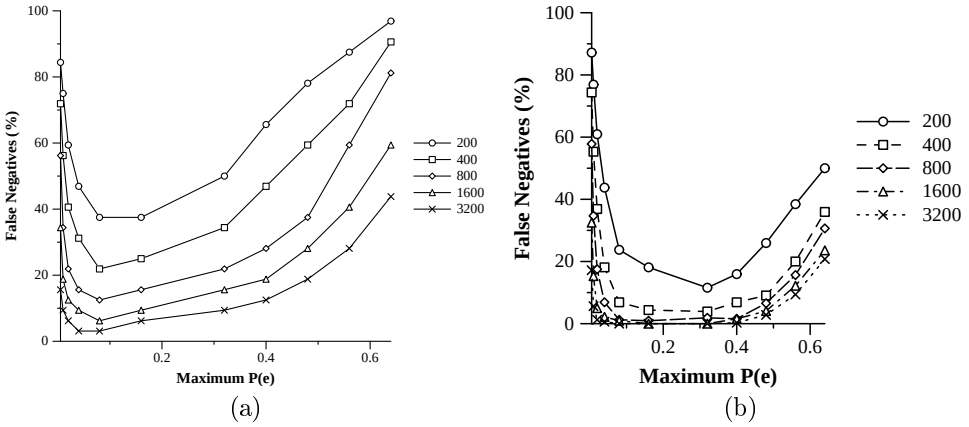


Figure 2: (a) Max p(e) vs. FN/FP, by sequence length, for Single Pivot on the 35-taxon tree, (b) Max p(e) vs. FN, by sequence length, for HS-strict on the 35-taxon tree

the false negative rate climbs even faster to 100%. We show figures for the other three methods.

The false positive rates are as before: low for heuristic parsimony and exceptionally low for the Buneman Tree method, and not as uniformly low for the neighbor-joining and Single Pivot algorithms. With respect to the false negative rates, we see the same "bowl-like" pattern we saw before: an initial decline, followed by an increase in false negative rates as the divergence of the tree is turned up. The Buneman Tree method (not shown) does very badly with respect to false negative rates, and Single Pivot does almost as poorly; neighbor-joining and parsimony do much better than the other two methods.

Perhaps the most striking aspect of this study is that the relative performance between neighbor-joining and parsimony is quite different here than on the 35-taxon tree. Whereas on the 35-taxon tree there were portions of the parameter space where each method outperformed the other, on this tree parsimony is significantly better throughout the parameter space. In fact, examining performance on "short" sequences (consisting of at most 800 nucleotides), parsimony's average false negative rate is less than 30% that of neighbor-joining! The major reason for this distinction in performance is probably that the 93-taxon tree contains many very short edges (moreover, the "longest" branch is 30 times as long as the shortest – calculations based upon $p(e)$ values, not corrected distances). On such trees, neighbor-joining would have a difficult time, according to Theorem 1, and indeed this is reflected in the shape of the false negative curve for neighbor-joining. The false negative rate for neighbor-joining starts its climb upwards earlier on this tree than it did on the 35-taxon tree, and it climbs higher here than it did in the 35-taxon tree, for every sequence length.

Although not shown in these figures, we explored the performance of these four methods for extremely long sequences (12,800 nucleotides), and even at such lengths the two distance methods still had very high false negative rates: neighbor-joining still missed more than 40% of the edges, and the Buneman Tree missed close to 100%. However, parsimony's performance on this tree was relatively good. Although the false negative rate climbed (and indeed, parsimony may *not* be consistent on this tree – shorter trees than the model tree were found with regularity at high mutation settings), the false negative rate for parsimony fell below 10% for sequences of length 800 and more, even at the highest $p(e)$ setting we examined (.64). Parsimony's false positive rate showed a similar improvement over neighbor-joining at the high end of the $p(e)$ settings, at all sequence lengths we examined (i.e. up to 12,800 nucleotides).

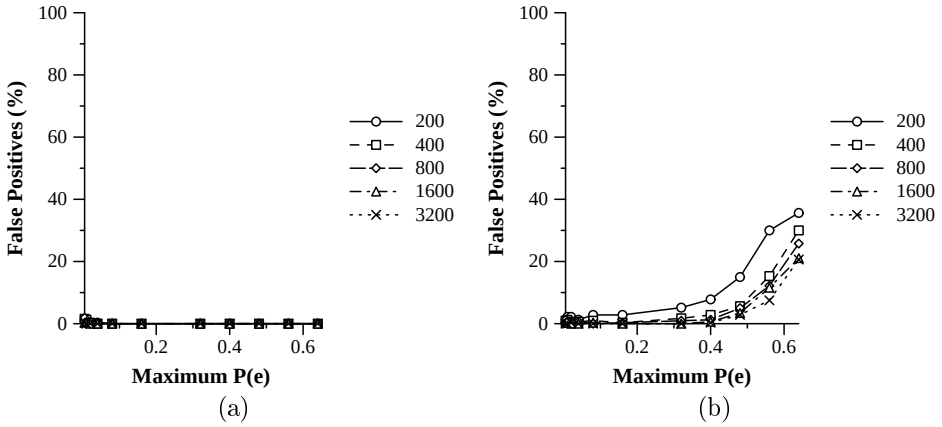


Figure 3: (a) Max p(e) vs. FP, by sequence length, for Buneman on the 35-taxon tree, (b) Max p(e) vs. FP, by sequence length, for HS-strict on the 35-taxon tree

3.3. Conclusions

The comparative analysis of the four methods reveals certain clear trends. First, the Buneman Tree has exceptionally low false positive rates in all the cases we examined, and this is as predicted in [3]; unfortunately, it has the worst false negative rates, and is therefore not really acceptable. The Single Pivot algorithm's performance falls between that of the Buneman Method and the neighbor-joining method on all the trees we examined, and is therefore not competitive with neighbor-joining. Therefore, the major competition is between neighbor joining and heuristic maximum parsimony.

The conditions under which parsimony and neighbor joining will outperform each other is of great interest to systematic biologists, and has been the focus of many experimental studies (as we have already discussed). Our study contributes the following two observations to this discussion:

First, when the model tree has at most moderate divergence (i.e. when λ^* is not particularly large), then both maximum parsimony and neighbor-joining do well. There is a slight advantage in using maximum parsimony over neighbor-joining when the sequences are short and the divergence is low, but this reverses when the sequences are longer.

Second, when the model tree has high divergence, both methods can do poorly. Even under high divergence, neighbor-joining will converge to the model tree given long enough sequences, but on sequence lengths that are typical in real data (up to about 2000 or 3000 nucleotides) it is likely to have very poor accuracy. Maximum parsimony may not converge to the true tree at all, under high divergence, but tends to do *better* (on these trees) than neighbor-joining, when given realistic sequence lengths.

In summary, then, we find that *divergence* (i.e. the maximum distance in the evolutionary tree) has impact on all the methods we examined, but has more impact on neighbor-joining than it has on maximum parsimony.

4. Hybrid Tree Methods

Our experimental study of the four phylogenetic methods we examined indicates that the Buneman and HS-strict trees are almost always very close to being contractions of the model tree, and that the Buneman Tree is a true contraction more than 99% of the time, even on realistic length sequences (i.e. bounded by length 1000). Our experimental study also indicates that neighbor-joining trees typically have a lower incidence of false negatives than parsimony, except from very

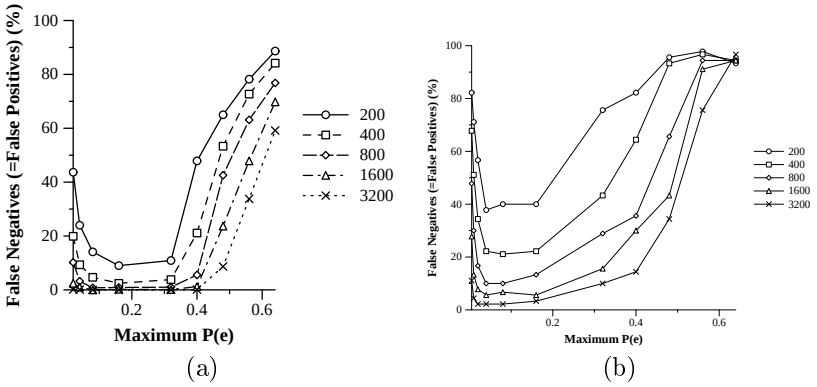


Figure 4: (a) Max p(e) vs. FN/FP by sequence length, for NJ on the 93-taxon tree, (b) Max p(e) vs. FN/FP by sequence length, for Single Pivot on the 93-taxon tree

short sequences, or when the tree has high divergence. These experimental results suggest a **hybrid** approach for tree reconstruction, in which outputs of different methods are appropriately combined so as to get better estimates than either of the methods. Here we describe how we accomplish this combination step.

Let T_1 be a tree which is presumed to have a low false positive rate (and which may be a contraction of the model tree). Let T_2 be a tree which is presumed to have a low false negative rate, so that it may include many of the bipartitions of the model tree. Our objective is to *refine* T_1 to include as many of the bipartitions of T_2 as possible, thus creating a *hybrid* T_3, of T_1 and T_2. Creating T_3 from T_1 and T_2 is very easy to do, and can be done in linear time:

Lemma 1. *The tree T_3 defined by $C(T_3) = C(T_1) \cup \{c \in C(T_2) : c$ is compatible with $C(T_1)\}$ is unique, always exists, and can be reconstructed from T_1 and T_2 in $O(n)$ time.*

For the definition of "compatible", see Section 2. The proof follows easily from characterizations of when binary characters are compatible, material that can be obtained from [30]. Two observations should be clear. If T_1 is a contraction of the model tree, then this technique cannot yield a worse estimation than T_2, but if in addition T_1 is not a contraction of T_2, this technique provably produces a tree *strictly closer* to the model tree than T_2!

We have tested this technique on various combinations, and have found a particular combination of methods which works very well. We first refine the Buneman Tree to incorporate as much of the HS-strict tree as possible, and then further refine this tree to incorporate as much as possible from the neighbor-joining tree. We call this the **Hybrid Tree**. (We do not include the Single Pivot algorithm, because it does not have either the low false positive rate of the Buneman Tree method, nor the low false negative rate of the neighbor-joining method.)

Note that this hybrid method is a statistically consistent method for inferring trees, since its starting point, the Buneman Tree method, is statistically consistent. Furthermore, by constraining the amount of time permitted to the Heuristic search for the most parsimonious trees, the Hybrid Tree is constructible in polynomial time (although we would expect greater accuracy to arise by permitting more time to the heuristic search).

4.1. Comparison of Hybrid Tree to other methods

We now present the results of our experiments in which we compare the Hybrid method to its three component methods (neighbor joining, the Buneman Tree method, and heuristic parsimony).

