ALGORITHMS AND DATA STRUCTURES FOR DYNAMIC GRAPH PROBLEMS

Jeffrey Westbrook

CS-TR-229-89

October 1989

Algorithms and Data Structures for Dynamic Graph Problems

Jeffery R. Westbrook

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF COMPUTER SCIENCE

October 1989

Algorithms and Data Structures for Dynamic Graph Problems

Jeffery R. Westbrook

We give data structures, algorithms, and lower bounds for three problems on dynamic graph, i.e., graphs that are being modified on-line.

We consider the problem of maintaining the connected components of a graph undergoing edge insertions and backtracking edge deletions. We analyze several algorithms for the *set union with backtracking* problem, a variant of the classical disjoint set union (equivalence) problem in which an extra operation, *de-union* undoes the most recently performed union operation not yet undone. The most efficient such algorithms have an amortized running time of $O(\log n / \log\log n)$ per operation, where n is the total number of elements in all the sets. We prove that any separable pointer-based algorithm for the problem requires $\Omega(\log n / \log\log n)$ time per operation, thus showing that our upper bound an amortized time is tight. We use these fast algorithms to build an algorithm with the same resource bounds that maintains the connected components of an undirected graph undergoing edge insertion and backtracking edge deletion. By a simple reduction, the lower bounds for set union with backtracking can be applied to the connected components problem.

We examine the twin problems of maintaining the bridge-connected components or the biconnected components of a graph being changed by vertex and edge insertions. We give a basic algorithm that is suitable for both problems. With simple data structures, this algorithm runs in $O(n\log n+m)$ time, where n is the number of vertices and m is the number of operations. We develop a modified version of the dynamic trees of Sleator and Tarjan that is suitable for efficient recursive algorithms, and use it to reduce the running time of the algorithms for both problems to $O(m\alpha(m,n))$, where α is a functional inverse of Ackermann's function. This time bound is optimal. All of the algorithms use $O(n)$ space.

We investigate the problem of maintaining a minimum spanning forest and the connected components of an embedded, edge-weighted planar graph undergoing changes in edge weight as well as edge and vertex insertion and deletion. To implement the algorithms, we develop a data structure called an *edge-ordered dynamic tree*, which is a variant of the dynamic tree data structure of Sleator and Tarjan. Using this data structure, our algorithms run in $O(\log n)$ amortized time per operation and $O(n)$ space.

Acknowledgements

I am deeply indebted to my advisor, Robert E. Tarjan, with whom I did the work in this dissertation. The inspiration and guidance he gave me are irreplaceable. I am also grateful to my dissertation readers, Bernard Chazelle and Robert Sedgewick, for their time and comments. Most of the figures in this dissertation were drawn using routines written by Robert Sedgewick.

My parents and family have provided me with constant comfort and care in my years of graduate school. I thank my father, David Rex Westbrook, for setting an example to which I could aspire.

I could easily fill a second volume in acknowledging the roommates, officemates, fellow students, and friends who made my years at Princeton enjoyable and who helped me survive to complete this dissertation. I am grateful in particular to Heather Booth, who led me through my first years as a computer scientist; to Tama Hasson, who led me through my last years; and to Paul Lucier, who led me all over the place during every year. Thanks also go to Cynthia Hill, whose wisdom and compassion eased some difficult times.

Finally, I would like to thank Sharon Rodgers of the Department of Computer Science for being so helpful during all stages of writing this dissertation.

Table of Contents

1. Introduction to Dynamic Graph Problems

Let $G = (V, E)$ be an undirected graph, possibly with weighted edges or vertices. A *dynamic graph problem* on G is a problem of maintaining information about some property or properties of G while performing a sequence of modifications to G. Modifications may include the insertion or deletion of vertices, the insertion or deletion of edges, and, if the edges or vertices are weighted, changes in their weights. Queries about the property being maintained are intermixed with the modifications. The sequence of queries and modifications must be performed on-line, that is, each operation must be completed before the next one is performed, and the future sequence of operations is not known by the algorithm that solves the problem.

In this dissertation we give data structures, algorithms, and lower bounds for three dynamic graph problems. In chapters two and three we consider the problem of maintaining the connected components of a graph undergoing edge insertions and backtracking edge deletions. We begin in chapter two by giving an amortized analysis of algorithms for the *set union with backtracking* problem, a variant of the classical disjoint set union (equivalence) problem in which an extra operation, *de-union* undoes the most recently performed union operation not yet undone. In chapter three, we use the results of chapter two to solve dynamic connectivity with backtracking. In chapter four we examine the twin problems of maintaining the bridge-connected components or the biconnected components of a graph being changed by vertex and edge insertions. In chapter five we investigate the problem of maintaining a minimum spanning forest and the connected components of an embedded, edge-weighted planar graph undergoing changes in edge weight as well as edge and vertex insertion and deletion.

An increasing amount of research is being devoted to the study of on-line graph algorithms, a study that has wide applications to interconnection networks, CAD/CAM, and distributed computing. In addition to those considered here, the list of dynamic graph problems that have been studied includes maintaining transitive closure

[12,13,14,28], and maintaining shortest paths [8,19,28]. The study of dynamic planar graphs is particularly active. Dynamic planar graphs arise naturally in many applications, such as VLSI design, graphics, and vision. They also occur in many two-dimensional computational geometry problems, and are used by algorithms that build planar subdivisions such as Voronoi diagrams. Some of the popularity of planar graph problems is probably also a result of the fact that they seem more amenable to fast solutions than similar problems on general graphs. The introductions to each chapter of this dissertation provide further discussion of the motivation and related results for our problems.

In developing our on-line graph algorithms, our paradigm is the use of dynamic data structures and amortized analysis. Many examples of the uses of dynamic data structures may be found in Tarjan's monograph [25] and Mehlhorn's three-volume set [15]. Another general source is the monograph of Overmars [17]. Recent work has centered on such topics as fractional cascading [2,3,16], persistence [1,4,6,20], fast heaps [7,9,10], dynamic perfect hashing [5], and data structures for planar graphs [18,23]. In this dissertation we use red-black (balanced binary) trees [11], splay trees [22], and the dynamic tree data structure of Sleator and Tarjan [21,22], perhaps best known for its use in network flow problems. In the algorithms presented here we introduce two new data structures based on the Sleator/Tarjan dynamic tree, discuss data structures for set union with backtracking, and give faily delicate amortized analyses of the running times of these data structures. Of amortized analysis we shall say more below, but for the moment we note that the amortized cost of an operation is defined as the total cost of a worst-case sequence of operations divided by the number of operations.

Set union with backtracking has been studied by Mannila and Ukkonen, who proposed a way to modify standard set union algorithms to handle de-union operations. We analyze several algorithms based on their approach. The most efficient such algorithms have an amortized running time of $O(\log n / \log\log n)$ per operation, where n is the total number of elements in all the sets. These algorithms use $O(n \log n)$ space, but the space usage can be reduced to $O(n)$ by a simple change. We prove that any separable pointer-based algorithm for the problem requires $\Omega(\log n / \log\log n)$ time per operation, thus showing that our upper bound an amortized time is tight. We use these fast algorithms for set union with backtracking to build an algorithm for maintaining the connected components of an undirected graph undergoing edge insertion and backtracking edge

deletion. This algorithm runs in $O(\log n / \log\log n)$ amortized time per operation and uses $O(n)$ space, where n is the number of vertices. By a simple reduction, the lower bounds for set union with backtracking can be applied to the connected components problem.

To solve the twin problems of maintaining the bridge-connected components or the biconnected components of a dynamic undirected graph, we begin by giving a basic algorithm that is suitable for both problems. With simple data structures, this algorithm runs in $O(n\log n+m)$ time, where n is the number of vertices and m is the number of operations. We develop a modified version of the dynamic trees of Sleator and Tarjan that is suitable for efficient recursive algorithms, and use it to reduce the running time of the algorithms for both problems to $O(m\alpha(m,n))$, where α is a functional inverse of Ackermann's function [26]. This time bound is optimal. All of the algorithms use $O(n)$ space.

To implement the algorithms for maintaining a minimum spanning forest of an embedded planar graph, we develop a data structure called an *edge-ordered dynamic tree*, which is a variant of the dynamic tree data structure of Sleator and Tarjan. Using this data structure, our algorithms run in $O(\log n)$ amortized time per operation and $O(n)$ space.

The techniques of amortized analysis are tailor-made for analyzing algorithms based on dynamic data structures. A general overview of amortized analysis is given in the survey paper by Tarjan [24], and many examples can be found in [15,25]. Given some cost metric such as time or space, an amortized analysis attempts to show that while some operations in an on-line sequence may have high cost, these operations must be balanced by other operations with low cost. In this dissertation we use two types of amortized analysis: *charge-sharing* analysis and *potential function* analysis. In charge-sharing, a bound on the amortized cost per operation is shown by giving a scheme to share the cost of any single operation among itself and other operations that have occurred previously. Then it is shown that under the sharing scheme, no one operation is charged more than the amortized bound. This technique is used in chapter 2.

In a potential function argument, we define a mapping Φ from configurations of the data structure to the real numbers. We let Φ_i denote the value of the potential function after the ith operation. If t_i is the actual cost of the ith operation then the amortized cost

a_i

of the ith operation is given by $a_i = t_i + \Delta\Phi$, where $\Delta\Phi = \Phi_i - \Phi_{i-1}$. If $\tilde{a}$ is an upper bound on a_i for all i, we have the following bound for $T(m)$, the time to perform a sequence of m operations:

$$T(m) = \sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - \Phi_i + \Phi_{i-1}) \leq \Phi_0 - \Phi_m + m\,\tilde{a}$$

We will always define Φ such that $\Phi_0 = 0$ and $\Phi_m => 0$, which implies that $T(m) \leq m\,\tilde{a}$. Using the definition of amortized cost, we can show that if some function f has amortized cost a, and some operation Q performs b amortized calls to to f, then Q has amortized cost $a \cdot b$.

We present remarks, conclusions, and open problems at the end of the chapter to which they are relevant, rather than in a final chapter. The results of chapter two have appeared in [27], and the material of chapter four has been presented at the *Capital City Conference on Combinatorics and Theoretical Computer Science*, 1989. Chapter five is joint work with Roberto Tamassia of Brown University.

References for chapter 1.

[1] B. Chazelle, "How to search in history," *Information and Control* 64 (1985), 77-99.

[2] B. Chazelle, "A functional approach to data structures and its use in multidimensional searching," *SIAM J. Comput.* 17 (1988), 427-462.

[3] B. Chazelle and L. J. Guibas, "Fractional cascading: I. a data structuring technique; II. applications," *Algorithmica* 1 (1986), 133-191.

[4] R. Cole, "Searching and storing similar lists," *J. Algorithms* 7 (1986), 202-220.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H.Rohnert, and R. E. Tarjan, "Dynamic perfect hashing: upper and lower bounds," *Proc. 29th IEEE Symp. of Foundations of Computer Science* (1988), 524-531.

[6] J. Driscoll, H. N. Gabow, N. Sarnak and R. E. Tarjan, "Making data structures persistent," *Proc. 18th Symp. of Foundations of Computer Science* (1986), 109-121.

[7] J. Driscoll, H. N. Gabow, R. Shrairman and R. E. Tarjan, "Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation," *Comm. ACM* 31 (1988), 1343-1354.

[8] S. Even and H. Gazit, "Updating distances in dynamic graphs," *Methods of Operations Research* 49 (1985), 371-387.

[9] M. L. Fredman, R. Sedgewick, D. D. Sleator and R. E. Tarjan, "The pairing heap: a new form of self-adjusting heap," *Algorithmica* 1 (1986), 111-129.

[10] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. Assoc. Comput. Mach.* 34 (1987), 596-615.

[11] L. J. Guibas and R. Sedgewick, "A Dichromatic framework for balanced trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978), 8-21.

[12] T. Ibaraki and N. Katoh, "On-line computation of transitive closure for graphs," *Inform. Process. Lett.* 16 (1983), 95-97.

[13] G. F. Italiano, A. Marchetti Spaccamela, U. Nanni, "Dynamic data structures for series parallel digraphs," *Proc. Workshop on Algorithms and Data Structures (WADS 89)* (Lecture Notes in Computer Science), Springer-Verlag, New York (1989).

[14] J. A. La Poutré and J. van Leeuwen, transitive reduction of graphs," *Proc. International Workshop on Graph-Theoretic Concepts in Computer Science (WG 87)* (Lecture Notes in Computer Science), Springer-Verlag, New York (1988), 106-120.

[15] K. Mehlhorn, *Data Structures and Algorithms* (3 volumes). Springer-Verlag, New York (1984).

[16] K. Mehlhorn and S. Näher, "Dynamic fractional cascading," *Proc. ACM Symp. on Computational Geometry,* (1985).

[17] M. Overmars, *The Design of Dynamic Data Structures*. (Lecture Notes in Computer Science), Springer-Verlag , New York (1983).

[18] F. P. Preparata and R. Tamassia, "Fully dynamic techniques for point location and transitive closure in planar structures," *Proc. 29th IEEE Symposium on Foundations of Computer Science,* (1988), 558-567.

[19] H. Rohnert, "A dynamization of the all-pairs least cost path problem," *Proc. 2nd Annual Symp. on Theoretical Aspects of Computer Science* (Lecture Notes in Computer Science), Springer-Verlag, New York (1985), 279-286.

[20] N. Sarnak and R. E. Tarjan, "Planar point location using persistent search trees," *Comm. ACM* 29 (1986), 669-679.

[21] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Comput. Sys. Sci.* 26 (1983), 362-391.

[22] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc. Comput. Mach.* 32 (1985), 652-686.

[23] R. Tamassia, "A dynamic data structure for planar graph embedding" *Proc. 15th Int. Conf. on Automata, Languages, and Programming* (1988), 576-590.

[24] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Alg. Disc. Meth.* 6 (1985), 306-318.

[25] R. E. Tarjan, *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia (1983).

[26] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. Assoc. Comput. Mach.* 31 (1984), 245-281.

[27] J. Westbrook and R. E. Tarjan, "Amortized analysis of algorithms for set union with backtracking," *SIAM J. Comput.* 18 (1989), pp. 1-11.

[28] D. Yellin, "A dynamic transitive closure algorithm," Research Report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, (1988).

2. *Amortized Analysis of Algorithms for Set Union with Backtracking*

1. Introduction.

The classical disjoint set union problem is that of maintaining a collection of disjoint sets whose union is $U = \{1,2,...,n\}$ subject to a sequence of m intermixed operations of the following two kinds:

find (x): Return the name of the set currently containing element x.

union (A,B): Combine the sets named A and B into a new set, named A.

The initial collection consists of n singleton sets, $\{1\}, \{2\}, \ldots, \{n\}$. The name of initial set $\{i\}$ is i. For simplicity in stating bounds we assume $m = \Omega(n)$. This assumption does not significantly affect any of the results, and it holds in most applications.

Several fast algorithms for this problem are known [11,14]. They all combine a rooted tree set representation with some form of path compaction. The fastest such algorithms run in $O(\alpha(m,n))$ amortized time per operation, where α is a functional inverse of Ackermann's function [11,14]. No better bound is possible for any pointer-based algorithm that uses a separable set representation [12]. For the special case of the problem in which the subsequence of union operations is known in advance, the use of address arithmetic techniques leads to an algorithm with an amortized time bound of $O(1)$ per operation [3].

Mannila and Ukkonen [8] studied a generalization of the set union problem called *set union with backtracking*, in which the following third kind of operation is allowed:

de-union: Undo the most recently performed union operation that has not yet been undone.

This problem arises in Prolog interpreter memory management [7]. Mannila and Ukkonen showed how to extend path compaction techniques to handle backtracking. They posed the question of determining the inherent complexity of the problem, and they claimed an $O(\log \log n)$ amortized time bound per operation for one algorithm based on their approach. Unfortunately, their upper bound argument is faulty.

In this chapter we derive upper and lower bounds on the amortized efficiency of algorithms for set union with backtracking. We show that several algorithms based on the approach of Mannila and Ukkonen run in $O(\log n/\log \log n)$ amortized time per operation. These algorithms use $O(n \log n)$ space, but the space can be reduced to $O(n)$ by a simple change. We also show that any pointer-based algorithm that uses a separable set representation requires $\Omega(\log n/\log \log n)$ amortized time per operation. All the algorithms we analyze are subject to this lower bound. Improving the upper bound of $O(\log n/\log \log n)$, if it is possible, will require the use of either a nonseparable pointer-based data structure or of address arithmetic techniques.

The remainder of this chapter consists of four sections. In Section 2 we review six algorithms for set union without backtracking and discuss how to extend them to handle backtracking. In Section 3 we derive upper bounds for the amortized running times of these algorithms. In Section 4 we derive a lower bound on amortized time for all separable pointer-based algorithms for the problem. Section 5 contains concluding remarks and open problems.

2. Algorithms for Set Union with Backtracking

The known efficient algorithms for set union without backtracking [11,14] use a collection of disjoint rooted trees to represent the sets. The elements in each set are the nodes of a tree, whose root contains the set name. Each element contains a pointer to its parent. Associated with each set name is a pointer to the root of the tree representing the set. Each initial (singleton) set is represented by a one-node tree.

To perform union (A,B), we make the tree root containing B point to the root containing A, or alternatively make the root containing A point to the root containing B and swap the names A and B between their respective elements. (This not only moves the name A to the right place but also makes undoing the union easy, as we shall see below.) The choice between these two alternatives is governed by a *union rule*. To perform

find (x), we follow the path of pointers from element x to the root of the tree containing x and return the set name stored there. In addition, we apply a *compaction rule*, which modifies pointers along the path from x to the root so that they point to nodes farther along the path.

We shall consider the following possibilities for the union and compaction rules:

Union Rules:

Union by weight: Store with each tree root the number of elements in its tree. When doing a union, make the root of the smaller tree point to the root of the larger, breaking a tie arbitrarily.

Union by rank: Store with each tree root a nonnegative integer called its *rank*. The rank of each initial tree root is zero. When doing a union, make the root of smaller rank point to the root of larger rank. In the case of a tie, make either root point to the other, and increase the rank of the root of the new tree by one.

Compaction Rules (see Figure 1):

Compression: After a find, make every element along the find path point to the tree root.

Splitting: After a find, make every element along the find path point to its grandparent, if it has one.

Halving: After a find, make every other element along the find path (the first, third, etc.) point to its grandparent, if it has one.

The two choices of a union rule and three choices of a compaction rule give six possible set union algorithms. Each of these has an amortized running time of $O(\alpha(m,n))$ per operation [14].

We shall describe two ways to extend these and similar algorithms to handle de-union operations. The first method is the one proposed by Mannila and Ukkonen; the second is a slight variant.

We call a union operation that has been done but not yet undone *live*. We denote a pointer from a node x to a node y by (x,y). Suppose that we perform finds without doing any compaction. Then performing de-unions is easy: to undo a set union we merely

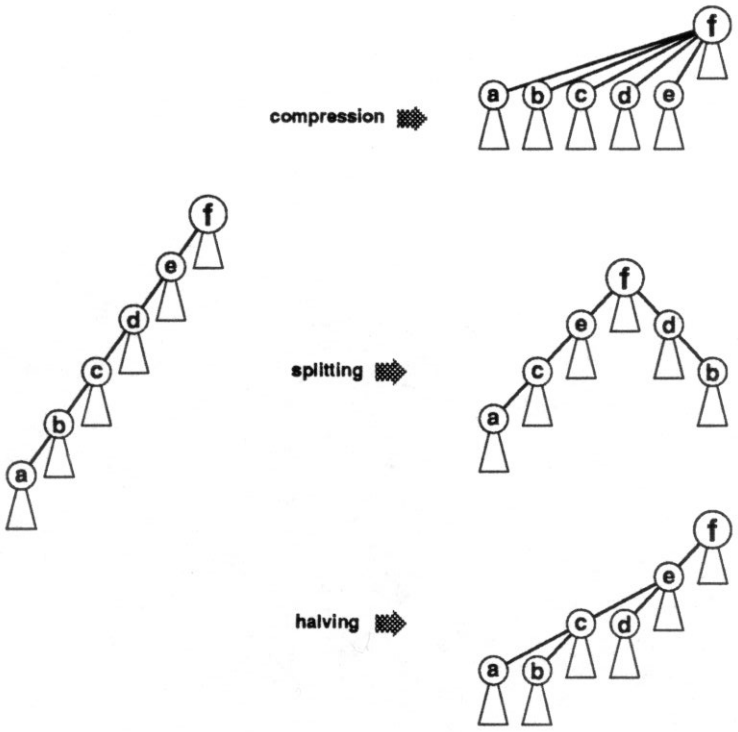


Figure 1: Path compression, path splitting, and path halving. The element found is ''a''.

make null the pointer added to the data structure by the union. To facilitate this, we maintain a *union stack*, which contains the tree roots made nonroots by live unions. To perform a de-union, we pop the top element on the union stack and make the corresponding parent pointer null.

This method works with either of the two union rules. Some bookkeeping is needed to maintain set names and sizes or ranks. Each entry on the union stack must contain not only an element but also a bit that indicates whether the corresponding union operation swapped set names. If union by rank is used, each such entry must contain a second bit that indicates whether the union operation incremented the rank of the new tree root. The time to maintain set names and sizes or ranks is $O(1)$ per union or de-union; thus each union or de-union takes $O(1)$ time, worst-case. Either union rule guarantees a maximum tree depth of $O(\log n)$ [14]; thus the worst-case time per find is $O(\log n)$. The space needed by the data structure is $O(n)$.

Mannila and Ukkonen's goal was to reduce the time per find, possibly at the cost of increasing the time per union or de-union and increasing the space. They developed the following method for allowing compaction in the presence of de-unions. Let us call the forest maintained by the noncompacting algorithm described above the *reference forest*. In the compacting method, each element x has an associated *pointer stack* $P(x)$, which contains the outgoing pointers that have been created during the course of the algorithm but have not yet been destroyed. The bottommost pointer on this stack is one created by a union. Such a pointer is called a *union pointer*. The other pointers on the stack are ones created by compaction. They are called *find pointers*. Each pointer (x,y) of either type is such that y is a proper ancestor of x in the reference forest.

Each pointer has an *associated union operation*, which is the one whose undoing would invalidate the pointer. To be more precise, for a pointer (x,y) the associated union operation is the one that created the pointer (z,y) such that z is a child of y and an ancestor of x in the reference forest. As a special case of this definition, if (x,y) is a union pointer then $z = x$ and the associated union operation is the one that created (x,y). A pointer is *live* if its associated union is live.

Unions and de-unions are performed as in the noncompacting method. Compactions are performed as in the set union algorithm without backtracking, except that each new pointer (x,y) is pushed onto $P(x)$ instead of replacing the old pointer leaving x. When following a find path from an element x, the algorithm pops dead pointers from the top of $P(x)$ until $P(x)$ is empty or a live pointer is on top. In the former case, x is the root of its tree; in the latter case, the live pointer is followed.

This algorithm requires a way to determine whether a pointer is live or dead. For this purpose the algorithm assigns each union operation a distinct number as it is performed. Each entry on the union stack contains the number of the corresponding union. Each pointer on a pointer stack contains the number of the associated union and a pointer to the position on the union stack where the entry for this union was made. This information can be computed in $O(1)$ time for any pointer (x,y) when it is created. If (x,y) is a union pointer, the information is computed as part of the union. If (x,y) is a find pointer, then the last pointer on the find path from x to y when (x,y) was created has the same associated union as (x,y) and has stored with it the needed information. To test whether a pointer is live or dead, it is merely necessary to access the union stack entry whose

position is recorded with the pointer and test first, if the entry is still on the stack, and second, whether its union number is the same as that stored with the pointer. If so, the pointer is live; if not, dead.

The implementation of de-union must be changed slightly, to preserve the invariant that in every pointer stack all the dead pointers are on top. To perform a de-union, the algorithm pops the top entry on the union stack. Let x be the element in this entry. The algorithm pops $P(x)$ until it contains only one pointer, which is the union pointer created by the union that is to be undone. The algorithm restores the set names and sizes or ranks as necessary, and pops the last pointer from $P(x)$. Because of the compaction, the state of the data structure after a de-union will not in general be the same as its state before the corresponding union.

We call this method the *lazy method* since it destroys dead pointers in a lazy fashion. Either of the union rules and any of the compaction rules can be used with the method. The total running time is proportional to m plus the total number of pointers created. (With any of the compaction rules, a compaction of a find path containing $k \geq 2$ pointers results in the creation of $\Omega(k)$ pointers, $k-1$ in the case of compression or splitting and $\lfloor k/2 \rfloor$ in the case of halving.)

An alternative to the lazy method is the *eager method*, which pops pointers from pointer stacks as soon as they become dead. To make this popping possible, each union stack entry must contain a list of the pointers whose associated union is the one corresponding to the entry. When a union stack entry is popped, all the pointers on its list are popped from their respective pointer stacks as well. Each such pointer will be on top of its stack when it is to be popped. To represent such a pointer, say (x,y), in a union stack entry, it suffices to store x. With this method, numbering the union operations is unnecessary, as is popping pointer stacks during finds.

The time required by the eager method for any sequence of operations is only a constant factor greater than that required by the lazy method, since both methods create the same pointers but the eager method destroys them earlier. With either union rule, the eager method uses $O(n \log n)$ space in the worst case, since the maximum tree depth is $O(\log n)$ and all pointers on any pointer stack point to distinct elements. (From bottom to top, the pointers on $P(x)$ point to shallower and shallower ancestors of x.)

The lazy method also has an $O(n \log n)$ space bound [4]. For any node x, consider the top printer on $P(x)$, which is to a node, say y. Even if the pointer from x to y is currently dead, it must once have been live, and all pointers currently on $P(x)$ point to distinct nodes on the tree path from x to y as it existed when the pointer from x to y was live. Thus there can be only $O(\log n)$ such pointers. The total number of pointers is therefore $O(n \log n)$. The total number of numbers needed to distinguish relevant union operations is also $O(n \log n)$, which implies that the total space needed is $O(n \log n)$, as claimed.