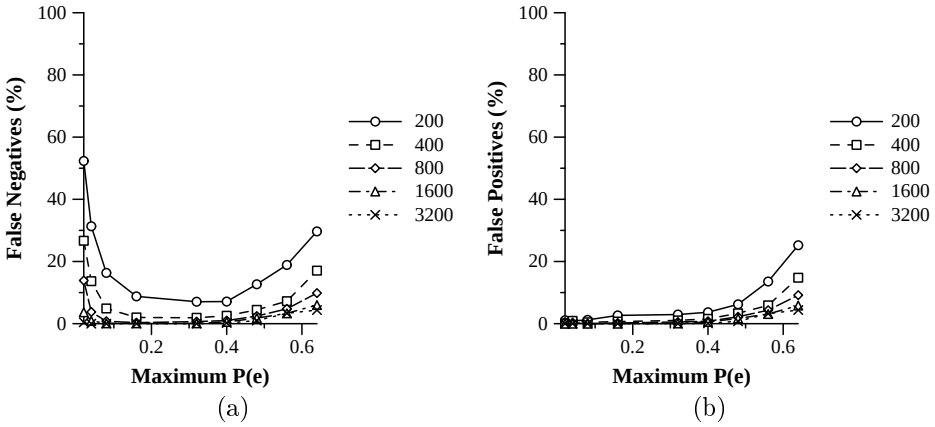


Figure 5: (a) Max p(e) vs. FN, by sequence length, for HS-strict on the 93-taxon tree, (b) Max p(e) vs. FP, by sequence length, for HS-strict on the 93-taxon tree

Although not shown in these results, we also compared Single Pivot on the same data sets. We found the same performance as before: its performance lay very clearly between that of neighbor joining and the Buneman Tree method, and hence was not competitive with either neighbor joining nor with heuristic parsimony.

In the figures below, we will let **HY** refer to the Hybrid method, **BT** refer to the Buneman Tree method, **NJ** refer to neighbor joining, and **HS** refer to heuristic parsimony.

Performance of Hybrid on the 35-taxon tree In Figures 6(a) and 6(b) we show the false negative and false positive rates for these four methods under different levels of divergence, when given short sequences (200 nucleotides).

The Hybrid's false negative rate is often much better than its constituent methods. On the shortest sequences, for example, the Hybrid has consistently lower false negative rate than any of its constituents. On longer sequences, the Hybrid is sometimes worse (with respect to the false negative rate) than neighbor-joining, but always better than the HS-strict tree. The false positives rate of the Hybrid Method falls between the rates for the HS-strict and neighbor-joining methods, and so it is moderate with respect to false positives.

In Figure 7(a) we present the false negative rates for all these methods on the 35-taxon tree for the maximum mutation setting $p(e) = .64$, as we let the sequence length increase. At this setting, there is a fair amount of homoplasy (each site changes about 12 times on the tree). We see that when the sequence length is below 1600, the fewest false negatives are obtained from the Hybrid method, but that for longer sequences, neighbor-joining does better than these two methods (although the Hybrid continues to outperform - albeit slightly - the HS-strict method). The performance with respect to false positives is similar, and is omitted, though the Hybrid obtains as many or more false positives than the HS-strict tree throughout the range of sequence lengths.

Performance of the Hybrid on the 93-taxon tree We present in Figures 7(b) and 8 the same analysis for the 93-taxon trees we gave in Figures 11 and 12.

On the 93 taxon tree, the Hybrid Method has either the same or better false negatives rates than any of the constituent parts throughout the range. This relative performance advantage with respect to the false negative rate over its constituent parts is most visibly noticeable on the short sequences, bounded by at most 800 nucleotides, where the Hybrid's average false negative rate is 90% that of HS-strict's and only 27% that of neighbor-joining, but this pattern is consistent throughout

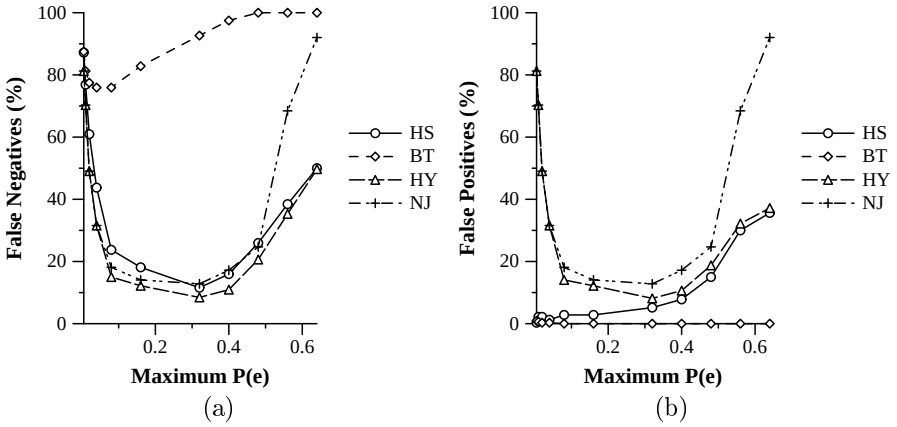


Figure 6: (a) Max p(e) vs. FN, sequence length = 200, on the 35-taxon tree, (b) Max p(e) vs. FP, sequence length = 200, on the 35-taxon tree

the sequence lengths we examined (i.e. up to 3200 nucleotides). The false positive rate is acceptably low, as well, throughout the range of mutation settings; except at low mutation rates, it is close to that of maximum parsimony, and is much lower than that of neighbor joining under high divergence.

5. Comparison to previous experimental studies

There have been many previous experimental studies published in the systematic biology literature, which have addressed the same basic question: how accurately does a given method reconstruct the topology of the model tree, under various conditions. Almost all of these studies have used the same basic methodology: a model tree is constructed on the basis of an analysis of a real biological dataset, sequences are generated at the leaves of that model tree, and performance is evaluated with respect to how well the reconstructed topology compares to the model tree topology. One major difference between our study and most previous studies is that our study has examined large trees (most previous studies have examined trees on at most 20 or so taxa). Another major difference between our study and most others (however, see [22]) is that we have explicitly *varied* the mutation settings on the trees, so as to explore how evolutionary rates affect the performance of different phylogenetic methods. Consequently, our discovery that the accuracy of neighbor-joining and other polynomial time distance methods degrades quickly with increasing divergence, has not been reported in the systematic biology literature.

We now specifically address the two other papers that have experimentally addressed performance on large trees: one is by Hillis [12], and examined a 228 taxon tree, and the other is by Rice and Warnow [22], and examined the conditions under which exact accuracy in topology estimation could be recovered by various methods. Our study here extends the results of Rice and Warnow, by considering degrees of accuracy (i.e. not considering failure to recover the topology precisely as complete failure). We find that although the major phylogenetic methods may fail to recover exactly correct topologies under similar conditions (notably under high divergence), they fail in different ways: maximum parsimony and the Buneman Tree method will tend to have low false positives, even when they fail to recover the true tree, and neighbor-joining will tend to have *lower* false negative rates than maximum parsimony. This relative performance holds, except when the tree has high divergence and the sequences are not particularly long, in which case neighbor joining may have worse false negative rates than maximum parsimony.

A comparison to the Hillis study is also very interesting. Hillis compared neighbor-joining and

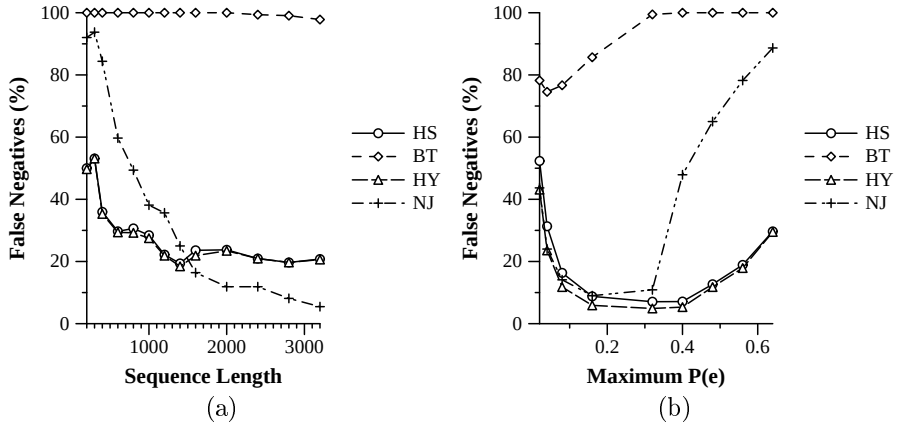


Figure 7: (a) Sequence length vs. FN, max p(e) = 0.64, 35-taxon tree, (b) Max. p(e) vs. FN, sequence length = 200, 93-taxon tree

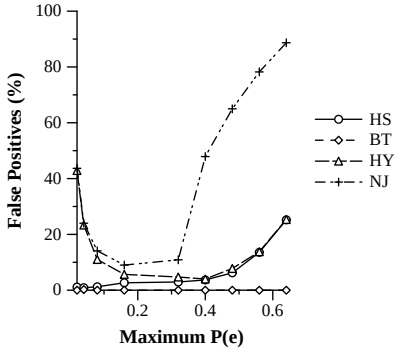


Figure 8: Max. p(e) vs. FP, sequence length = 200, 93-taxon tree

maximum parsimony on a 228 taxon biologically based tree, and observed that both methods recovered the model tree topology from 5000 nucleotides. He concluded that many large trees should be reconstructible using *either* neighbor-joining or parsimony, provided that all edges are short enough (that is, there is not too much evolution happening on any single edge of the tree). However, our experiments, although preliminary, suggest otherwise. Large divergence in a tree can be obtained with enough short edges, and our experiments suggest that the performance of neighbor-joining is compromised under high divergence. It is also worth noting that the model tree used in Hillis's study had *very low* divergence (the average number of changes of a random site on that tree was 2.3, which is exceptionally low, as was commented upon by Purvis and Quicke [19]); hence, Hillis' finding that neighbor joining and maximum parsimony performed well on his tree is completely consistent with our study. Our study also tends to confirm his observation about intensive taxonomic sampling (to reduce edge lengths) will improve parsimony's performance, to the degree that parsimony's performance on the 93 taxon tree was exceptionally good, and it had on average shorter edges than the 35 taxon tree, for each mutation setting. However, we do not agree with Hillis' summary conclusion that neighbor joining would do well under any model condition, provided that the edge lengths were small enough. Our experiments simply do not support that. Instead, our experiments suggest that large divergence in itself causes problems for neighbor joining to a much greater extent than it causes to maximum parsimony.