The choice between the lazy and eager methods is not clear-cut. As we shall see at the end of Section 3, a small change in the compaction rules reduces the space needed by either method to $O(n)$.

3. Upper Bounds on Amortized Time

The analysis to follow applies to both the lazy method and the eager method. Ignoring the choice between lazy and eager pointer deletion, there are six versions of the algorithm, depending on the choice of a union rule and a compaction rule.

As a first step on the analysis, we note that compression with either union rule is no better in the amortized sense than doing no compaction at all, i.e. the amortized time per operation is $\Omega(\log n)$. The following class of examples shows this. For any k, form a tree of 2^k elements by doing unions on pairs of elements, then on pairs of pairs, and so on. This produces a tree called a *binomial tree* B_k, whose depth is k. (See Figure 2.) Repeat the following three operations any number of times: do a find on the deepest element in B_k, undo the most recent union, and redo the union. Each find creates $k-1$ pointers, which are all immediately made dead by the subsequent de-union. Thus the amortized time per operation is $\Omega(k) = \Omega(\log n)$.

Both splitting and halving perform better; each has an $O(\log n / \log \log n)$ amortized bound per operation, in combination with either union rule. To prove this, we need a definition. For an element x, let *size (x)* be the number of descendants of x (including itself) in the reference forest. The *logarithmic size* of x, $lgs(x)$, is $\lfloor lg\ size(x) \rfloor$ *.

* For any x, $lg\, x = \log_2 x$.

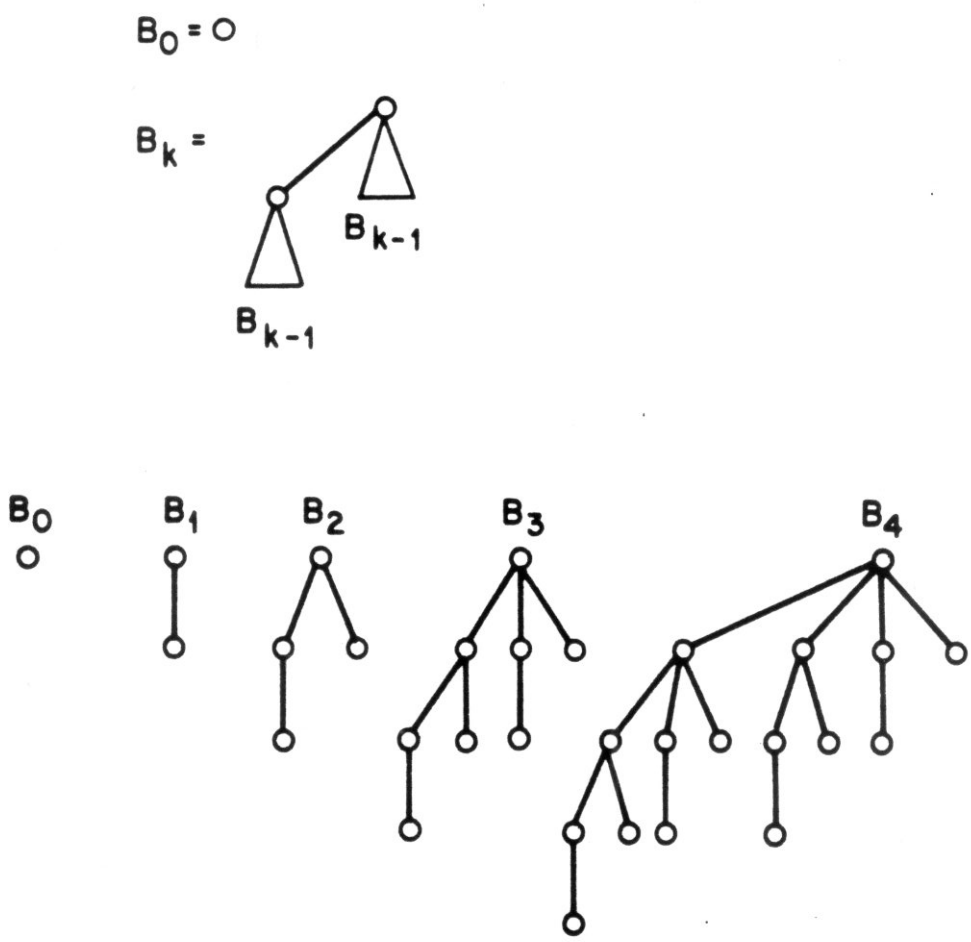


Figure 2: Binomial trees.

We need the following lemma concerning logarithmic sizes when union by weight is used.

Lemma 1 [11].

Suppose union by weight is used. If node v is the parent of node w in the reference forest, then $lgs(w) < lgs(v)$. Any node has logarithmic size between 0 and $lg\ n$ (inclusive).

Proof. When a node v becomes the parent of another node w, $size(w) \leq 2\ size(v)$ by the union by weight rule. Later unions can only increase $size(v)$ and cannot increase $size(w)$ (unless the union linking v and w is undone). The lemma follows. □

Theorem 1.

Union by weight in combination with either splitting or halving gives an algorithm for set union with backtracking running in $O(\log n/\log\log n)$ amortized time per

operation.

Proof. We shall charge the pointer creations during the algorithm to unions and finds in such a way that each operation is charged for $O(\log n/\log\log n)$ pointer creations. For an arbitrary positive constant $c < 1$, we call a pointer (x,y) *short* if $lgs(y) - lgs(x) \leq c\ lg\ lg\ n$ and *long* otherwise. (The logarithmic sizes in this definition are measured at the time (x,y) is created.) We charge the creation of a pointer (x,y) as follows:

(i) If y is a tree root, charge the operation (union or find) that created (x,y).

(ii) If y is not a tree root and (x,y) is long, charge the find that created (x,y).

(iii) If y is not a tree root and (x,y) is short, charge the union that most recently made y a non-root.

A find with splitting creates two new paths of pointers, and a find with halving creates one new path of pointers. Thus $O(1)$ pointers are charged to each operation by (i). The number of long pointers along any path can be estimated as follows. For any long pointer (x,y), $lgs(y) - lgs(x) > c\ lg\ lg\ n$. Logarithmic sizes strictly increase along any path and are between 0 and $lg\ n$ by Lemma 1. Thus if there are k long pointers on a path, $lg\ n \geq k\ c\ lg\ lg\ n$, which implies $k \leq lg\ n/(c\ lg\ lg\ n)$. Thus a find with either splitting or halving can create only $O(\log n/\log\log n)$ long pointers, which means that $O(\log n/\log\log n)$ pointers are charged to each find by (ii).

It remains for us to bound the number of pointers charged by (iii). Consider a union operation that makes an element x a child of another element y. Let I be the time interval during which pointers are charged by (iii) to this union. During I, the sizes, and hence the logarithmic sizes, of all descendants of x remain constant. Interval I ends with the undoing of the union.

For each descendant w of x, at most one pointer (w,x) can be charged by (iii) to the union, since the creation of another such pointer charged by (iii) cannot occur at least until x again becomes a root and then becomes a nonroot, which can only happen after the end of I. Thus the number of pointers charged by (iii) to the union is at most one per descendant w of x such that $lgs(x) - lgs(w) \leq c\ lg\ lg\ n$.

Since logarithmic sizes strictly increase along tree paths, any two elements u and v with $lgs(u) = lgs(v)$ must be unrelated, i.e. their sets of descendants are disjoint. This means that the number of descendants w of x with $lgs(w) = i$ is at most $size(x)/2^i \le 2^{lgs(x)+1-i}$, and the number of descendants w of x with $lgs(x) - lgs(w) \le c\ lg\ lg\ n$ is at most

$$\sum_{i=lgs(x)-\lfloor c\ lg\ lgn\rfloor}^{lgs(x)} 2^{lgs(x)+1-i} \le 2^{\lfloor c\ lg\ lg\ n\rfloor+2} = O((\log n)^c) = O(\log n/\log\log n),$$

since $c < 1$. Thus there are $O(\log n/\log\log n)$ pointers charged to the union by (iii). □

The same result holds if union by rank is used instead of union by weight, but in this case the proof becomes a little more complicated because logarithmic sizes need not strictly increase along tree paths. We deal with this by slightly changing the definition of short and long pointers. We need the following lemma.

Lemma 2 [14].

Suppose union by rank is used. If node v is the parent of node w in the reference forest, then $0 \le lgs(w) \le lgs(v) \le lg\ n$ and $0 \le rank(w) < rank(v) \le lg\ n$.

Proof. The first group of inequalities is immediate. The definition of union by rank implies $rank(w) < rank(v)$. A proof by induction on the rank of v shows that $size(v) \ge 2^{rank(v)}$, which implies that $rank(v) \le lg\ n$. □

Theorem 2.

Union by rank in combination with either splitting or halving gives an algorithm for set union with backtracking running in $O(\log n/\log\log n)$ amortized time per operation.

Proof. We define a pointer (x,y) to be *short* if $\max\{lgs(y)-lgs(x), rank(y)-rank(x)\} \le c\ lg\ lg\ n$ and *long* otherwise, where $c < 1$ is a positive constant. We charge the creation of pointers to unions and finds exactly as in the proof of Theorem 1 (rules (i), (ii), and (iii)). The number of pointers charged by rule (i) is $O(1)$ per union or find, exactly as in the proof of Theorem 1. A long pointer (x,y) satisfies at least one of the inequalities $lgs(x) - lgs(x) > clg\ lg\ n$ and $rank(y) - rank(x) > c\ lg\ lg\ n$. Along any tree path only $O(\log n/\log\log n)$ long pointers can satisfy the former inequality and only

$O(\log n/\log\log n)$ long pointers can satisfy the latter, by Lemma 2. It follows that only $O(\log n/\log\log n)$ pointers can be charged per find by rule (ii).

To count short pointers, we make one more definition. For a non-root element x, let $p(x)$ be the parent of x in the reference forest. A non-root x is *good* if $lgs(x) < lgs(p(x))$ and *bad* otherwise, i.e. if $lgs(x) = lgs(p(x))$. The definition of *lgs* implies that any element can have at most one bad child. The bad elements thus form paths called *bad paths* of length $O(\log n)$; all elements on a bad path have the same logarithmic size. We call the element of largest rank on a bad path the *head* of the path. The head of a bad path is a bad element whose parent is either a good element or a tree root.

Consider a union operation that makes an element x a child of an element y. We count short pointers charged to this union as follows:

(1) *Short pointers leading from good elements.* If v and w are good elements such that $lgs(v) = lgs(w)$, then v and w are unrelated in the reference forest, i.e., they have disjoint sets of descendants. The analysis that yielded the count of short pointers in the proof of Theorem 1 applies to the good elements here to yield a bound of $O((\log n)^c) = O(\log n/\log\log n)$ short pointers leading from good elements that are charged to the union by (iii).

(2) *Short pointers leading from bad elements.* Consider the number of bad paths from which short pointers can lead to x. The head of such a path is an element w such that $p(w)$ is either good or a tree root, and $lgs(x) - lgs(w) \le c\ lg\ lg\ n$. Heads of different bad paths have different parents. The analysis that counts short pointers in the proof of Theorem 1 yields an $O((\log n)^c)$ bound on the number of bad paths from which short pointers can lead to x. Along such a bad path, rank strictly increases, and the definition of shortness implies that only the $c\ lg\ lg\ n$ elements of largest rank along the path can have short pointers leading to x. The total number of short pointers leading from bad nodes that are charged to the union by (iii) is thus $O(c\log\log n\,(\log n)^c) = O(\log n/\log\log n)$. □

We conclude this section by discussing how to reduce the space bound for both the lazy method and the eager method to $O(n)$. This is accomplished by making the following simple changes in the compaction rules. If union by size is used, the compaction of a find path is begun at the first node along the path whose size is at least $lg\ n$. If union by

rank is used, the compaction of a find path is begun at the first node whose rank is at least $lg\ lg\ n$. With this modification, only $O(n/\log n)$ nodes have find pointers leaving them, and the total number of pointers in the data structure at any time is $O(n)$. The analysis in Theorems 1 and 2 remains valid, except that there is an additional time per find of $O(\log \log n)$ to account for the initial, noncompacted part of each find path.

4. A General Lower Bound on Amortized Time

We shall prove that the bound in Theorems 1 and 2 is best possible for a large class of algorithms for set union with backtracking,. Our computational model is the *pointer machine* [5,6,10,12] with an added assumption about the data structure called *separability*. Related results follow. Tarjan [12] derived an amortized bound in this model for the set union problem without backtracking. Blum [1] derived a worst-case-per-operation lower bound for the same problem. Mehlhorn, Näher, and Alt [9] derived an amortized lower bound for a related problem. Their result does not require separability.

The algorithms to which our lower bound applies are called *separable pointer algorithms*. Such an algorithm uses a linked data structure that can be regarded as a directed graph, with each pointer represented by an edge. The algorithm solves the set union with backtracking problem according to the following rules:

(i) The operations are presented on-line, i.e. each operation must be completed before the next one is known.

(ii) Each set element is a node of the data structure. There can be any number of additional nodes.

(iii) (Separability). After any operation, the data structure can be partitioned into node-disjoint subgraphs, one corresponding to each currently existing set and containing all the elements in the set. The name of the set occurs in exactly one node in the subgraph. *No edge leads from one subgraph to another.*

(iv) The cost of an operation *find*(x) is the length (number of edges) of the shortest path from x to the node that holds the name of the set containing x. This length is measured at the beginning of the find, i.e. before the algorithm changes the structure as specified in (v).

(v) During any *find*, *union*, or *de-union* operation, the algorithm can add edges to the data structure at a cost of one per edge, delete edges at a cost of zero, and move, add, or delete set names at a cost of zero. The only restriction is that separability must hold after each operation.

The eager method of Section 2 obeys rules (i)-(v). This is also true of the lazy method, if we regard pointers as disappearing from the model data structure as soon as they become dead. This does not affect the performance of the algorithm in the model, since once a pointer becomes dead it is never followed.

Theorem 3.

For any n, any $m = \Omega(n)$, and any separable pointer algorithm, there is a sequence of m find, union, and de-union operations whose cost is $\Omega(m \log n/\log\log n)$.

Proof. We shall prove the theorem for n of the form 2^{2^k} for some $k \geq 1$ and for $m \geq 4n$. The result follows for all n and $m = \Omega(n)$ by padding the expensive problem instances constructed below with extra singleton sets on which no operations take place and with extra finds.

In estimating the cost of a sequence of operations, we shall charge the cost of adding an edge to the data structure to the deletion of the edge. Since this postpones the cost, it cannot increase the total cost of a sequence.

We construct an expensive sequence as follows. The first $n-1$ operations are unions that build a set of size n by combining singletons in pairs, pairs in pairs, and so on. The remaining operations occur in groups, each group containing between 1 and $2n-2$ operations. Each group begins and ends with all the elements in one set. We obtain a group of operations by applying the appropriate one of the following two cases (if both apply, either may be selected). Let $b = \lfloor lg\ n/(2\ lg\ lg\ n) \rfloor$.

(1) If some element in the (only) set is at distance at least b away from the set name, do a find on this element.

(2) If some sequence of l de-unions will force the deletion of $l\,b$ edges from the data structure (to maintain separability), do these de-unions. Then do the corresponding unions in the reverse order, restoring the initial set of size n.

We claim that if there is only one set, formed by repeated pairing, then case (1) or case (2) must apply. If this is true, we can obtain an expensive sequence of operations by generating successive groups of operations until more than $m - 2n + 2$ operations have occurred, and then padding the sequence with enough additional finds to make a total of m operations. The cost of such a sequence is at least $(m-3n+3)b = \Omega(m \log n/\log\log n)$.

It remains to prove the claim. Suppose case (2) does not apply. We shall show that case (1) does. Let $f = (lg\ n)^2$. For $0 \le i \le lg\ n/lg\ f$ we define a partition P_i of the nodes of the data structure by

$P_i = \{X \mid X$ is the collection of nodes in the subgraph corresponding to one of the sets that would be formed by doing $f^i - 1$ de-unions$\}$

Observe that $|P_i| = f^i$. Also $f^{lgn/lgf} = n$, so P_i is defined for $i \le lg\ n/lg\ f$. In particular P_b is defined, since $b = \lfloor lg\ n/(2\ lg\ lg\ n) \rfloor = \lfloor lg\ n/lg\ f \rfloor$.

For $0 \le i \le lg\ n/lg\ f$, we define the collection D_i of *deep sets* in P_i as follows:

$D_i = \{X \in P_i \mid$ all elements in X are at distance at least i from the name of the single set$\}$.

Let $d_i = |D_i|$. We shall show that $d_b > 0$, which implies the existence of an element at distance at least b away from the name of the single set; hence case (1) applies.

Let k_i be the number of edges that lead from one set in P_i to another. We have $k_i \le bf^i$, since otherwise performance of $f^i - 1$ de-unions would force the deletion of bf^i edges from the data structure, and case (2) would apply.

Now we derive a recursive bound on d_i. We have $d_1 = f - 1$, since only one of the f sets in P_1 can contain the only set name. We claim that $d_{i+1} \ge f\,d_i - k_i$. To verify the claim, let us consider D_i. Since $n = 2^{2^k}$ and the union structure of the only set forms a binomial tree, each set X in D_i consists of f sets in P_{i+1}, all of whose elements are at distance at least i from the name of the only set. For an element $x \in X$ to be at distance exactly i from the set name, some edge must lead from x to a set in P_i other than X; otherwise X would not be in D_i. There are k_i such edges. Each such edge can eliminate one set in P_{i+1} from being in D_{i+1}. But this leaves $f\,d_i - k_i$ sets in D_{i+1}, namely the $f\,d_i$ sets into which the sets in D_i divide, minus at most k_i eliminated by edges between different sets in P_i. That is, $d_{i+1} \ge fd_i - k_i$, as claimed.

Applying the bound $k_i \le bf^i$ gives $d_{i+1} \ge fd_i - bf^i$. Using $d_1 = f - 1$, a proof by induction shows that $d_i \ge f^{i-1}(f-(i-1)b-1)$.

We wish to show that $d_b > 0$. This is true provided that $(f-(b-1)b-1) > 0$. But $f = (lg\ n)^2$ and $b = \lfloor lg\ n/(2lg\ lg\ n) \rfloor$, giving $(f-(b-1)b-1) = (f-b^2+b-1)$ $\ge \frac{3}{4}(lg\ n)^2 > 0$, since we are assuming $n \ge 4$, which implies $b^2 \le (lg\ n)^2/4$ and $b \ge 1$. Thus $d_b > 0$, which implies that some element is at distance at least b from the set name, i.e. case (1) applies. □

5. Remarks

Our bound of $\Theta(\log n/\log\log n)$ on the amortized time per operation in the set union problem with backtracking is the same as Blum's worst-case bound per operation in the set union problem without backtracking [1]. Perhaps this is not a coincidence. Our lower bound proof resembles his. Furthermore the data structure he uses to establish his upper bound can easily be extended to handle de-union operations; the worst-case bound per operation remains $O(\log n/\log\log n)$ and the space needed is $O(n)$.

The compaction methods have the advantage over Blum's method that as the ratio of finds to unions and de-unions increases the amortized time per find decreases. The precise result is that if the ratio of finds to unions and de-unions in the operation sequence is γ and the amortized time per union and de-union is defined to be $\Theta(1)$, then the amortized time per find is $\Theta(\log n/(\max\{1, \log(\gamma\log n)\}))$. This bound is valid for any value of γ, and it holds for splitting or halving with either union rule, and it is the best bound possible for any separable pointer algorithm. This can be proved using straightforward extensions of the arguments in sections 3 and 4. The space bound can be made $O(n)$ by an extension of the idea proposed at the end of section 3. If the de-union operations occur in bursts, the time per operation decreases futher, but we have not attempted to analyze this situation.

Perhaps the most interesting open problem is whether the lower bound in section 4 can be extended to nonseparable pointer algorithms. (In place of separability, we require that the out-degree of every node in the data structure be constant.) We conjecture that the bound in Theorem 3 holds for such algorithms. The techniques of Mehlhorn, Näher, and Alt [9] suggest an approach to this question, which might yield at least an

$\Omega(\log \log n)$ bound if not an $\Omega(\log n / \log \log n)$ bound on the amortized time.

Recently, Fredman and Saks [2] have given an $\Omega(\alpha(m,n))$ bound on the amortized cost per operation in the cell probe model of Yao [15]. It is also possible to show an $\Omega(\log n / \log\log n)$ bound on the worst-case cost per operation [M. Fredman, private communication, 1989]. In this powerful and general model, memory is organized into cells, each of which can hold $\log n$ bits. In answering a query, a cell-probe algorithm is allowed to randomly access cells based on the information gathered from previous probes. The separable pointer and cell probe machines are related as follows: if the number of bits in a separable pointer machine node is bounded by $\beta(n)$, then the separable pointer machine can be simulated by a cell probe machine, with running time increasing by a factor of $\beta(n)/\log n$. We conjecture that the bound in Theorem 3 holds for cell probe algorithms.

References for chapter 2.

[1] N. Blum, "On the single-operation worst-case time complexity of the disjoint set union problem," *SIAM J. Comput.* 15 (1986), 1021-1024.

[2] M. L. Fredman and Michael E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proc. 21st ACM Symp. on Theory of Computing*, Seattle, Washington (May 1989), pp. 345-354.

[3] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *J. Comput. Sys. Sci.* 30 (1985), 209-221.

[4] G. Gambiosi, G. F. Italiano, and M. Talamo, "Getting back to the past in the union-find problem," *5th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science* 294, Springer-Verlag, Berlin, 1988, 8-17.

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[6] A. N. Kolmogorov, "On the notion of algorithm," *Uspehi Mat. Nauk.* 8 (1953), 175-176.

[7] H. Mannila and E. Ukkonen, "On the complexity of unification sequences," *Third International Conference on Logic Programming*, July 14-18, 1986, *Lecture Notes in Computer Science* 225, Springer-Verlag, New York, 1986, 122-133.

[8] H. Mannila and E. Ukkonen, "The set union problem with backtracking," *Proc. Thirteenth International Colloquium on Automata, Languages, and Programming* (ICALP 86), Rennes, France, July 15-19, 1986, *Lecture Notes in Computer Science* 226, Springer-Verlag, New York, 1986, 236-243.

[9] K. Mehlhorn, S. Näher, and H. Alt, "A lower bound for the complexity of the union-split-find problem," *SIAM J. Comput.* 17 (1988), 1093-1102.

[10] A Schönhage, "Storage modification machines," *SIAM J. Comput.* 9 (1980), 490-508.

[11] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. Assoc. Comput. Mach.* 22 (1975), 215-225.

[12] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintian disjoint sets," *J. Comput. Sys. Sci.* 18 (1979), 110-127.

[13] R. E. Tarjan, "Amortized comutational complexity," *SIAM J. Alg. Disc. Meth.* 6 (1985), 306-318.

[14] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. Assoc. Comput. Mach.* 31 (1984), 245-281.

[15] A. C. Yao, "Should tables be sorted?" *J. Assoc. Comput. Mach.* 28 (1981), 615-628.

3. *Maintaining Connected Components with Backtracking*

Perhaps the most fundamental equivalence relation on the constituents of an undirected graph $G=(V,E)$ is defined by its connected components. Let $V_1, V_2, \cdots, V_k$ be the partition of V such that two vertices are in the same class if and only if there is a path connecting them, and let $E_1, E_2, \cdots, E_k$ be the partition of E such that an edge is in E_i if and only if its endpoints are in V_i. The subgraphs $G_i=(V_i, E_i)$, $1 \le i \le k$, form the connected components, abbreviated *components*, of G.

The problem of finding the components of a fixed graph G is well-understood. Let n be the number of vertices and m the number of edges. Hopcroft and Tarjan [6] and Tarjan [11] give sequential algorithms based on depth-first search that run in time $O(n+m)$. Logarithmic-time parallel algorithms for finding components, bridge-blocks, and blocks are given in references [1], [8], and [12] (see also the survey paper [7]).

The problem of answering queries about edge and vertex membership in the components of a dynamic graph has been addressed in references [2,3,5,9]. Even and Shiloach [3] consider the component problem for a graph undergoing edge deletions. They give algorithms with constant query time, $O(n \log n)$ total update time in the case that G is a tree or forest, and $O(mn)$ update time for general G, where m and n are the numbers of edges and vertices, respectively, in the initial graph. Reif [9] gives an algorithm for the same problem that runs in time $O(n\,g + n \log n)$ when given an initial graph embedded in a surface of genus g. Frederickson [5] gives an algorithm that performs queries in constant time and edge insertions and deletions in time $O(\sqrt{m_i})$, where m_i is the number of edges in G at the time of the i^{th} update. Even and Shiloach also observe that if only edge insertions are allowed, the component problem can be solved by straightforward application of a fast disjoint set union algorithm.

In this chapter we consider the problem of maintaining the components of a dynamic graph. We assume that the graph is initially null (although the problem is essentially equivalent if we begin with a given set of vertices), and perform an

intermixed sequence of the following operations:

make vertex (): Return a new vertex v that forms a singleton component labeled v.

find component (v): Return the name of the component containing vertex $v \in V$.

insert edge (u,v): Insert an edge between vertices u and v. If the edge combines two components named A and B into a new component, name it A.

delete edge: Remove from the graph the most recently inserted edge that has not yet been removed.

To solve this problem, the partition of the vertices given by the connected components is explicitly maintained, using one of the four fast algorithms for set union with backtracking analyzed in the previous chapter. A *make vertex* () operation causes the creation of a new set containing only the new vertex v. The new set is named v. At any time, a *find component* (v) query is answered by returning the result of *find* (v).