6. Summary

Our experimental study examined four promising phylogenetic tree reconstruction methods, including the two major estimation methods in systematic biology (the polynomial time distance method *neighbor joining*, and the heuristic used to "solve" the NP-hard optimization problem, *maximum parsimony*), and two polynomial time methods introduced by the theoretical computer science community. Our experimental performance study involved simulating sequence evolution on different model trees, and demonstrated that high divergence in a model tree significantly impairs the accuracy of the three polynomial time distance methods we studied, but does not have *as extreme* an effect upon the accuracy of heuristic maximum parsimony. A more detailed examination of the types of topological errors these methods had revealed that the neighbor joining method had the lowest false negative rates, while the maximum parsimony heuristic and the Buneman Tree method had the lowest false positive rates, under conditions of high divergence.

We used this observation to develop a method based upon combining outputs of these three methods, thus creating a *hybrid method*, and demonstrated the performance of this new method experimentally. Our experimental study showed that this hybrid has either the same number or fewer false negatives than the best of its component methods more than 98% of the time, over the parameter space we explored. It has a distinctly better false negative rate than *any* of its constituent parts on over 60% of the datasets generated on the 93-taxon tree, and over 30% of the datasets generated on the 35-taxon tree. Moreover, the Hybrid Tree is statistically consistent throughout the parameter space of trees under the general Markov model, because the underlying Buneman Tree method is consistent.

References

[1] R. Agarwala, V. Bafna, M. Farach, B. Narayanan, M. Paterson, and M. Thorup. On the approximability of numerical taxonomy: fitting distances by tree metrics. *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms,* 1996.

[2] K. Atteson, *The performance of neighbor-joining algorithms of phylogeny reconstruction*, Computing and Combinatorics, Third Annual International Conference, COCOON '97, Shanghai, China, August 1997, Proceedings. Lecture Notes in Computer Science, 1276, Tao Jiang and D.T. Lee, (Eds.). Springer-Verlag, Berlin, (1997) 101–110.

[3] V. Berry and O. Gascuel, *Inferring evolutionary trees with strong combinatorial evidence.* Proceedings of COCOON 1997.

[4] P. Buneman, The recovery of trees from measures of dissimilarity, in *Mathematics in the Archaeological and Historical Sciences*, F. R. Hodson, D. G. Kendall, P. Tautu, eds.; Edinburgh University Press, Edinburgh, (1971) 387–395.

[5] P. L. Erdős, M. A. Steel, L. A. Székely, and T. J. Warnow, Constructing big trees from short sequences. Proceedings, ICALP, 1997.

[6] P. L. Erdős, M. A. Steel, L. A. Székely, and T. Warnow, *A few logs suffice to build (almost) all trees I*, submitted to Random Structures and Algorithms. ALso appears as DIMACS Technical Report, 97-71.

[7] M. Farach and S. Kannan, *Efficient algorithms for inverting evolution,* Proc. of the 28th Ann. ACM Symposium on the Theory of Computing, 1996.

[8] J. Felsenstein, Cases in which parsimony or compatibility methods will be positively misleading, *Syst. Zool.* **27** (1978), 401–410.

[9] J. Felsenstein, PHYLIP – Phylogeny Inference Package (Version 3.2), *Cladistics,* 5:164-166, 1989.

[10] J. Felsenstein, *Phylogenies from molecular sequences: inference and reliability*, Annu. Rev. Genet., **22** (1988) 521-565.

[11] L. R. Foulds, R. L. Graham, The Steiner problem in phylogeny is NP-complete, *Adv. Appl. Math.* **3**(1982), 43–49.

[12] D. Hillis, Inferring complex phylogenies, *Nature* Vol **383** 12 September, 1996, 130–131.

[13] D. Hillis, Huelsenbeck, J., and C. Cunningham. 1994. Application and accuracy of molecular phylogenies. *Science*, 264:671-677.

[14] J. Huelsenbeck. The robustness of two phylogenetic methods: four-taxon simulations reveal a slight superiority of maximum likelihood over neighbor-joining. *Mol. Biol. Evol.* 12(5):843-849, 1995.

[15] Huelsenbeck, J.P. and D. Hillis. 1993. Success of phylogenetic methods in the four-taxon case. *Syst. Biol.* 42:247-264.

[16] Huelsenbeck, J. 1995. Performance of phylogenetic methods in simulation. *Syst. Biol.* 44:17-48.

[17] T.H. Jukes and C.R. Cantor, *Evolution of Protein Molecules*, in: H.N. Munro, ed., Mammalian Protein Metabolism, Academic Press, New York, (1969) 21-132.

[18] Kuhner, M. and J. Felsenstein, 1994. A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Mol. Biol. Evol.* 11:459-468.

[19] A. Purvis and D. Quicke, TREE (Trends in Ecology and Evolution), 12(2): 49-50, 1997.

[20] K. Rice. Ecat- an evolution simulator. Available from `http://www.cis.upenn.edu/~krice/progs`.

[21] K. Rice, M. Donoghue, and R. Olmstead, *Analyzing large datasets: rbcL 500 revisited*, Systematic Biology, (1997).

[22] K. Rice and T. Warnow, *Parsimony is hard to beat!*, Proceedings, COCOON 1997.

[23] N. Saitou, M. Nei, The neighbor-joining method: a new method for reconstructing phylogenetic trees, *Mol. Biol. Evol.* **4** (1987), 406–425.

[24] N. Saitou and M. Imanishi. Relative efficiencies of the Fitch-Margoliash, maximum parsimony, maximum-likelihood, minimum-evolution, and neighbor-joining methods of phylogenetic reconstructions in obtaining the correct tree. *Mol. Biol. Evol.* 6:514-525, 1989.

[25] M. Schoniger and A. von Haeseler, Performance of maximum likelihood, neighbor-joining, and maximum parsimony methods when sequence sites are not independent. *Syst. Biol.* (1995) 44:4 533-547.

[26] M. L. Sogin, G. Hinkle, and D. D. Leipe, *Universal tree of life*, Nature, **362** (1993) page 795.

[27] John Sourdis and Masatoshi Nei Relative efficiencies of the maximum parsimony and distance-matrix methods in obtaining the correct phylogenetic tree. *Mol. Biol. Evol.* (1996) 5:3 293-311.

[28] Strimmer, K. and A. von Haeseler. 1996. Accuracy of Neighbor-Joining for n-Taxon Trees. *Syst. Biol.*, 45(4):516-523.

[29] D. L. Swofford, PAUP: Phylogenetic analysis using parsimony, version 3.0s. Illinois Natural History Survey, Champaign. 1992.

[30] T. Warnow, *Tree compatibility and inferring evolutionary history*, J. of Algorithms (16), 1994, pp. 388-407.

[31] T. Warnow, *Some combinatorial problems in Phylogenetics*, Invited to appear in the proceedings of the International Colloquium on Combinatorics and Graph Theory, Balatonlelle, Hungary, July 15-20, 1996, eds. A. Gyárfás, L. Lovász, L.A. Székely, in a forthcoming volume of Bolyai Society Mathematical Studies.

[32] M.S. Waterman, T.F. Smith, and W.A. Beyer, *Additive evolutionary trees*, Journal Theoretical Biol., **64** (1977) 199-213.

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 193–203

An experimental study of word-level parallelism in some sorting algorithms

Naila Rahman Rajeev Raman

Department of Computer Science
King's College London
Strand, London WC2R 2LS, U. K.
e-mail: {naila, raman}@dcs.kcl.ac.uk

ABSTRACT

A number of algorithms for fundamental problems such as sorting, searching, priority queues and shortest paths have been proposed recently for the unit-cost RAM with word size w. These algorithms offer dramatic (asymptotic) speedups over classical approaches to these problems. We describe some preliminary steps we have taken towards a systematic evaluation of the practical performance of these algorithms. The results that we obtain are fairly promising.

1. Introduction

In classical data structures and algorithms for manipulating sets of ordered keys, it is common to assume that the relative order of two keys can only be determined by comparing them in unit time. While this assumption makes for generality, real computers have many other unit-time operations besides comparisons. Indeed, the hash table, another common data structure, applies unit-time operations other than comparisons to a key (e.g. arithmetic operations to evaluate a hash function). In order to support unit-time (arithmetic and logical) operations on word-sized operands, a "sequential" processor needs parallelism at the level of hardware circuits, which we refer to as **word-level parallelism (WLP)**. Some processors available nowadays support operations on 64-bit operands, and it is expected that this will become the norm in the near future. If so, this would be a substantial amount of WLP, which can be exploited in far more imaginative ways than simply performing comparisons, especially if the keys are integers or floating-point numbers.

A formal framework for exploiting WLP is provided by the random-access machine (RAM) model, by including the word size w of the machine as a parameter of the model, and charging a unit cost for standard operations on w-bit operands. There has been a great deal of recent interest in developing algorithms for fundamental problems such as sorting, searching and priority queues on this model: a recent survey [13] cites over 20 papers which have been published since 1995 on these topics. This research has resulted in fundamental new insights regarding the power of this model. By exploiting WLP in ingenious ways, several novel algorithms for core problems have been obtained, which are significantly faster than their classical counterparts: e.g., searching, insertions and deletions in a set of n integer or floating-point keys can be accomplished in $O(\sqrt{\log n})$ time per operation [2, 3, 11, 16, 13] and priority queue operations in $O(\log \log n)$ time [18]. By contrast, classical solutions to these problems require $\Theta(\log n)$ time per operation. Striking improvements over comparison-based approaches have also been obtained for sorting [3, 5, 16] and computing shortest paths [9, 18, 19, 16].

These results are related to an older tradition of RAM algorithms (see e.g. [21, 15]) and follow a direction first taken by [11, 12]. One way to view the difference between the 'old' and 'new' algorithms is that the old algorithms have a complexity which increases with w, but are quite fast

if w is relatively small[1]. The new algorithms by and large complement the older ones by offering speedups for larger values of w, that is, when the amount of WLP becomes large.

However, with a few exceptions, there have not been very many experimental evaluations of the new algorithms. What we would hope to achieve in a systematic programme of experimental evaluation of these algorithms is to (i) determine the word size at which (simplified and streamlined) versions of the new algorithms outperform existing ones (ii) experimentally study the techniques used in the new algorithms with a view to obtaining hybrid algorithms which will provide speedups in real life (if not for the current generation of mass-market CPUs, then at least for the next generation).

It should be said that we do not expect that the new algorithms—even in suitably simplified form—will lead to improvements when run on the current generation of CPUs. This is because most current mass-market CPUs only fully support operations on 32-bit data, with limited support (of varying effectiveness) for 64-bit data, and the new algorithms need fairly large amounts of WLP to be effective (e.g. the main algorithm of [11] required w to be of the order of a few thousand bits before it improved on comparison-based algorithms). On the other hand, we believe that recent hardware developments should make it much easier to translate these advances into practical speedups. The new algorithms mentioned above first reduce the original data structuring problem, which is defined for word-sized data, to one on much shorter data. Then, multiple data are packed into a single word and processed in a SIMD manner, by using standard instructions in rather elaborate ways. However, this may be greatly sped up by making use of built-in SIMD instructions with which many current processors (e.g. HP PA-RISC, Intel Pentium MMX and Sun Ultra-Sparc) are equipped.

In this abstract we describe some preliminary steps towards the goals outlined above. First we describe a simulator which allows an algorithm to be simulated on a (virtual) RAM with word size w, for any $w \geq 32$ specified by the user. The objectives of the simulator and the approach to its implementation are described in Section 2.

In Section 3, we describe a simplified implementation of the Kirkpatrick-Reisch (KR) sorting algorithm [15]. This algorithm has a running time of $O(n(1 + \log\lceil\frac{w}{\log n}\rceil))$ as compared to the running time of radix sort [7], which is $O(n\lceil\frac{w}{\log n}\rceil)$. Our results indicate, e.g., that for a few million keys and $w \geq 256$, KR is superior to radix sort on a RAM with word size w. Our results also suggest that a *hybrid* KR sort, which switches to radix sort when appropriate, would outperform both KR and radix sort for a few million keys when $w \geq 128$ both on a RAM and on real-life CPUs. Since KR is generally considered an impractical algorithm, we feel these conclusions are surprising.

In Section 4, we discuss a 'packed' merging algorithm introduced in [1]. We are given two sorted lists of n keys, where each key is at most w/k bits long. Assuming a sorted list of n keys is represented in n/k words, with k keys packed to a word, the algorithm of [1] merges the two lists in $O((n \log k)/k)$ time. An experimental study with packing 8 items to a word on a 64 bit machine revealed that although the packed merging algorithm was clearly inferior to the standard merging algorithm on a RAM with wordsize 64 bits, in real life (using `long long int`s, a 64-bit integer type supported by Gnu C++) the difference was much smaller. We believe this result highlights some advantages of algorithms which use WLP which might not be apparent from operation counts.

All timing results given here were obtained on an UltraSparc 2×300 MHz with 512 Mb memory running SunOS 5.5.1 and using the Gnu C++ compiler (gcc 2.7.2.3). This machine is normally lightly loaded.

2. Simulator

In this section we describe a (software) simulator which allows us to experimentally evaluate the performance of algorithms which use WLP on machines with larger word sizes than currently available. The main objectives of the simulator are:

a. To be as transparent as possible,

[1] We consider w to be 'relatively small' if $w < c \log n$ for a small constant c.

b. To provide an indication of the performance of an algorithm on a hypothetical w-bit machine,

c. To be efficient, defined as:

> For sufficiently large n, n successive executions of an operation on w-bit integers on the simulator, running on a w'-bit machine, should take not much more than w/w' times the time taken by n executions of the corresponding operation on the native integer type of the w'-bit machine.

We are aiming for a maximum *simulation overhead* of 1 (this number is somewhat arbitrary) for most operations, where

$$\text{simuation overhead} = \left(\frac{\text{average time for } w\text{-bit operation}}{\text{average time for } w'\text{-bit operation} \cdot (w/w')} \right) - 1.$$

Objective (c) is not meant to be a formal goal as it clearly cannot be achieved in general (e.g. operations such as multiplication require time which is super-linear in the size of the input). Instead it is an indication of the criterion we will use for evaluating the efficiency of the simulator. Note that (c) implies one solution to (b) as well—divide the running time of the simulated algorithm by w/w' to get a rough measure.

Apart from the obvious reason for wanting the simulator to be efficient — i.e. to run simulations quickly — property (c) may enable the use of an algorithm running on a virtual machine with large wordsize to efficiently solve a problem in the real world. As an example, we mention a string-matching problem which arose in musical pattern-matching [8]. A naive algorithm for this problem runs in $O(mn)$ time where m is the length of the pattern and n the length of the text, and nothing better is known for parameter values of real-life interest, and a variant of this algorithm, which uses WLP, runs in $O(n\lceil m/w \rceil)$ time on a machine with wordsize w. The current implementation of the WLP variant is an order of magnitude faster than demanded by the application, but has the restriction that $m \leq 32$ since integers are 32 bits long. Although this restriction does not matter at the moment, an efficient simulator would offer a simple way to weaken it as and when needed, while maintaining acceptable performance.

2.1. The class `Longint`

Our current simulator is simply a class `Longint` in C++ which behaves like w-bit integers, where w is fixed at the time of compiling the class[2]. All standard operations which C++ allows on integers will be provided in the class including arithmetic, bitwise logical, comparison, shifts, automatic conversion of `int`s, `char`s etc. to `Longint`, other mixed-mode expressions, overloading of the input and output stream operators etc. The current status of the implementation is described in Appendix A.1. Due to the operator overloading mechanism in C++, it is only necessary to make minor changes to a standard program written in C++ which operates on (say 32-bit) `int` data, and get a program which operates on `Longint`s. See Appendix A.2 for some of the minor changes which may have to be made.

At the time of compiling the class can be made to maintain counts of all operations performed on instances of class `Longint`. The operation counts thereby provided can be used as rough indicators of the performance of a simulated algorithm. Note that C++ compilers are not permitted to optimise expressions involving objects, so the source code has to be hand-optimised to e.g. eliminate common subexpressions[3]. Hence the operation counts are only an approximate indicator of the number of instructions performed on the virtual w-bit CPU, and of course they are probably an even cruder indicator of the actual running time on a real w-bit CPU (if and when that becomes a reality) as

[2] A cleaner approach might be to make a class template `Longint<w>`, but this seemed to incur an overhead, at least using Microsoft Visual C++.

[3] Our experiments also include counts of operations on `int`s, this warning applies to those counts as well.

several architectural optimisations e.g. instruction pipelining, instruction-level parallelism, locality of reference etc. are ignored. Nevertheless, we believe that since algorithm designers primarily use the RAM model, operation counts are a useful indicator of algorithm performance.

Alternative methods of simulation were considered, and rejected for various reasons including portability, "future-proofing" and ease of construction including building an emulator for say 686 or Ultrasparc assembly code produced by a compiler, and constructing a language, compiler and virtual machine.

2.2. Related Work

LEDA provides the class `integer`, which allows arbitrary-precision integer arithmetic. We cannot use this class directly as rules of arithmetic are usually different for the `int` and `unsigned int` types supported by C++ (e.g. all arithmetic on `unsigned int` is done mod 2^w). Also, by fixing the wordsize at compile time we are able to get much faster object creation, destruction and copying, significantly reducing the overhead. Even the current version of the simulator, where the code for the individual operations is not optimised, seems comparable in speed with the LEDA `integer` class for most operations (not including multiplication) when dealing with comparable-sized numbers. We have done some pilot experiments with tailoring the code specifically for each 'reasonable' wordsize, for the crucial shift operation. These suggest that the optimised version would perform shifts on 128-bit integers about 3 to 3.5 times faster than in LEDA, and with an overall simulation overhead of just over 100%. Further improvements might result from coding in assembly language.

Gnu C++ compilers also support the type `long long int` (64-bit integers) which the compiler translates into 32-bit assembly code if necessary. Microsoft Visual C++ also has a similar data type. The simulation overhead (as defined in (c)) seems to be small: about 25% on an UltraSparc and perhaps about 50% on a Pentium II. Unfortunately, larger integers are not supported, and nor is addressing using `long long int`s.

3. KR sort

In this section, we describe a simplified implementation of the Kirkpatrick-Reisch (KR) sorting algorithm [15]. This algorithm has a running time of $O(n(1 + \log\lceil\frac{w}{\log n}\rceil))$, and works roughly as follows. In $O(n)$ time, the problem of sorting n keys of $2t$ bits each is reduced to the problem of sorting n keys of t bits each (initially $t = w/2$). When the keys to be sorted are $O(\log n)$ bits long (i.e. after $\log(\lceil\frac{w}{\log n}\rceil)$ levels of recursion) they are sorted using radix sort. While we use the same general idea, our implementation differs in several respects.

3.1. Implementation overview

In the following discussion, assume n is the number of items, w is the initial wordsize of our keys, and $2t$ is the size of the keys input to the current recursive level. As in KR we consider each key of $2t$ bits as consisting of a *range* (its top t bits) and a *value* (its bottom t bits). In each level of recursion, we group together some keys with the same range, and choose one of them as a *representative* of the group. Thus, in each level of recursion, each key either becomes a representative at that level (i.e. is *promoted*) or is not a representative at that level. An important aspect of our implementation is to explicitly maintain the promotion histories of each key in the input, i.e, we maintain with each key a sequence of bits which records, for each level of recursion, whether the key was a representative at that level or not (KR does this implicitly).

At each level of recursion, keys which share the same promotion history PH entering this level of recursion, belong to a common sorting sub-problem. This set of keys is denoted by $k(PH)$. Each key in the original input contributes a consecutive sequence of $2t$ bits (which sequence is determined by its promotion history) to be sorted at this level. A key x is a representative from the group of keys $k(PH)$ if, among all the keys in $k(PH)$ with the same range as x, it has the maximum value.

Avg op count per item, $w = 64$					Avg op count per item, $w = 128$					Avg op count per item, $w = 256$				
n	KR	Radix sort			n	KR	Radix sort			n	KR	Radix sort		
		11	16	22			11	16	22			11	16	22
10K	124	76	254	10K	10K	158	152	507	20K	10K	191	303	1K	40K
100K	78	67	65	1K	100K	110	134	130	2K	100K	143	267	260	4K
1M	70	66	46	136	1M	102	132	92	267	1M	133	264	184	534
2M	70	66	45	86	2M	102	132	90	166	2M	133	264	180	333
4M	70	66	45	61	4M	101	132	89	116	4M	F	264	178	232
8M	70	66	45	49	8M	F	132	88	91	8M	F	264	177	182

Figure 1: Experimental evaluation of KR sort vs radix sort on RAM simulator. F denotes the test failed because we ran out of memory; experiments limited to 400Mb of virtual memory.

All representatives at this level are promoted, and contribute their ranges to the recursive sorting problem, while all other keys are not promoted, and contribute their values to the recursive sorting problem. All promotion histories are updated appropriately.

In the recursive sorting problem, we sort each key by the promotion history, as well as by the t bits it contributed, with the promotion history being the primary key. If t is sufficiently small ($t = 16$ currently), we solve the recursive problem by two passes of counting sort—one for the t bits and one for the promotion histories.

When the recursive call finishes, the keys with the same promotion history at the start of this level of recursion are in a contiguous segment, sorted by the t bits they contributed to the recursive call, with the representatives at the end of the segment. We now count the number of keys which have the same range as their representative, and using this information, we permute the keys in this contiguous segment so that they are sorted according to their full $2t$ bits.