To facilitate *delete edge*, we maintain an edge stack, whose records contain an integer counter. To implement *insert edge* (u,v), we compare the labels returned by *find component* (u) and *find component* (v). If the labels differ, then u and v belong to different components, and these components will be combined by the insertion of edge $\{u,v\}$. This operation is called a *component link*. We perform a union of the two sets containing u and v, and push a new record on the edge stack. The counter of the new record is initialized to zero.

On the other hand, if u and v belong to the same component, then the insertion of edge $\{u,v\}$ has no effect on the components. We simply increment the counter in the top record on the edge stack. Thus, this counter records the number of non-combining edge insertions that have taken place since the most recent component link. To perform *delete edge*, the counter in the top record is examined. If the counter is zero, the top record is popped off the stack and a de-union is executed. Otherwise, the counter is simply decremented.

Let n be the maximum number of vertices. Since G is connected after $n-1$ component links, the edge stack contains at most $n-1$ records. Each edge stack push or pop takes $O(1)$ time. Combining these observations with the upper bounds for set union with backtracking, we find that the above algorithm runs in $O(\log n / \log\log n)$ amortized time

per operation, and $O(n)$ space.

Note that we do not explicitly maintain the edge sets for each component. If an edge is known to be incident to vertex v, however, then the component containing the edge can be found by executing *find component* (v). The edge partition can be explicitly maintained with a second set union data structure, but at the cost of increasing the space to $O(m)$. Other variants of the problem are possible. For example, we may use the disjoint set union algorithm to maintain additional information about the components, such as size or edge of maximum weight, with no loss in efficiency.

The construction of the algorithm can be used in reverse to give a simple reduction from the disjoint set union problem to the dynamic connected components problem. Each set element is represented by a distinct vertex, and the connected components of the graph correspond to the disjoint sets. The query *find* (v) is answered by executing *find component* (v). A canonical vertex is associated with each set name; *union* (A,B) is implemented by the insertion of an edge between the canonical vertices for A and B.

Rules (i) through (v) of chapter two, section 4, can be appropriately adapted to define a class of separable pointer algorithms for the dynamic connected components problem. The separability rule (rule iii) now refers to connected components of the graph. That is, we require that the linked data structure can be partitioned into subgraphs corresponding to the connected components of the graph being represented, and that no edge lead between these subgraphs. Given any separable pointer algorithm for dynamic components with backtracking, the above reduction gives a separable pointer algorithm for disjoint set union with backtracking. Hence, for any n, any $m = \Omega(n)$, and any separable pointer algorithm, there is a sequence of m *find component*, *insert edge*, and *delete edge* operations whose cost is $\Omega(m \log n / \log\log n)$.

We note that for the dynamic component problem in which only edge insertions are allowed, the above reduction from disjoint set union allows us to apply both the separable pointer machine lower bound of Tarjan [10] and the cell probe model lower bound of Fredman and Saks [4]. Thus the upper bound of $O(\alpha(m,n))$ amortized time per operation for this version of the dynamic component problem is tight.

References for Chapter 3

[1] B. Awerbuch and Y. Shiloach, "New connectivity and MSF algorithms for shuffle-exchange network and PRAM," *IEEE Trans. on Computers* C-36 (1987), pp. 1258-1263.

[2] G. A. Cheston, "Incremental algorithms in graph theory," Tech. Rep. No. 91 (PhD. Diss.), Dept. of Computer Science, University of Toronto, (1976).

[3] S. Even and Y. Shiloach, "On-line edge deletion," *J. Assoc. Comp. Mach.* 28 (1981), pp. 1-4.

[4] M. L. Fredman and M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proc. 21st ACM Symposium on Theory of Computing,* Seattle, Washington (May 1989), pp. 345-354.

[5] G. N. Frederickson, "On-line updating of minimum spanning trees," *SIAM J. Computing* 14 (1985), pp. 781-798.

[6] J. Hopcroft and R. E. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Comm. ACM* 16 (1973), 372-378.

[7] R. Karp and V. Ramachandran, "Parallel Algorithms for Shared Memory Machines," Tech. Rep. No. CSD 88/408, Dept. of Computer Science, U.C. Berkeley (1988). (To appear in *Handbook of Theoretical Computer Science,* North-Holland.)

[8] G. L. Miller and V. Ramachandran, "A new graph triconnectivity algorithm and its parallelization," *Proc. 19th Annual ACM Symp. on Theory of Computing* (1987), pp. 335-344.

[9] J. H. Reif, "A topological approach to dynamic graph connectivity," *Inform. Process. Lett.* 25 (1987), pp. 65-70.

[10] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. Sys. Sci.* 18 (1979), pp. 110-127

[11] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. Computing* 1 (1972), pp. 146-160.

[12] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Computing* 14 (1985), pp. 862-874.

4. *Maintaining Bridge-Connected and Biconnected Components*

1. Introduction

In this chapter we consider a pair of natural and important equivalence relations on the constituents of an undirected graph $G=(V,E)$: its bridge-connected and biconnected components. Let $E_1, E_2, \cdots, E_k$ be the partition of E such that two edges are in the same part of the partition if and only if they are contained in a common simple cycle, and let V_i, $1 \le i \le k$, be the set of vertices that are endpoints of the edges in E_i. The subgraphs $G_i = (V_i, E_i)$ form the biconnected components, or *blocks*, of G. A vertex appearing in more than one block is called an *articulation point*, and its removal disconnects G. There is either no block or one block containing any given pair of vertices. An edge contained in no cycle is in a block by itself. Such an edge is called a *bridge*, and its removal disconnects the graph. The bridge-connected components, or *bridge-blocks,* of G are the components of the graph formed by deleting all the bridges. The bridge-blocks partition V into equivalence classes such that two vertices are in the same class if and only if there is a (not necessarily simple) cycle of G containing both of them. Figure 1 shows an undirected graph along with its blocks and bridge-blocks.

The problems of finding the blocks, and bridge-blocks of a fixed graph are well-understood. Hopcroft and Tarjan [9] and Tarjan [19] give sequential algorithms that run in time $O(n+m)$ where $n = |V|$ and $m = |E|$. Logarithmic-time parallel algorithms for finding components, bridge-blocks, and blocks are given in references [1] and [22] (see also the survey paper [11]).

In this chapter we study the problems of answering queries about the blocks or bridge-blocks of a dynamic graph. We allow two on-line graph update operations to be performed on an initially null graph G:

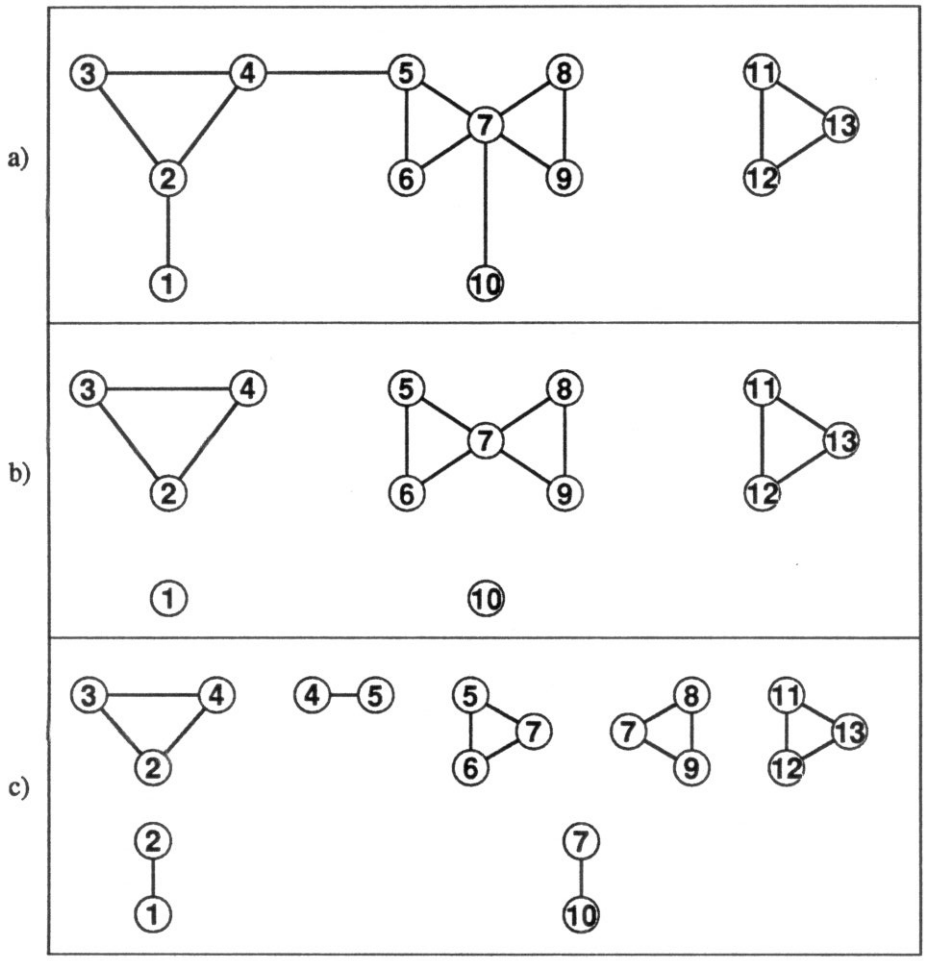


Figure 1: a) Undirected graph *G*. b) Bridge-blocks of *G*. c) Blocks of *G*. Multiply-appearing vertices are articulation points.

make vertex (*A*): Add a new vertex with no incident edges to *G*. Label by "*A*" the bridge-block formed by the new vertex. Return the name of the new vertex; the calling program must use this name to refer to the new vertex in subsequent operations. (The name is actually a pointer into the data structure maintained by the algorithm.)

insert edge (*u*,*v*,*A*): Insert a new edge between the vertices named *u* and *v*. Label by "*A*" any new bridge-block or block that results from the edge insertion.

In the bridge-block problem we allow the following query:

find block (u): Return the label of the bridge-block containing the vertex named u.

Similarly, in the block problem we allow the following query:

find block (u,v): Return the label of the block, if any, containing the pair of vertices $\{u,v\}$.

We also consider a restricted variant of the problem in which we are given an initially connected graph $G_0 = (V_0,E_0)$. We allow $O(|E_0|)$ preprocessing time, and then process on-line a sequence of intermixed queries and edge insertions. In this variant, the *make vertex* operation is not allowed. Our algorithms do not explicitly maintain the edge set, but if the endpoints of an edge are known, then *find block* can be used to determine which block contains the edge.

The block and bridge-block problems are natural problems to consider in the general study of on-line graph algorithms. They appear as subproblems of other on-line graph problems, such as incremental planarity testing [4]. The incremental planarity testing problem is to maintain a representation of a planar graph as edges are being added, and to determine the first edge addition that makes the graph nonplanar. Maintaining the blocks of a dynamic graph also arises in the implementation of efficient search strategies for logic programming [Graeme Port, private communication, 1988]. As we saw in chapter three, a substantial amount of research has been done on various dynamic connectivity problems, but we know of no previous algorithms for the block or bridge-block problems that run in sublinear time per operation.

This chapter is organized as follows. In Section 2 we develop a simple algorithm for the bridge-block problem that runs in $O(n\log n+m)$ total time and uses $O(n)$ space, where n is the number of vertices added to the initially null graph and m is the number of operations. In Section 3 we give a similar algorithm for the block problem that also runs in $O(n\log n+m)$ time and $O(n)$ space. In Sections 4-7 we decrease the total running time of the bridge-block and block algorithms to $O(m\alpha(m,n))$. To achieve this bound we introduce link/condense trees, a modified version of the dynamic trees of Sleator and Tarjan [14,15], and apply a fairly delicate analysis. The link/condense trees support condensing of adjacent nodes, and two such trees can be linked together in amortized time $O(\log k)$, where k is the size of the smaller tree. Section 4 describes the data structure as it applies to the bridge-block problem and Section 5 contains the amortized analysis of the

link/condense tree operations. Section 6 describes the modifications needed to apply link/condense trees to the block problem, and Section 7 contains additional analysis. Finally, Section 8 contains a simple reduction from the disjoint set union problem to the block and bridge-block problems. This allows us to apply the known lower bounds for set union to these two problems. Our results are summarized in the following table:

	Initially connected G	Initially null G
Bridge-blocks	$O(m\alpha(m,n))$	$\Theta(m\alpha(m,n))$
Blocks	$\Theta(m\alpha(m,n))$	$\Theta(m\alpha(m,n))$

The $O(\alpha(m,n))$ upper bound for the on-line bridge-block and block problems is somewhat surprising, since both these problems differ fundamentally from the on-line connected component problem that shares this bound. In the latter, a single edge insertion can combine at most two components into one. A single edge insertion, however, can create a cycle in the graph that might combine as many as $\Theta(n)$ blocks or bridge-blocks into one. This fact seems to make the maintenance of blocks and bridge-blocks under both edge insertion and edge deletion quite difficult. By simply alternating edge insertions with edge deletions, we can generate a sequence of operations that at every step changes the number of blocks in the graph by $\Theta(n)$.

To conclude this section, we note that although *find block* as defined above returns only a label, the algorithms presented below can be extended to return other information about the blocks or bridge-blocks, such as an edge or vertex of maximum weight, with no loss in efficiency. In general, we can maintain any data that can be updated in constant time when two blocks or bridge-blocks are combined into one. If we are willing to increase the space used to $O(m)$, we can also list the edges or vertices in a block or bridge-block in time $O(\alpha(m,n) + k)$, where k is the number of items listed.

2. Maintaining Bridge-Blocks On-Line

The bridge-blocks and bridges of a connected graph have a natural tree structure that we call the *bridge-block tree*. The collection of bridge-block trees given by the components of a graph $G = (V,E)$ is called the *bridge-block forest*, or BBF. The nodes in a bridge-block tree are of two types: square nodes, which represent the vertices of G;

and round nodes, which represent the bridge-blocks. If r is a round node then *label*(r) denotes the label of the corresponding bridge-block. Two round nodes are connected by a tree edge whenever a bridge connects the corresponding two bridge-blocks. If vertex v is contained in bridge-block A then the square node that represents v is connected by a tree edge to the round node that represents A. Every square node is a leaf. Each bridge-block tree is rooted at an arbitrarily selected round node. We denote by *parent*(x) the parent of tree node x. Given the BBF for G, the query *find block*(u) can be answered by computing *label*(*parent*(u)). (In general, we let the vertex name, here u, refer to both the vertex in the graph and to its square node representative in the bridge-block tree.) Figure 2a shows the BBF for the graph of Figure 1.

The effect of an *insert edge*(u,v,A) operation on the bridge-block forest depends on whether u and v are in the same bridge-block, different bridge-blocks but the same component, or different components. In the first case the bridge-block structure of the graph, and consequently the BBF, is unaffected. In the remaining two cases, however, the bridge-block structure is affected in opposite ways.

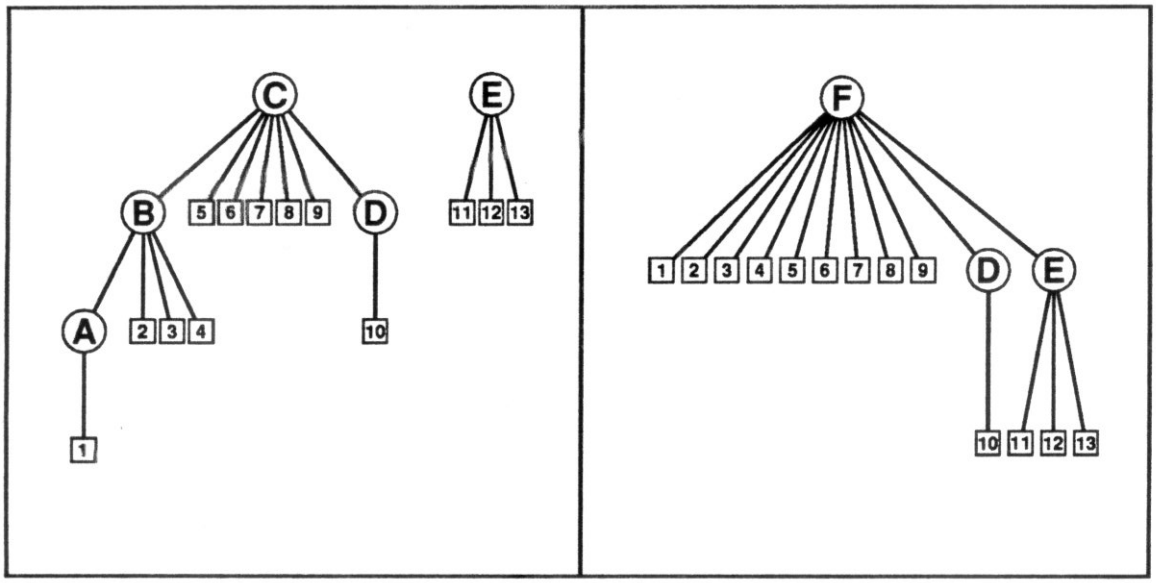
If u and v are in different components of the graph then the inserted edge connects the two components by creating a bridge between them. One of the two corresponding bridge-block trees, say the tree containing u, is rerooted at *parent*(u), and then *parent*(u) is made a child of *parent*(v). This process, called a *component link*, can occur at most $n-1$ times, after which the graph is connected.

On the other hand, if u and v are in the same component but different bridge-blocks, the inserted edge creates a new cycle that reduces the number of bridge-blocks in the component. The round nodes on the path connecting u and v in the corresponding tree must be replaced by a single round node. Every node previously adjacent to one of the round nodes on the path becomes adjacent to the new single round node. This process is called *path condensation*. After at most $n-1$ condensations, the graph has only a single bridge-block. Figure 2b gives examples of both cases of edge insertion.

To implement the bridge-block operations, the BBF is explicitly maintained with a data structure that supports the following functions.

Maketree (A : label or null value):

Create a new tree consisting of a single node. If A is null, make a square node; otherwise, make a round node labeled A; Return a pointer to the new node.



(a) (b)

Figure 2: a) Bridge-block forest of G. b) BBF after performing *insert edge* $(1,5,F)$ and *insert edge* $(9,11,E)$

Link $(x,y$:square or round nodes):

Link two trees together by first making x the root of its tree (this is called an *eversion*), and then making x a child of y.

Findpath $(u,v$: square nodes):

Find the tree path P between but not including square nodes u and v. If u and v are the same node, return *parent* (u).

Condensepath $(P$: *path*; A : label):

Perform path condensation on P and label by A the the resultant single node $\tilde{r}$.

With these functions, we implement the bridge-block operations as follows:

make vertex (A):

Let u = *Maketree* (null). Perform *Link* $(u$,*Maketree* $(A))$. Return u (to be used by the calling program as the name of the new vertex).

find block (u):

Return *label* (*Findpath* (u,u)).

insert edge (u,v,A):

(1) If u and v are in the same component then execute *Condensepath* (*Findpath* $(u,v),A$) and terminate.

(2) If u and v are in different components, determine which is in the smaller component. Assume u is. Let $x = Findpath\,(u,u)$ and $y = Findpath\,(v,v)$.

(3) Execute *Link* (x,y).

In the *insert edge* procedure we use an on-line component maintenance subroutine to determine if two vertices are in the same component of G and to determine the size of a component. This subroutine is a straightforward application of a fast disjoint set union algorithm [21]. Appropriate calls to the update functions of this subroutine must be made when making a new vertex or performing a component link.

The tree data structure is built using *condensible nodes*. A condensible node x consists of a block of storage, $N(x)$, containing an arbitrary but fixed collection of fields, and a set of subnodes, $S(x)$. The subnode sets are maintained with a fast disjoint set union algorithm [21]. The name of set $S(x)$ is simply $N(x)$. A condensible node is initialized with one subnode. To make a pointer p to node x, we store in p the name of some subnode $s \in S(x)$. Given such a pointer p, a *pointer step* consists of accessing $N(x)$ by performing *find* (p). Two nodes x and y can be condensed into a single node z by the following procedure: create a new storage block $N(z)$; let $S(z) = union(S(x),S(y))$; update appropriately the fields of $N(z)$ using the fields of $N(x)$ and $N(y)$; and discard $N(x)$ and $N(y)$. The union of the two subnode sets suffices to make all pointers to x and y become pointers to z. Note that condensation destroys x and y. If a data structure initially contains n condensible nodes, then any sequence of m pointer steps and condensations runs in worst-case time $O(m\alpha(m,n))$, and the data structure uses $O(n)$ space.

For each vertex v in the graph we store a pointer to its condensible node representative in the data structure. For a node x, N(x) contains a parent pointer and label field. If x is a square node, the label field is null; if x is a round node, the label field holds a bridge-block label. For any node x, *parent* (x) is computed by a pointer step using the parent pointer. The four tree functions take as arguments pointers to condensible nodes.

To perform *Link* (x,y), the tree containing x is everted by walking up from x to the tree root, reversing the direction of all parent pointers along the path. The number of pointer steps is the initial depth of x, which is at most the size of the tree. After the evert,

a pointer to y is stored in the parent field of x. Since x and y are passed in the form of pointers, this is actually implemented by storing the value of y in the parent field of the storage block returned by *find* (x). This requires one additional pointer step.

The *Findpath* (u,v) algorithm proceeds by walking up the tree simultaneously from u and from v in lock-step, until the paths from u and v intersect at their nearest common ancestor. The path P is returned as a list of nodes (not in order along the path), with the nearest common ancestor at the end. The number of pointer steps required is at most $2|P|$. Let z be the parent of the nearest common ancestor. To perform *Condensepath* (P,A), P is condensed into a single node $\tilde{r}$. Node $\tilde{r}$ is labeled by A and made a child of z. The number of node condensations is $|P| - 1$.

Lemma 1:

In any sequence of operations there are $O(n)$ pointer steps during condensing edge insertions.

Proof. Suppose a path P is being condensed. To generate P, $O(|P|)$ pointer steps are required, but $|P| - 1$ round nodes are condensed. After $n-1$ condensations the graph is bridge-connected. □

Lemma 2:

The total number of pointer steps during Link operations is $O(n\log n)$.

Proof. The number of square nodes in a bridge-block tree is at least the number of round nodes. Hence the cost of an evert in a tree of k square nodes is $\Theta(k)$. Since we always evert the smaller tree, we arrive at the following recursive upper bound on the total number of pointer steps, $T(n)$, needed to combine n components into one, where c is a sufficiently large constant.

$$T(1) = 0$$

$$T(n) \leq \max_{1 \leq k \leq n/2} \{T(k) + T(n-k) + ck\}$$

It is well-known that this recurrence has the solution $T(n) = O(n \log n)$. □

Theorem 1:

Given an initially null graph G_0, a sequence of $m = \Omega(n)$ *find block*, *make vertex*, and *insert edge* operations can be processed in $O(n\log n+m)$ time and $O(n)$ space,

where n is the number of vertices inserted.

Proof. First, we note that the total time spent in the on-line components subroutine is $O(m\alpha(m,n)) = O(n \log n + m)$. Next, we bound the total number of pointer steps and condensations that occur during processing of an input sequence. There is at least one pointer step per bridge-block operation. Let k be the number of pointer steps and node condensations occurring during path condensation and component linking. From lemmas 1 and 2, $k = O(n \log n)$. The amortized cost of a pointer step or node condensation is $O(\alpha(k+m,n))$. Thus the total running time is $O((k+m)\alpha(k+m,n))$. Since $\alpha(m,n) = 1$ for $m \geq n \log\log n$ [18], this expression is $O(n \log n+m)$. The space bound follows from the properties of condensible nodes and the observation that the number of square nodes bounds the number of round nodes. □

Corollary:

Given an initially connected graph G_0 and $O(|E_0|)$ preprocessing time, a sequence of $m = \Omega(n)$ *find block* and *insert edge* operations can be processed in $O(m\alpha(m,n))$ time.

Proof. The bridge-blocks and initial BBF of G_0 can be found in time $O(|E_0|)$ using one of the algorithms in references [9,19]. By lemma 1, the total number of pointer steps and condensations is $O(m)$, giving the bound. □

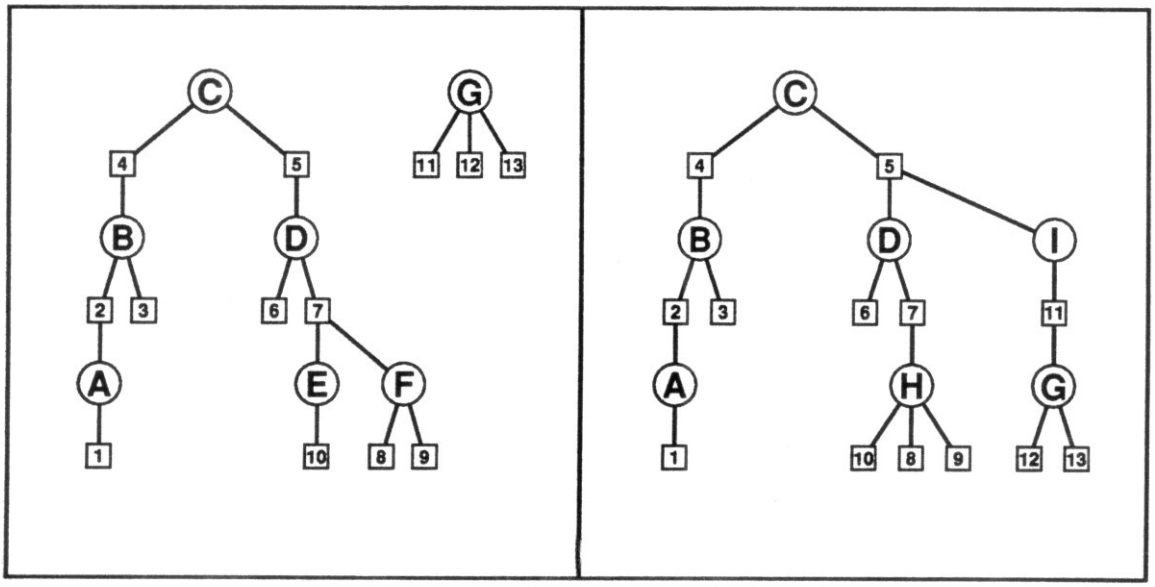
3. Maintaining Blocks On-line

The problem of maintaining blocks on-line is similar to that of maintaining bridge-blocks, and the algorithms are almost identical. We represent each block of G by a round node and each vertex by a square node. The square nodes now play a more important role, however, and the *block forest*, or BF, is different in character from the bridge-block forest. Whenever a vertex of G belongs to a given block, we create a tree edge between the corresponding square and round nodes in the BF. There are no other edges; no two square nodes and no two round nodes are adjacent. Since a single vertex can be an articulation point joining many blocks, and a block may be adjacent to many articulation points, the block tree generally has internal square nodes. If $\{u,v\}$ is a vertex pair that appears in block B, then in the block tree either u and v are both children of B or one is a parent of B and one is a child of B. The query *find block* (u,v) can be answered by returning the label of the single round node that lies on the tree path between u and v.