We highlight some features of our algorithm:

i. Keys are not moved while going down recursive levels, but are moved on the way up. Furthermore, on the way up the keys are moved within (hopefully small) localised segments. We believe this would contribute to a greater efficiency in the real world.

ii. In contrast to KR, no data is saved on the stack—the 'state' of the algorithm is encoded in the promotion histories, which are passed to the recursive call.

iii. The actual implementation discards the promotion histories on the way up and instead maintains an implicit representation to avoid moving promotion histories and thus saving on operations. (This implicit representation is of size 2^r where r is the number of levels of recursion; since $r \leq \log w$ this is of size at most w).

3.2. Experimental Results

We have performed many experimental evaluations of the above KR sort variant, as well as of other variants, both on the simulator and on 32- and 64-bit integers on a Pentium II (using Microsoft Visual C++) and Ultra Sparc (using GNU C++). We have a fairly complete set of data on the simulator and the Ultra Sparc at present.

Figure 1 summarises the experiments we have performed with the KR sort version described in the previous section on the simulator. Both algorithms were tested on n random w-bit data, for $n = 10\text{K}, 100\text{K}, 1\text{M}, 2\text{M}, 4\text{M}$ and 8M and $w = 64, 128$ and 256 ($\text{K} = 1000$, $\text{M} = 1000\text{K}$). The table shows per-item total operation counts, i.e., the total of all the operations on `Longints` and `ints`, divided by n. Radix sort was tested with radix $11, 16$ and 22. These radix values minimised (locally) operation counts for $w = 64$ and were retained for larger values of w. For a RAM with wordsize w our conclusions are basically as follows:

Time(s) per pass					
n	KR (hash)	KR (no hash)	Radix sort		
			11	16	22
10K	0.01	0.01	0.002	0.01	0.76
100K	0.19	0.11	0.03	0.05	0.87
1M	2.45	1.56	0.48	0.64	2.09
2M	5.05	3.28	0.95	1.40	3.49
4M	10.04	6.68	1.91	2.64	6.26
8M	21.52	13.91	4.09	5.84	11.75

Figure 2: Experimental evaluation of 64-bit KR sort vs radix sort.

- If, for a given wordsize and input size, we choose the radix which minimises the number of operations, then with this radix, each pass of radix sort requires between 11 to 12 instructions per item being sorted. For example, for $w = 128$, we need 88 operations per item for 8 passes when $n = 8$M and 92 operations per item for 8 passes when $n = 1$M.
- Except for small values of n, each range reduction pass of KR sort requires 31 to 32 instructions per item being sorted. These numbers were obtained by comparing the per-item operation counts for word sizes k and $2k$ for a given value of n. Sorting 2k-bit keys requires one pass more than sorting k-bit keys. For example, 133 operations per item for $w = 256$ and $n =$ 2M vs. 102 operations per item for $w = 128$ and $n = 2$M gives the cost of one KR pass as $133 - 102 = 31$.

We therefore conclude that on a RAM with wordsize w, one pass of KR sort halves the number of bits to be considered, but costs between 2.8 and 2.9 times the number of operations, compared to one pass of radix sort. Hence, running KR sort all the way to 16 bits may not be the optimum choice. For example, going from 64 to 32 bits only eliminates about 2 passes of radix sort, but costs upto 2.9 times as much. The following *hybrid* KR-radix sort will match or outperform either KR or radix sort alone: apply KR sort until the wordsize is such that radix sort beats KR sort, then switch to radix sort. We intend to perform further experiments with hybrid KR-radix sort.

Figure 2 summarises experiments performed on the Ultra Sparc with 64-bit integers (using GNU C++ `long long int`). Since KR sort requires a large array of size $2^{w/2}$ words to group keys, for $w \geq 64$ we are forced to implement this array using some form of hashing - we use double hashing [7] with a universal hash function [10] and a hash table density of at most 0.5. The results indicate that the overhead of hashing causes KR sort to perform poorly for $w = 64$. In order to determine the performance of the algorithm without the overhead of hashing - i.e. without the cost of computing a hash function and of collision resolution - Figure 2 also details timings for KR sort where every access to a hash table is replaced by an access to an arbitrary (random) address in an array of size $O(n)$. Of course, in this case the data is not actually sorted. We again detail results for radix sort with radix 11, 16 and 22. The experiments were on n random 64-bit data, for $n =$ 10K, 100K, 1M, 2M, 4M and 8M (K=1000, M=1000K).

We conclude that except for small values of n, with hashing each pass of KR sort takes about 3.6 to 3.8 times as long as each pass of radix sort. Without hashing each pass of KR sort takes about 2.4 to 2.8 times as long as each pass of radix sort.

For real machine integers, the results suggest that for 128-bit integers KR sort will be competitive with radix sort if we can group keys without using a hash table. Using hashing KR sort will outperform radix sort for 256-bit integers. We believe the *hybrid* KR-radix sort will outperform radix sort on 128-bit integers even if we use a hash table - since 1 KR range reduction pass can be used to replace 4 radix sort passes at a lower cost.

4. Packed Operations

We now consider the packed merging algorithm of [1]. The input to this algorithm is two sorted packed lists, each containing n (w/k)-bit unsigned keys. Each list is in packed format, i.e., consists of n/k words, each containing k keys stored in equal-sized fields. (When referring to the order of bits in a word, 'left' is in the direction of the most significant bit and 'right' is in the direction of the least significant bit.) The lists are assumed to be in nonincreasing order, i.e., the keys in a word are nonincreasing from left to right, and all keys in a word are no smaller than all keys in the next word in the list. The packed merging algorithm of [1] produces a packed merged list in $O((n \log k)/k)$ time from this input. Packed merging plays an important role in the new WLP algorithms:

(a) It is used extensively as a subroutine, e.g., a priority queue of [18] is based on packed merging.

(b) Repeated packed merging leads to an $O(n\frac{\log n \log k}{k})$ time packed merge-sorting algorithm for w/k-bit keys. This is a key ingredient in the AHNR sorting algorithm [5]. Essentially the AHNR algorithm applies $\lceil \log \log n \rceil$ passes of the KR range reduction, at which point the keys to be sorted are $O(w/\log n)$ bits long. Switching to packed merge sort at this point terminates the recursion at $O(n \log \log n)$ cost. Since each of the passes of the KR range reduction takes $O(n)$ time we get an $O(n \log \log n)$ sorting algorithm.

However, we should not expect AHNR to simultaneously beat a hybrid KR algorithm and a fast comparison-based algorithm like quicksort, even for moderately large wordsizes. Suppose AHNR switches to packed sorting when the keys are $b \geq 16$ bits long, b a power of 2, and suppose this costs $c'n\frac{\log n \log k}{k}$ steps, where $k = w/b \geq 2$ is the packing factor. If we were to use radix sort (with radix 16) to finish off the sorting we would incur a cost of $c \cdot (b/16) \cdot n$ steps. Therefore, if packed sorting is better than radix sort at this point, we conclude that $w \geq 16 \cdot (c'/c) \cdot \log n \log k$. Conservatively estimating $c'/c \geq 2$ — radix sort has very low constants, while packed sort is based on merge sort, not the fastest of comparison-based sorts — and $k \geq 4$ — otherwise it seems likely that the cost of sorting the original input using quicksort will be less than the cost of just the packed sorting phase — we conclude that we should not expect AHNR to beat hybrid KR sort and quicksort simultaneously unless $w \geq 64 \log n$.

4.1. Implementation of packed merge

The calculation above led us to study the packed merge in isolation (without the additional overhead of merge sort). The two main primitives used by the packed merge are:

`bitonic_sort`: Given a bitonic sequence of k keys (each of w/k bits) packed into a single word, this implements Batcher's bitonic merge to output a word which contains the keys in sorted order, in $O(\log k)$ steps. The sorting can be either non-increasing or non-decreasing order. The ideas used are slightly optimised versions of those described in [1]. Perhaps the main difference is that [1] assumes that the keys are really $w/k - 1$ bits long, so that they can be stored in fields of w/k bits each leaving the leftmost bit in each field free (the *test* bit); we allow keys to be as large as the field size allows. This reduces the operation count.

`word_merge`: Given two words, each containing a sorted sequence of k keys of w/k bits, with one word sorted in non-decreasing order and the other in non-increasing order, this outputs two new words, the first containing the k largest keys in the input and the second containing the k smallest keys in the input. Again, the user may choose if the keys in an output word are to be in non-increasing or non-decreasing order.

The two main high-level optimisations we made to the algorithm as described by [1] were to use the word-merge primitive within a sequental merging algorithm rather than within a parallel algorithm, and secondly, to avoid the use of the *Reverse* operation by judiciously choosing the sense in which each sorted word is output by `word_merge`. The pseudocode is given in Appendix B. One optimisation that we could make regards the 'componentwise comparison' operation shown in Fig 3, which is essential for bitonic sorting. We compute it as:

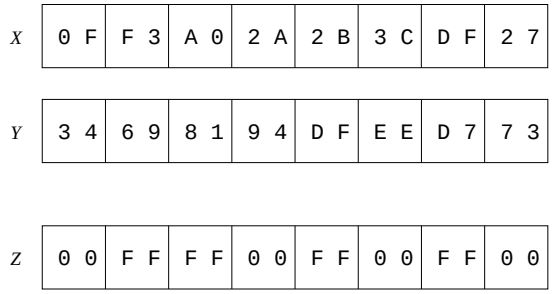


Figure 3: Pairwise comparison operation. The contents of fields in words X and Y are shown in hexadecimal, and are to be regarded as unsigned integers. The word Z contains all 1s in a field if the corresponding field in X is greater than or equal to the one in Y, and all 0s otherwise.

	Op Counts ($\times 10^6$)		Timings (s)			
$2n$	normal merge	packed merge	normal (random)	packed merge	packing time	normal (skewed)
512K	3.66	35.98	0.09	0.10	0.03	0.07
1M	7.33	71.96	0.18	0.20	0.07	0.13
2M	14.68	143.91	0.36	0.41	0.14	0.26
4M	29.36	287.82	0.73	0.83	0.28	0.51

Figure 4: Experimental evaluation of packed vs normal merge

```
Z = ((X | mask1) - (Y & mask2)) & ~(X ^ Y) | (X & ~Y) & mask1;
Z |= (Z - (Z >> (f-1)));
```

where `f` is the width of a field and `mask1` is a mask which has 1s in the leftmost bit of each field and zeros elsewhere. However, one instruction ("Packed Compare for Greater Than" [14]) in the Intel MMX instruction set suffices to achieve this.

4.2. Experimental results

We performed experiments comparing normal merge with packed merge. Normal merge was tested on two kinds of inputs:

random: two sorted lists each consisting of n random 64-bit integers, and

skewed: two sorted lists of size n, one containing the values: $3, 3, \ldots, 3, 0$ and the other the values $2, 2, \ldots, 2, 1$.