Figure 3a gives the block forest for the graph of Figure 1.

A *make vertex* (u) operation adds a single unconnected square node to the block forest. An *insert edge* (u,v,A) operation can have two opposite effects: either it links two components, increasing the number of blocks, or it creates a new cycle, possibly decreasing the number of blocks. If a component link occurs, then the inserted bridge forms a new block labeled by A. One of the two block trees, say the tree containing u, is rerooted at u; u becomes a child of A, and A becomes a child of v.

If the inserted edge creates a new cycle then path condensation occurs. In a block tree the path between u and v consists of alternating square and round nodes. The round nodes on the path are condensed into a single round node, $\tilde{r}$. The square nodes on the path may be ignored, since condensation of the round nodes ensures that these square nodes become children of $\tilde{r}$. Let w be the nearest common ancestor of the path nodes. (Recall that this implies that w is part of the path.) If w is a round node, then $\tilde{r}$ becomes a child of the parent of w. If w is a square node, it is not affected by condensation, and $\tilde{r}$ becomes a child of w. Examples of a linking edge insertion and a condensing edge insertion with a square nearest common ancestor are given in Figure 3b.



(a) (b)

Figure 3: a) Block forest of G. b) BF after performing *insert edge* $(8,10,H)$ and *insert edge* $(5,11,I)$.

We maintain the block forest with the same condensible node data structure used for the bridge-block problem. The *Link, Findpath,* and *Condensepath* functions described in the previous section can be modified in a straightforward way to perform the block tree transformations described above. With these functions, the block operations are implemented as follows:

make vertex (A):

Return *Maketree* (null). (This does not create a block, so label A ignored.)

find block (u,v):

Return *label* (*Findpath* (u,v)).

insert edge (u,v,A):

(1) If u and v are in the same component then execute *Condensepath* (*Findpath* $(u,v),A$) and terminate.

(2) If u and v are in different components, determine which is in the smaller component. Assume u is.

(3) Let $\hat{r} = Maketree\,(A)$.

(4) Execute *Link* $(u,\hat{r})$. Execute *Link* $(\hat{r},v)$.

The lemmas and analysis given for the bridge-block problem can be easily adapted to the block problem.

Theorem 2:

The on-line block problem can be solved in $O\,(n\log n+m)$ time and $O\,(n)$ space.

Corollary:

If the graph is initially connected, the on-line block problem can be solved in $O\,(m\alpha(m,n))$ time.

4. A Data Structure for an Optimal Bridge-Block Algorithm

In this section we replace the data structure used in section 2 with a more sophisticated data structure called the *link/condense tree*. We observe that in the bridge-block algorithm of section 2 most of the work occurs in processing component links. The link/condense tree is designed to perform everts efficiently, and hence speed component links, without increasing the time to perform *Findpath* and *Condensepath.* By maintaining the bridge-block forest with link/condense trees, any sequence of $n-1$ component links can be performed in $O\,(n\alpha(m,n))$ time, and it remains the case that $n-1$ path condensations take time $O\,(n\alpha(m,n))$.

The link/condense tree is derived from the dynamic tree data structure of Sleator and Tarjan [14,15]. For a full description of dynamic trees the reader should consult their papers. Below we will consider mainly the new aspects of link/condense trees. The following summary description of dynamic trees is taken from [15, p. 678]. (See Figure 4.)

> We represent each dynamic tree T by a *virtual tree* V containing the same nodes as T but having a different structure. Each node of V has a left child and a right child, either or both of which may be a null, and zero or more middle children. We call an edge joining a middle child to its parent *dashed* and all other edges *solid*. Thus the virtual tree consists of a hierarchy of binary trees, which we call *solid subtrees*, interconnected by dashed edges. The relationship between T and V is that the parent in T of a node v is the symmetric-order successor of v in its solid subtree in V, unless v is last in its solid subtree, in which case its parent in T is the parent of the root of its solid subtree in V. In other words, each solid subtree in V corresponds to a path in T, with symmetric order in the solid subtree corresponding to linear order along the path, from first vertex to last vertex.

The link/condense virtual tree is built with condensible nodes. Let v be a tree node contained in solid subtree U. $N(v)$ contains pointers to the following nodes: $vparent(v)$, the parent of v in the virtual tree V; $left(v)$ and $right(v)$, the left and right children of v in the solid subtree; and $pred(v)$ and $succ(v)$, the symmetric order predecessor and successor of v in the solid subtree. In general, $succ(v)$ points to the parent of v in T. In addition, $N(v)$ contains pointers $leftmost(v)$ and $rightmost(v)$. These are used only when v is the root of a solid subtree, at which time they point to the leftmost and rightmost nodes in this solid subtree. If t is the root of U and $l = leftmost(t)$, $r = rightmost(t)$, then $pred(l) = succ(r) = t$.

As a way to implement eversion efficiently, we allow a node v to occasionally enter *reversed state*, in which case the meanings of $left(v)$ and $right(v)$, $leftmost(v)$ and $rightmost(v)$, and $pred(v)$ and $succ(v)$ are reversed. (That is, $left(v)$ points to the right child, $right(v)$ points to the left child, etcetera.) To implement reversal, $N(v)$ contains a bit $reverse(v)$. The reversal state of node v is given by the exclusive-or of the *reverse* values stored in v and all its ancestors in the solid subtree.

The solid subtrees are built using splay trees [15]. A *splay at node x* moves x to the root of its solid subtree by applying a standard binary tree rotation to every edge along the path from x to the root (Figure 5a). A rotation rearranges left and right children while

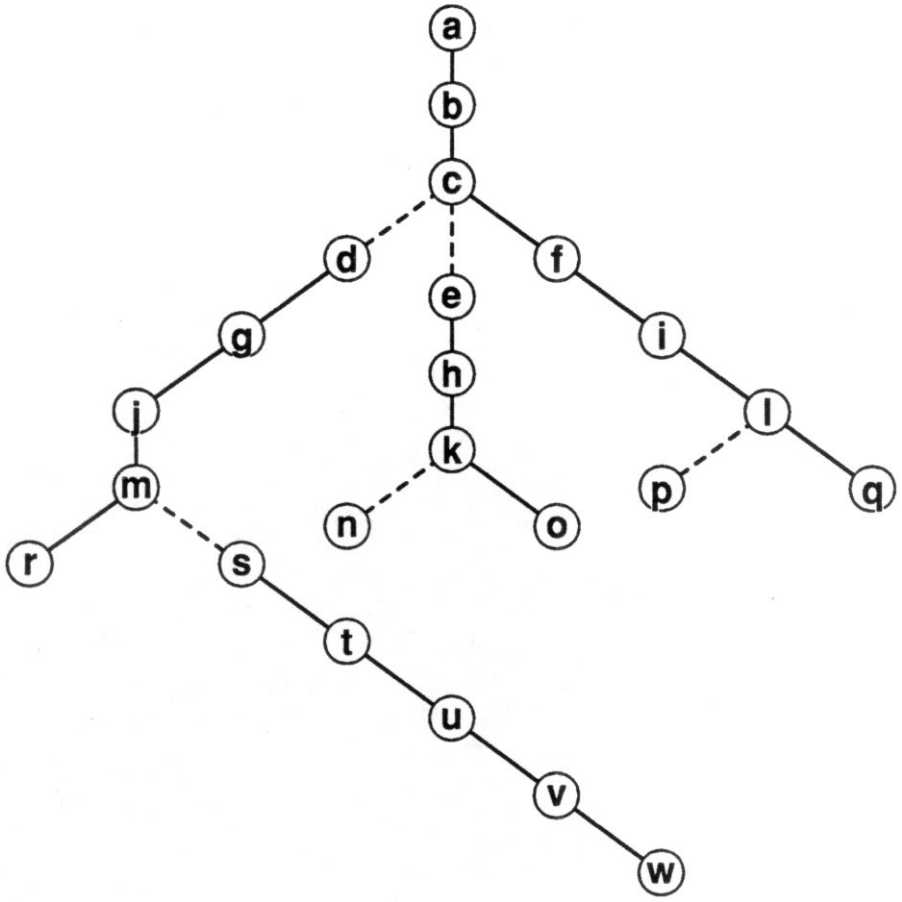


Figure 4a: A dynamic tree: the actual tree. Dashed edges separate paths corresponding to solid subtrees in the virtual tree [15].

preserving the symmetric order. Middle children are unaffected. The sequence of rotations is determined by the structure of the path [15]. To perform links efficiently, we use a procedure called an *extended splay at node v*, abbreviated *e-splay at v*. An extended splay at v moves v to the root of its virtual tree without changing the structure of the actual tree that the virtual tree represents. The e-splay algorithm is described fully in Sleator and Tarjan [15]. Besides rotation, the extended splay uses a second primitive called *splicing*, which exchanges a middle child with the left child of a solid subtree root (Figure 5b).

Reference [15] gives rules for updating the reversal bit and left, right and parent pointers of the nodes affected by a rotation or splice. Since rotation preserves symmetric order, it does not affect the values of $pred(v)$ or $succ(v)$. If the rotation replaces the old

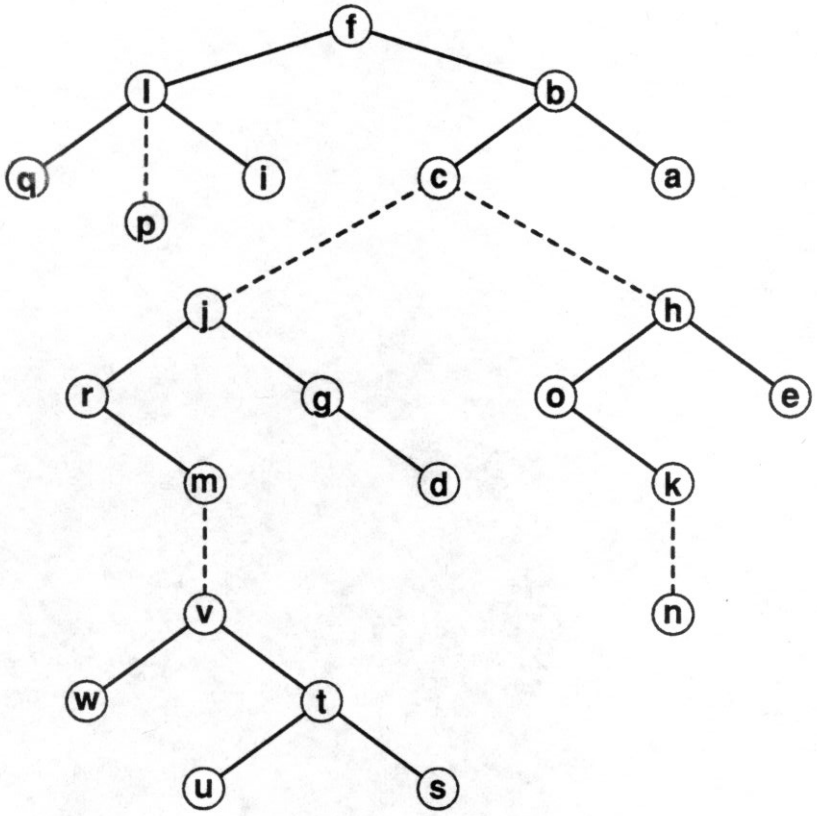


Figure 4b: The virtual tree representing the actual tree of Figure 4a [15].

subtree root r with a new root v, however, then the values of *leftmost* (r) and *rightmost* (r) must be copied to v, and *pred* (*leftmost* (r)) and *succ* (*rightmost* (r)) must be updated to point to v.

If w is the root of a solid subtree, u is the (possibly null) left child of w, and v is a middle child of w, then splicing makes v the left child and w a middle child. The additional pointers introduced in this paper are updated as follows (a prime indicates the new value):

$$
\begin{array}{llll}
leftmost'(u) = leftmost(w) & & pred'(leftmost'(w)) = w \\
leftmost'(w) = leftmost(v) & & pred'(leftmost'(u)) = u \\
rightmost'(u) = pred(w) & & succ'(pred'(w)) = w \\
pred'(w) = rightmost(v) & & succ'(rightmost'(u)) = u
\end{array}
$$

A *null splice at w* occurs when u is taken to be null. This simply makes the left subtree of w into a middle child without replacing it by another left subtree. (Null splices occur during everts.)