Packed merge was tested on two sorted packed lists containing n random 8-bit integers[4]. Hence a packing factor of 8 was allowed, and the input consisted of $n/8$ words for each list. These tests were perfomed on four input sizes: n = 262K, 512K, 1M and 2M (where K = 1024 and M = 1024K) and with two different data types:

`Longint`**:** experiments were run on the simulator. The results shown are the sum of `Longint` and `int` operations. Under the simulator, the normal merge produced virtually identical results for skewed and random data, so only one result is shown.

[4]In fact the integers were 7 bits long since we do not yet support unsigned `Longints` so the test in line (9) of the packed merge in Appendix B would not work.

Gnu C++ `long long int`: experiments were run on the UltraSparc. Results shown are the average running times over three runs.

The results of the experiments are given in Fig 4. It is interesting to note that despite a factor of 10 difference in the operation counts for packed merge and normal merge on random data, the difference in 'real' running time is only 10-15%. One factor may be that the packed algorithm only reads and writes $2n$ bytes from memory while the normal algorithm reads and writes $16n$ bytes. In order to compensate for this we separately measured the time to pack the input assuming it was given as 8-bit values in in unpacked format (i.e. as $2n$ `long long int`s). Even adding this in twice (for packing and unpacking) still leaves packed merge no worse than a factor of 2 off from normal merge. Note that the cost of packing and unpacking is not included in the operation counts for packed merge.

Possibly one of the gains made by the packed algorithm is the lower frequency of branch instructions, which are usually expensive. One way to quantify this may be to assume that a predictable branch has a similar cost to no branch. With the skewed data as input, the branches become very predictable and the running time of merge on skewed data is about 30% faster, and we could use this figure as a rough indicator of the cost of a branch. Another factor might be the granularity of the computation: once a `long long int` is loaded into registers, many operations are performed on it before it is stored back into main memory.

Acknowledgements

We thank Tomasz Radzik for useful discussions. The simulator developed from a suggestion by Jesper Träff.

References

[1] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation* **136** (1997), pp. 25–51.

[2] A. Andersson. Sublogarithmic searching without multiplications. In *Proc. 36th IEEE Symp. Found. Comp. Sci.*, 1995, pp. 655–663.

[3] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th IEEE Symp. Found. Comp. Sci.*, 1996, pp. 135–141.

[4] A. Andersson and M. Thorup. A pragmatic implementation of monotone priority queues. From the DIMACS'96 implementation challenge.

[5] A. Andersson, T. Hagerup, S. Nilsson and R. Raman. Sorting in linear time? To appear in *J. Computer and Systems Sciences*, selected papers STOC '95. Prel. vers. in *Proc. 27th ACM Symp. Theory of Computation*, 1995, pp. 427–436.

[6] A. Andersson. What are the basic principles for sorting and searching? In B. Randell, ed. *Algorithms: Proc. Joint ICL/ U. Newcastle Seminar*. U. Newcastle-upon-Tyne, 1996.

[7] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[8] T. Crawford, C. S. Iliopoulos and R. Raman. String-matching techniques for musical similarity and melodic recognition. *Computing and Musicology* **11** (1998), to appear.

[9] B. V. Cherkassky, A. V. Goldberg and C. Silverstein. Buckets, heaps, lists and monotone priority queues. In *Proc. 8th ACM-SIAM Symp. Discr. Algs.*, 1997, pp. 83–92.

[10] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, **25** (1997), pp. 19–51.

[11] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comp. Sys. Sci.*, **47** (1993), pp. 424–436.

[12] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comp. Sys. Sci.*, 48:533-551, 1994.

[13] T. Hagerup. Sorting and searching on the Word RAM. In *Proceedings, 15th Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Vol. 1373, pp. 366–398.

[14] Intel Architecture Software Developers Manual, Vol 1: Basic Architecture. Intel Corp., Santa Clara, CA, 1997.

[15] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theor. Comput. Sci* **28** (1984), pp. 263–276.

[16] R. Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proc. 4th European Symp. Algos.*, LNCS 1136, pp. 121–137, 1996.

[17] R. Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2): 81–87, 1997. An earlier version is available as TR 96-13, King's College London, 1996, from http://www.dcs.kcl.ac.uk.

[18] M. Thorup. On RAM Priority Queues. In *Proc. 7th ACM-SIAM Symp. Discr. Algs.*, 1996, pp. 59–67.

[19] M. Thorup. Undirected single-source shortest paths in linear time. In *Proc. 38th IEEE Symp. Found. Comp. Sci.*, 1997, pp. 12–21.

[20] M. Thorup. Faster deterministic sorting and priority queues in linear space, pp. 550–555. In *Proc. 9th ACM-SIAM Symp. Discr. Algs.*, 1998.

[21] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, **6** (1977), pp. 80–82.

A. Simulator details

A.1. Porting C++ code to the simulator

In addition to changing declaration of variables from `int`s to `Longint`s only the following changes need to be made:

- Constants may have to be explicitly cast to `Longint`. For example if `X` is a `Longint` of 128 bits, the expression

  ```
  X = 1 << 30 << 30;
  ```

  will not store 2^{60} in `X`. Since 1 is a constant of type `int` the expression on the RHS will evaluated as an `int`. (The expression `1 << 60` will be be flagged by the compiler.)

- Arrays indexed by a `Longint` have to be declared as instances of a separate class (similar to the `h_array` class in LEDA). This is implemented as a hash table using double hashing, with a universal hash function and a density that can be specified at compile time.

- Obviously the programmer should ensure that expressions which rely on a 32-bit word size continue to work as intended. (e.g the expression `(X << l) >> l` applied to a 32-bit value will shift out the left `l` bits of `X` on a 32-bit machine, but may leave `X` unchanged if it is a 32-bit value on a 64-bit machine).

A.2. Implementation Status

At the time of writing the following features had not been implemented:

- Overloading of `<<` and `>>` for input/output streams,
- Integer division and modulus,
- Unsigned `Longint`s,
- A complete suite of inward and (where possible) outward conversion utilities, and mixed-mode operators. For instance, `0.5 * i` where `i` is a `Longint`, is currently illegal.

Optimising the simulator is also a task we are currently addressing.

B. Pseudocode for packed merge

The code on the left is the normal code for merging. The one on the right is for packed merging and uses the representation and functions described in Section 3.

```
 (1) int i = j = k = 1;
 (2)
 (3) while((i <= q) && (j <= r))
 (4) {
 (5)  if(A[i] >= B[j])
 (6)     C[k++] = A[i++];
 (7)  else
 (8)     C[k++] = B[j++];
 (9) }
(10)
(11) if(i > q)
(12)   while (j <= r) {
(13)     C[k++] = B[j++];
(14)  }
(15) else
(16)  while (i <= q) {
(17)     C[k++] = A[i++];
(18)  }
```

```
 (1) int i = j = k = 1;
 (2)
 (3) X = A[i++], Y = B[j++];
 (4) ascending_bitonic_sort(Y);
 (5) word_merge(X, Y, DESCENDING, ASCENDING);
 (6) C[k++] = X;
 (7)
 (8) while((i <= q) && (j <= r)) {
 (9)   if(A[i] >= B[j])  // unsigned comparision
(10)     X = A[i++];
(11)   else
(12)     X = B[j++];
(13)   word_merge(X, Y, DESCENDING, ASCENDING);
(14)   C[k++] = X;
(15) }
(16)
(17) if(i > q) // items in A exhausted
(18)   while (j <= r) {
(19)     X = B[j++];
(20)     word_merge(X, Y, DESCENDING, ASCENDING);
(21)     C[k++] = X;
(22)   }
(23) else
(24)   // items in B exhausted, symmetric
(25)
(26) descending_bitonic_sort(Y, M, f);
(27) B[k++] = Y;
```

Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 204–212

Who is Interested in Algorithms and Why? Lessons from the Stony Brook Algorithms Repository

Steven S. Skiena[1]
Department of Computer Science
SUNY Stony Brook, NY 11794-4400
skiena@cs.sunysb.edu

ABSTRACT

We present "market research" for the field of combinatorial algorithms and algorithm engineering, attempting to determine which algorithmic problems are most in demand in applications. We analyze 249,656 WWW hits recorded on the Stony Brook Algorithms Repository (http://www.cs.sunysb.edu/~algorith), to determine the relative level of interest among 75 algorithmic problems and the extent to which publicly available algorithm implementations satisfy this demand.

1. Introduction

A primary goal of algorithm engineering is to provide practitioners with well-engineered solutions to important algorithmic problems. Our beliefs as to which problems *are* important to practitioners have been based primarily on anecdotal evidence. To provide more objective information, it seems useful to conduct "market research" for the field of combinatorial algorithms, by determining which algorithmic problems are most in demand in applications, and how well currently available implementations satisfy this demand.

This paper is an attempt to answer these questions. We present an analysis of 249,656 WWW hits recorded on the Stony Brook Algorithms Repository over a ten-week period, from February 10 to April 26, 1998. The Repository (http://www.cs.sunysb.edu/~algorith) provide a resource where programmers, engineers, and scientists can go to find implementations of algorithms for fundamental problems. User feedback and WWW traffic statistics suggest that it has proven valuable to people with widely varying degrees of algorithmic sophistication.

The structure of the Algorithms Repository makes it well suited to measure the interest in different algorithmic problems. For each of 75 fundamental algorithm problems, we have collected the best publicly available implementations that we could find. These problems have been indexed in major web search engines, so anyone conducting a search for information on a combinatorial algorithm problem is likely to stumble across our site. Further, special indexes and hyperlinks aboard our site help guide users to other relevant information.

This paper is organized as follows. In Section 2, we discuss the structure of the Algorithms Repository in more depth, to provide better insight into the nature of the data we present below. In Section 3, we analyze WWW traffic to determine the most popular and least popular algorithmic problems. In Section 4, we report on what our users are finding. Each implementation available on the Repository has been rated as to its usefulness for the corresponding problem. By studying these ratings, we can assess the current state of the art of combinatorial computing, and see how well it matches user demand. Finally, in Section 5, we attempt to get a handle on where the interest in algorithms is located, both geographically and professionally.

Any polling-based research is subject to a variety of bias and ambiguities. I make no grand claims as to how accurately this data measures the relative importance of different algorithmic research to society. I found many of the results quite surprising, and hope they will be of interest to the algorithmic community.

Problem Category	Index Hits	Subsection Hits	Problems
Data Structures	1067	2651	6
Numerical Problems	735	2271	11
Combinatorial Problems	539	2108	10
Graph Problems: Polynomial	842	3955	12
Graph Problems: Hard	729	2846	11
Computational Geometry	1247	5398	16
Set and String Problems	519	1898	9
Totals	5678	21127	75

Table 1: Hits by Major Section Index

2. The Stony Brook Algorithms Repository

The Stony Brook Algorithms Repository was developed in parallel with my book [6], *The Algorithm Design Manual*, and the structure of the repository mirrors the organization of my book. The world has been divided into a total of 75 fundamental algorithmic problems, partitioned among data structures, numerical algorithms, combinatorial algorithms, graph algorithms, hard problems, and computational geometry. See Table 6 or http://www.cs.sunysb.edu/~algorith for the list of 75 problems.

For each problem, the book poses questions to try to reveal the issues inherent in a proper formulation, and then tries to suggest the most pragmatic algorithm solution available. Where appropriate, relevant implementations are noted in the book, and collected on the Algorithms Repository, which has been mirrored on a CD-ROM included with the book. In total, we have identified a collection of 56 relevant algorithm implementations. Finding these codes required a substantial effort. Since many of these implementations proved applicable to more than one problem, the repository contains an average of three relevant implementations per problem.

Each problem page has a link to each relevant implementation page, as well as to pages associated with closely related problems. Each implementation page contains a link to the page associated with each problem to which it is applicable. Further, indexes contain links to implementations by programming language, subject area, and pictorial representation. Together these links enable the user to move easily through the site.

3. What are People Looking For?

Out of the almost quarter-million hits recorded on this site over the ten-week interval, 58289 of them were to primary html and shtml files. This latter count more accurately represents the number of mouse-clicks performed by users than the total hits, since most of the remaining hits are on image files associated with these pages. Therefore, we will limit further analysis to hits on these files.

Because user ID information is not logged on our WWW server, it is difficult to judge exactly how many different people accounted for these hits. Based on the roster of machines which accessed the site, I estimate that roughly 10,000 different people paid a visit during this 10 week study. Some fraction of hits came from webcrawler robots instead of human users, however I believe they had only a minor effect on our statistics. Observe that the least frequently clicked shtml file (containing the copyright notice for the site) was hit only 41 times versus 2752 hits for the most frequently accessed page (the front page). [2]

[2] By contrast, the page advertising my book was hit 1364 times, although I have no way of knowing whether any of them actually ordered it.

Programming Language	Index Hits	Implementations
C language	805	37
C++	929	11
Fortran	125	6
Lisp	99	1
Mathematica	104	3
Pascal	272	5
Totals	2334	63

Table 2: Hits by Programming Language Index

Most Popular Problems	Hits	Least Popular Problems	Hits
shortest-path	681	shape-similarity	156
traveling-salesman	665	factoring-integers	141
minimum-spanning-tree	652	independent-set	137
kd-trees	611	cryptography	136
nearest-neighbor	609	maintaining-arrangements	134
triangulations	600	text-compression	133
voronoi-diagrams	578	generating-subsets	133
convex-hull	538	set-packing	126
graph-data-structures	519	planar-drawing	120
sorting	485	median	118
string-matching	467	satisfiability	116
dictionaries	459	bandwidth	107
geometric-primitives	452	shortest-common-superstring	105
topological-sorting	424	feedback-set	83
suffix-trees	423	determinants	78

Table 3: Most and least popular algorithmic problems, by repository hits.

Table 1 reports the number of hits distributed among our highest level of classification – the seven major subfields of algorithms. Two different hit measures are reported for each subfield, first the number of hits to the menu of problems within the subfield, and second the total number of hits to individual problem pages within this subfield. Computational geometry proved to be the most popular subfield by both measures, although outweighed by the interest in graph problems split across two subtopics. Data structures recorded the highest "per-problem" interest, but I was surprised by the relative lack of enthusiasm for set and string algorithms.

Table 2 reports the number of hits distributed among the various programming language submenus. C++ seems to have supplanted C as the most popular programming language among developers, although there is clearly a lag in the size of the body of software written in C++. C remains the source language for over half the implementations available on the Algorithm Repository. User interest in Mathematica rivals that of Fortran, perhaps suggesting that computer algebra systems are becoming the language of choice for scientific computation. There was no submenu associated with Java, reflecting what was available when I built the repository. The total number of implementations in Table 2 is greater than 56 because seven codes are written in more than one language.

Table 3 reports the 15 most popular and least popular algorithmic problems, as measured by the

number of hits the associated pages received. Hit counts for all of the 75 problems appears in Table 6. Several observations can be drawn from this data:

- Although shortest path was the most popular of the algorithmic problems over the ten week time period, a preliminary study done two weeks earler showed traveling-salesman comfortably in the top spot (560 hits to 513). I attribute the late surge of interest in shortest path to coincide with the end of the academic year in the United States, and students from algorithms courses seeking an edge. Minimum spanning tree showed a similar surge in this time interval.
- Of the six data structure problems, only priority queues (number 17) and set union-find (number 37) failed to make the top 15 cut.
- People seem twice as interested in generating permutations than subsets (258 hits to 133 hits), presumably reflecting the perceived difficulty of the task.
- Surprisingly popular problems include topological sorting (number 14), suffix trees (number 15), the knapsack problem (number 19). These might reflect educational interest, although Table 5 shows that almost twice as many total .com hits were recorded than total .edu hits.
- Surprisingly unpopular problems include set cover (number 59), planar drawing (number 69), and satisfiability (number 71). Such obviously commercial problems as cryptography (number 64) and text compression (number 66) proved unpopular presumably because better WWW resources exist for these problems.

It is interesting to note that only 2752 hits occured to the front-page of the site, which means that most users never saw the main index of the site. This implies that most users initially entered the site through a keyword-oriented search engine, and gives credence to the notation that these hits measure problem interest more than just directionless wandering through the site.

4. What are They Finding?

The majority of visitors to the Algorithms Repository come seeking implementations of algorithms which solve the problem they are interested in. To help guide the user among the relevant implementations for each problem, I have rated each implementation from 1 (lowest) to 10 (highest), with my rating reflecting my opinion of the chances that an applied user will find that this implementation solves their problem.

My ratings are completely subjective, and in many cases were based on a perusal of the documentation instead of first-hand experience with the codes. Therefore, I cannot defend the correctness of my ratings on any strong objective basis. Still, I believe that they have proven useful in pointing people to the most relevant implementation. [3]

Table 7 records the number of hits received for each implementation, along with the problem for which it received the highest rating, as well its average rating across all problems. LEDA [2] received almost as many hits (2084) as the two following implementations, both associated with popular books [4] (1258) and [1] (994). The fourth most popular implementation was (surprisingly) Ranger [3] (846), an implementation of kd-trees. This reflects the enormous popularity of nearest-neighbor searching in higher dimensions, as well as the fact that I have not updated the list of implementations since the publication of the book in November. Arya and Mount's recently released ANN (http://www.cs.umd.edu/~mount/ANN/) would be a better choice. Note that these counts record the number of people who looked at the information page associated with each implementation. The actual number of ftps is unknown but presumably significantly lower.

[3] Any software developer who is dissatisfied with their ratings will perhaps be gratified to learn that my own *Combinatorica* [5] received the fourth lowest average score among the 56 rated implementations. Anybody who cannot better appreciate the merits of *Combinatorica* is clearly unfit to judge other people's software.

Most Needed Implementations	Rank by Mass	Hits	Δ	Least Needed Implementations	Rank by Mass	Hits	Δ
suffix-trees	57	15	-42	high-precision-arithmetic	16	29	13
bin-packing	74	34	-40	priority-queues	4	17	13
knapsack	59	19	-40	edge-coloring	43	57	14
kd-trees	42	4	-38	drawing-trees	37	53	16
eulerian-cycle	66	31	-35	maintaining-arrangements	47	64	17
polygon-partitioning	65	32	-33	matching	1	18	17
nearest-neighbor	36	5	-31	unconstrained-optimization	40	58	18
minkowski-sum	71	42	-29	satisfiability	50	70	20
simplifying-polygons	68	44	-24	dfs-bfs	9	30	21
motion-planning	73	55	-18	network-flow	2	23	21
traveling-salesman	19	2	-17	random-numbers	18	40	22
scheduling	64	48	-16	bandwidth	48	71	23
set-data-structures	53	37	-16	matrix-multiplication	25	49	24
edge-vertex-connectivity	60	45	-15	planar-drawing	41	68	27
thinning	62	47	-15	cryptography	33	63	30
graph-partition	52	39	-13	generating-graphs	12	46	34
string-matching	24	11	-13	fourier-transform	13	50	37
hamiltonian-cycle	32	21	-11	generating-subsets	28	66	38
set-cover	70	59	-11	generating-partitions	21	60	39
approximate-pattern-matching	34	24	-10	determinants	30	74	44

Table 4: Most needed and least needed implementations, based on program mass and hit ranks

domain	.com	.edu	.gov	.mil	.net	.org	$[0-9]*$	countries	totals
hits	15310	8421	266	341	5193	183	9149	19426	58289

Table 5: Hits by top level domain

Despite their shortcomings, I believe that these ratings provide a useful insight into the state of the art of combinatorial computing today. Hits per problem page measures the level of interest in a particular algorithmic problem. *Program mass*, the sum of the rankings of all implementations for a given problem, provides a measure of how much effort has been expended by the algorithm engineering community on the given problem. By comparing the ranks of each problem by program mass and the popularity, we can assess which problems are most (and least) in need of additional implementations.

Table 4 presents the results of such an analysis, showing the 20 most under (and over) implemented algorithmic problems. Suffix trees (rank 1) and kd-trees (rank 4) are the most needed data structure implementations, while the closely related problems of bin packing (rank 2) and knapsack (rank 3) are in the most need of algorithm implementations. There seems to be greater interest than activity in routing problems like Eulerian cycle/chinese postman (rank 5), traveling salesman (rank 11), and Hamiltonian cycle (rank 18). On the other hand, traditional algorithm engineering topics like matching (rank 61) and network flow (rank 65) have resulted in a relative abundance of codes for these problems.

5. Who is Looking?

By analyzing the domain names associated with each hit on the Algorithm Repository, we can see who is interested in algorithms. Table 5 records the number of hits by top-level domain. I believe that more hits were recorded by industrial users than educational ones, since the .com (15310) and .net (5193) domains together account for more than twice the number of .edu (8421) hits. Roughly one third of all hits came from users outside the United States, presumably similarly split between educational and industrial users.

It is interesting and amusing to see the distribution of hits by country code. No less than 84

nations visited the Algorithm Repository during this ten week interval, suggesting a much broader interest in algorithms than I would have thought. Hit count per nation is summarized in Table 8.

The most algorithmically inclined nation after the United States (presumably the source of most .com and .edu hits) was, not surprisingly, Germany (2340). The United Kingdom (1677), France (1445), and Spain (1054) each accounted for significantly more hits than Israel (315), Japan (709), and the Netherlands (590) – suggesting that the interest does not completely correlate with my perception of the amount of algorithmic research activity in these nations. Two of the largest producers of graduate students in computer science, China (39) and India (89), ranked surprisingly low in the number of hits despite the presence of substantial software industries. Presumably this reflects limited WWW access within these countries.

6. Conclusions

Analysis of hits to the Stony Brook Algorithm Repository provides interesting insights to the demand for algorithms technology, and the state of the art of available implementations. It would be interesting to repeat this analysis at regular intervals to see how the demand changes over time.

This most important conclusion of this work is that there is a demand for high quality implementations of algorithms for several important and interesting problems. I urge members of the algorithm engineering community to consider projects for problems on the left side of Table 4, for these represent the real open problems in the field. Indeed, I would be happy to add any results of this work to the Algorithm Repository for others to benefit from.

7. Acknowledgements

I would like to thank Ricky Bradley and Dario Vlah, who helped to build the software infrastructure which lies behind the Stony Brook Algorithm Repository.

References

[1] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, Wokingham, England, second edition, 1991.

[2] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[3] M. Murphy and S. Skiena. Ranger: A tool for nearest neighbor search in high dimensions. In *Proc. Ninth ACM Symposium on Computational Geometry*, pages 403–404, 1993.

[4] R. Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading MA, 1992.

[5] S. Skiena. *Implementing Discrete Mathematics.* Addison-Wesley, Redwood City, CA, 1990.

[6] S. Skiena. *The Algorithm Design Manual.* Springer-Verlag, New York, 1997.

		All Implementations		Best Implementation	
Problem	Hits	Impl. Count	Avg Score	Program Name	Rating
approximate-pattern-matching	298	3	6.3	agrep	10
bandwidth	107	2	7.5	toms	9
bin-packing	258	1	3.0	xtango	3
calendar-calculations	188	1	10.0	reingold	10
clique	248	3	5.3	dimacs	9
convex-hull	538	7	5.4	qhull	10
cryptography	136	3	5.3	pgp	10
determinants	78	4	4.5	linpack	8
dfs-bfs	268	7	3.9	LEDA	8
dictionaries	459	4	6.0	LEDA	10
drawing-graphs	275	3	7.0	graphed	9
drawing-trees	189	3	7.0	graphed	9
edge-coloring	169	4	4.5	stony	6
edge-vertex-connectivity	214	3	4.0	combinatorica	4
eulerian-cycle	262	2	3.0	combinatorica	3
feedback-set	83	1	4.0	graphbase	4
finite-state-minimization	290	4	5.7	grail	9
fourier-transform	197	5	5.0	fftpack	10
generating-graphs	208	5	6.8	graphbase	10
generating-partitions	163	5	6.6	wilf	8
generating-permutations	258	4	7.0	ruskey	8
generating-subsets	133	4	6.3	wilf	8
geometric-primitives	452	5	5.4	LEDA	8
graph-data-structures	519	6	6.7	LEDA	10
graph-isomorphism	251	2	6.5	nauty	10
graph-partition	232	2	6.0	link	8
hamiltonian-cycle	320	5	4.2	toms	6
high-precision-arithmetic	274	5	5.6	pari	9
independent-set	137	2	6.0	dimacs	7
intersection-detection	410	5	5.2	LEDA	7
kd-trees	611	3	4.0	ranger	8
knapsack	341	2	5.0	toms	6
linear-equations	218	3	6.6	lapack	10
linear-programming	317	5	4.4	lpsolve	9
longest-common-substring	175	2	5.0	cap	8
maintaining-arrangements	134	2	8.0	arrange	9
matching	363	10	5.2	goldberg	9
matrix-multiplication	200	5	5.0	linpack	7
median	118	2	5.0	handbook	6
minimum-spanning-tree	652	9	4.0	LEDA	6
minkowski-sum	219	1	4.0	eppstein	4
motion-planning	181	1	3.0	orourke	3
nearest-neighbor	609	4	5.2	ranger	7
network-flow	312	8	5.0	goldberg	10
planar-drawing	120	3	5.7	graphed	8
point-location	293	4	4.8	LEDA	7
polygon-partitioning	259	1	8.0	geompack	8
priority-queues	398	8	4.5	LEDA	9
random-numbers	230	6	4.8	simpack	7
range-search	289	4	4.8	LEDA	8
satisfiability	116	2	8.0	posit	8
scheduling	204	2	4.0	syslo	4
searching	235	3	5.6	handbook	7
set-cover	168	1	5.0	syslo	5
set-data-structures	241	3	4.3	LEDA	5
set-packing	126	1	5.0	syslo	5
shape-similarity	156	2	6.5	snns	7
shortest-common-superstring	105	1	8.0	cap	8
shortest-path	681	7	5.0	goldberg	9
simplifying-polygons	216	1	5.0	skeleton	5
sorting	485	7	4.9	moret	7
steiner-tree	190	2	7.5	salowe	8
string-matching	467	5	4.4	watson	7
suffix-trees	423	2	4.0	stony	6
text-compression	133	1	5.0	toms	5
thinning	206	1	9.0	skeleton	9
topological-sorting	424	6	3.5	LEDA	7
transitive-closure	195	2	4.0	LEDA	6
traveling-salesman	665	5	5.0	tsp	8
triangulations	600	8	5.9	triangle	9
unconstrained-optimization	168	3	6.3	toms	8
vertex-coloring	328	8	5.0	dimacs	7
vertex-cover	223	3	5.0	clique	6
voronoi-diagrams	578	6	5.3	fortune	9

Table 6: Hits by algorithmic problem, with implementation ratings

		Major Problem		All Problems	
Software	Hits	Problem Name	Rating	Count	Average
ASA	132	unconstrained-optimization	6	1	6.0
LEDA	2084	graph-data-structures	10	30	6.2
agrep	223	approximate-pattern-matching	10	1	10.0
arrange	87	maintaining-arrangements	9	3	7.3
bipm	127	matching	8	1	8.0
cap	119	shortest-common-superstring	8	2	8.0
clarkson	123	convex-hull	6	1	6.0
clique	154	clique	6	6	5.5
combinatorica	603	generating-graphs	8	28	4.0
culberson	133	vertex-coloring	6	2	5.0
dimacs	288	matching	9	10	5.6
eppstein	373	minkowski-sum	4	2	4.0
fftpack	107	fourier-transform	10	1	10.0
fortune	368	voronoi-diagrams	9	2	8.0
genocop	113	unconstrained-optimization	5	1	5.0
geolab	146	geometric-primitives	5	1	5.0
geompack	187	polygon-partitioning	8	2	8.0
goldberg	446	network-flow	10	3	9.3
grail	282	finite-state-minimization	9	1	9.0
graphbase	569	generating-graphs	10	17	4.4
graphed	398	drawing-graphs	9	7	6.4
handbook	994	dictionaries	8	12	4.9
htdig	107	text-compression	7	1	7.0
lapack	120	linear-equations	10	1	10.0
link	144	graph-partition	8	4	4.5
linpack	68	determinants	8	2	7.5
linprog	76	linear-programming	4	1	4.0
lpsolve	354	linear-programming	9	1	9.0
math	105	matrix-multiplication	6	1	6.0
moret	466	sorting	7	17	3.8
nauty	243	graph-isomorphism	10	1	10.0
north	209	drawing-graphs	7	2	7.0
orourke	598	geometric-primitives	6	8	4.4
pari	281	high-precision-arithmetic	9	2	9.0
pgp	44	cryptography	10	1	10.0
phylip	89	steiner-tree	7	1	7.0
posit	55	satisfiability	8	1	8.0
qhull	268	convex-hull	10	4	7.0
ranger	846	kd-trees	8	3	7.0
reingold	225	calendar-calculations	10	1	10.0
ruskey	206	generating-permutations	8	4	7.2
salowe	98	steiner-tree	8	1	8.0
sedgewick	1258	sorting	5	11	3.2
simpack	286	priority-queues	7	2	7.0
skeleton	187	thinning	9	2	7.0
snns	189	shape-similarity	7	1	7.0
stony	272	suffix-trees	6	3	6.0
syslo	506	set-cover	5	11	4.1
toms	546	bandwidth	9	24	5.0
triangle	232	triangulations	9	1	9.0
trick	193	vertex-coloring	7	2	5.5
tsp	331	traveling-salesman	8	1	8.0
turn	55	shape-similarity	6	1	6.0
watson	322	finite-state-minimization	8	2	7.5
wilf	259	generating-partitions	8	12	4.5
xtango	644	sorting	6	19	3.2

Table 7: Hits by implementation, with associated ratings

Count	Country	Code	Count	Country	Code
2340	Germany	de	50	Thailand	th
1692	Canada	ca	49	Venezuela	ve
1677	United Kingdom	uk	48	Soviet Union	su
1445	France	fr	48	Cyprus	cy
1054	Spain	es	47	Slovakia	sk
932	Australia	au	46	Yugoslavia	yu
842	Italy	it	46	Ukraine	ua
709	Japan	jp	44	Indonesia	id
591	Korea	kr	42	Kazakhstan	kz
590	Netherlands	nl	41	Turkey	tr
528	Sweden	se	39	China	cn
463	Finland	fi	32	Honduras	hn
423	Brazil	br	26	South Africa	za
417	Hong Kong	hk	23	Romania	ro
346	Norway	no	23	Philippines	ph
345	Belgium	be	21	Malta	mt
317	Switzerland	ch	20	Great Britain	gb
315	Israel	il	17	Lithuania	lt
302	Austria	at	17	Costa Rica	cr
294	Portugal	pt	12	Egypt	eg
282	Poland	pl	12	Dominican Republic	do
267	Slovenia	si	12	Armenia	am
238	Russia	ru	11	Trinidad and Tobago	tt
218	Singapore	sg	11	Jordan	jo
170	Denmark	dk	10	Iceland	is
162	Greece	gr	9	Uruguay	uy
159	Czech Republic	cz	9	Bahrain	bh
148	Chile	cl	7	Peru	pe
143	Mexico	mx	6	United Arab Emirates	ae
137	Taiwan	tw	5	Georgia	ge
125	Argentina	ar	4	Belize	bz
115	United States	us	3	Saudi Arabia	sa
115	Malaysia	my	3	Bolivia	bo
113	Hungary	hu	2	New Caledonia	nc
97	Columbia	co	2	Mauritius	mu
89	India	in	2	Moldova	md
85	Ireland	ie	2	Ecuador	ec
80	Croatia	hr	1	Pakistan	pk
77	Estonia	ee	1	Namibia	na
60	New Zealand	nz	1	Luxembourg	lu
58	Latvia	lv	1	Kuwait	kw
52	Bulgaria	bg	1	Barbados	bb

Table 8: Hits by Nation

AUTHOR INDEX