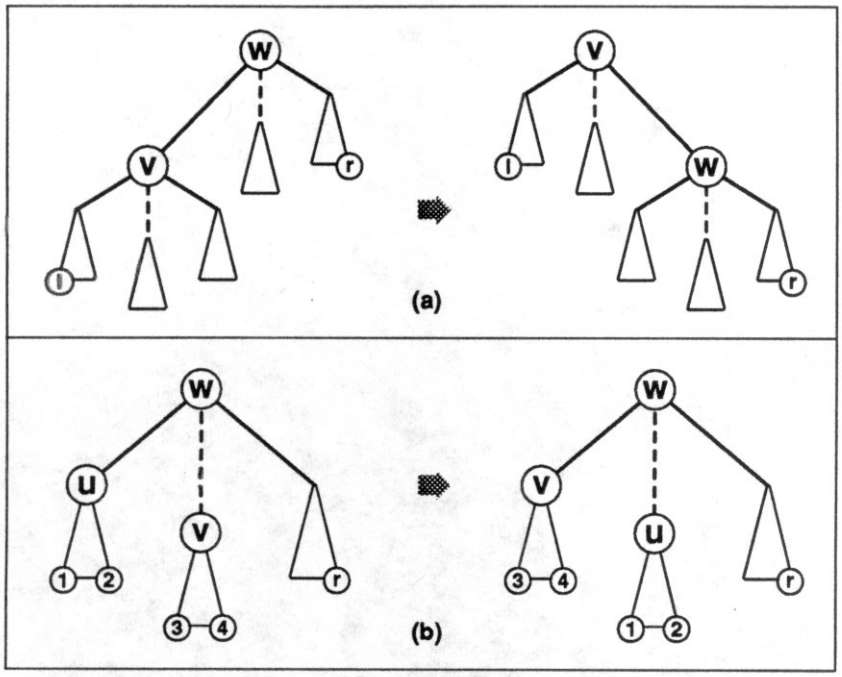


Figure 5: (a) Rotating the edge between v and w. If w is the root of its solid subtree, then pointers to l and r, the leftmost and rightmost nodes in the subtree, must be added to v. (b) Splicing v to w. Node 3 becomes the leftmost node in the solid subtree rooted at w, while nodes 1 and 2 become leftmost and rightmost, respectively, in the solid subtree rooted at u.

In the above, by $pred(v)$, $succ(v)$, $leftmost(v)$, and $rightmost(v)$ we mean the actual predecessor, successor, etc., of v. The fields containing these values may be switched if node v is in reversed state. During an extended splay at node x, the reversal state is computed for all nodes along the path from x to the root of its virtual tree. The reversal states of the leftmost and rightmost nodes in a solid subtree can be determined by examining which of their predecessor or successor pointers points back to the root. Both rotation and splicing require $O(1)$ pointer steps. The total time for an e-splay is bounded by a constant factor times the number of nodes on the path that is splayed.

In general, the linking together of two virtual trees is implemented by an e-splay in both. When the parent tree is much larger than the child tree, however, we will temporarily defer a full e-splay in the parent tree. To support this deferral process, the nodes of each virtual tree V are partitioned into a collection of disjoint sets, denoted D, which is a coarsening of the partition induced by the solid subtrees. (All nodes in a given solid

subtree belong to the same subset, but a set may include many solid subtrees.) These sets are maintained with a fast disjoint set union algorithm. We let $D\text{-}find(u)$ and $D\text{-}union(A,B)$ denote the standard set union operations on this partition. For node u, $N(u)$ must be augmented with an element name for use with $D\text{-}find$.

When *Maketree* creates a new single-node tree, a new set of D is created, containing only the new node. For each set S, the disjoint set union algorithm maintains a *virtual size* denoted $vsize(S)$. A new single set has virtual size 1. Whenever $D\text{-}union(A,B)$ occurs, the algorithm sets $vsize(A) = vsize(A) + vsize(B)$. Since node condensations may reduce the actual number of nodes in a set, we have $vsize(S) \geq |S|$. We denote by $vsize(V)$ the sum of the virtual sizes of the sets into which V is partitioned. Thus $vsize(V) \geq |V|$, with equality when no condensations have occurred. For convenience, we will use "size" to mean virtual size, unless otherwise noted.

If u is a middle child of v and $D\text{-}find(u) \neq D\text{-}find(v)$, the pointer $vparent(u)$ is called a *deferred link*. Deferred links are handled specially during the processes of everting a tree and of linking two trees together. The partition of V imposed by D satisfies the following two *deferred link invariants:*

I. Let v be any node in set S, and let r be the parent node of the first deferred link on the path from v to the root of V. If there is no such deferred link, then let r be the root of V. Then r is the same for all $v \in S$. Thus D defines a unique mapping $D\text{-}root$ such that $D\text{-}root(S) = r$, and the set $S \cup \{D\text{-}root(S)\}$ forms a connected subtree within V. Note that two sets could have the same $D\text{-}root$.

II. If S_2 is the set containing $D\text{-}root(S_1)$ then $vsize(S_2) \geq 2 \cdot vsize(S_1)$ (unless $D\text{-}root(S_1)$ is the root of V, in which case $S_1 = S_2$.

Lemma 3:

Let node v be an ancestor of node u in virtual tree V, and let $n = vsize(D\text{-}find(v))$. Then there are $O(\log n)$ deferred links on the path from u to v in V.

Proof. Straightforward application of invariant II. □

We now give a version of Sleator and Tarjan's extended splay that is modified to handle deferred links. The e-splay is a five pass process. In the first pass, we walk up the path from x to the root of its virtual tree. Each time we encounter a node y that is in a new solid subtree, we splay at y. After the first pass, the path from v to the root consists

of dashed edges and deferred links. In the second pass, we walk up the path, splicing at each dashed edge. After the second pass, the path consists of solid subtrees, containing only left children, separated by deferred links. In the third pass, we splay at each parent of a deferred link. After the third pass, the path from x to the root consists only of deferred links. In the fourth pass, we walk up the path, converting each deferred link into a regular dashed edge by uniting the sets that contain the parent and child of the deferred link, and splicing at the resultant dashed edge. After the fourth pass, all nodes on the path belong to the same set of the D partition, and x and the root of the virtual tree are in the same solid subtree. In the fifth pass, we splay at x, making x the virtual tree root. (See Figure 6.)

Lemma 4:

An extended splay preserves the deferred link invariants.

Proof. Only pass four of the e-splay affects the partition defined by D. After pass three, each node on the path belongs to a distinct set, whose D-root is also a path node. Let S' be the union of all sets produced by pass four. Clearly the nodes of S' form a connected subtree, and the root of the virtual tree is $D\text{-}root(S')$. Therefore invariant I still holds.

Let S be any set such that $D\text{-}root(S)$ is contained in S' following pass four. This implies that prior to pass four, $D\text{-}root(S)$ was contained in one of the sets that was united to form S'. Since the virtual size of S' is the sum of its constituent sets, invariant II must hold for set S after pass four. □

With the above machinery in place, we turn to the implementation of $Link(u,v)$. Basically, we perform an extended splay at u followed by a partial extended splay at v, and then make u a middle child of v. Following the extended splay at u, the right subtree of u contains the path from u to the root of its actual tree. To evert the tree at u, we perform a null splice at u and toggle the reverse bit in u. This reverses the direction of the solid path containing u, making u the root of the actual tree as well as of the virtual tree. (This implementation of evert is described in reference [14]).

Let k be the size of the D set containing u after the extended splay at u. If k is at most half the size of the D set containing v then no e-splay at v occurs and u is simply made a child of v by a deferred link. This preserves the deferred link invariants. Otherwise, a partial extended splay at v is done. To determine the extent of the e-splay, we

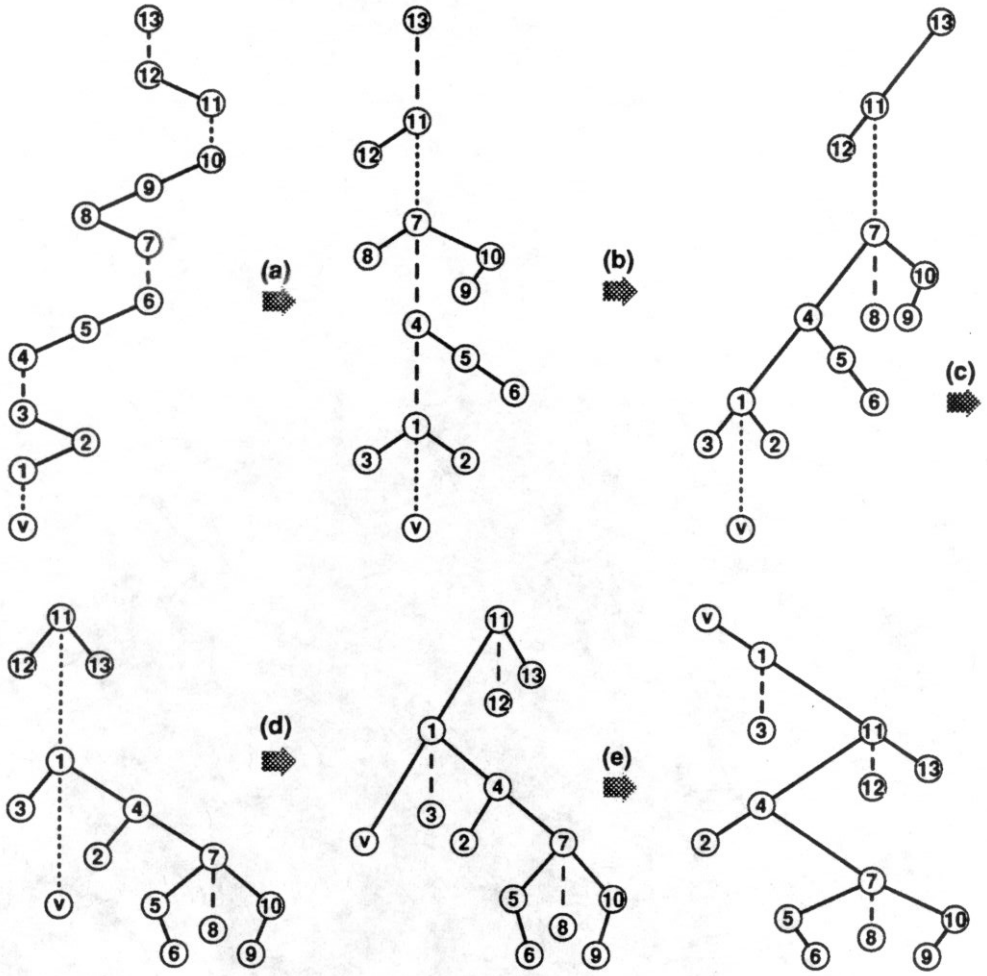


Figure 6: An extended splay at node v. Subtrees of nodes on the path have been deleted for clarity. Dotted lines represent deferred links. (a) First pass: splaying inside solid subtrees (see [15] for details of splaying). (b) Second pass: splicing dashed edges. (c) Third pass: splaying solid subtrees between deferred links. (d) Fourth pass: converting deferred links to dashed edges, followed by splicing. (e) Fifth pass: splaying along final solid path.

walk up the tree from v until encountering a deferred link from node x to node y such that the sum of N and the sizes of the sets on the path up to x is at most half the size of the set containing y. We treat x as the root of the virtual tree and do an extended splay at v. This e-splay makes v a child of y by a deferred link, after which u is made a middle child of v and we perform $D\text{-}union(D\text{-}find(u), D\text{-}find(v))$, thereby making the deferred link joining u and v into a dashed edge. If we reach the root of the virtual tree before finding such a deferred link, we simply do a full e-splay, moving v to the root of the virtual tree; then we attach u to v by a dashed edge as described above.

From Lemma 4, the two extended splays preserve the deferred link invariants. It is easily observed that if no e-splay at v occurs, or if the e-splay at v makes v the root of its virtual tree, the link preserves the deferred link invariants. Suppose a partial e-splay at v takes place, terminating at the k^{th} deferred link. Let the k^{th} link pass from node x to node y. Prior to the extended splay at v, node y is the D-root of the set containing x, and after the e-splay y is the D-root of the combined set resulting from the e-splay. After the e-splay, the size of the set containing v is $N+P_{k-1}(v)$, which by the termination condition on the initial walk up from v is at most half the size of the set containing y. Therefore both invariants are preserved in the case of a partial e-splay at v.

Now we turn to the implementation of *Findpath* (u,v). Let T be an actual tree and let V be the virtual tree representing T. *Findpath* (u,v) returns P as a list of pointers to nodes, ordered from u to v, so that two nodes are adjacent in P if and only if they are adjacent in T. As in section 2, we find a path from u to v by walking up the paths from u and from v to the root of the real tree T. We generate the two paths in lock-step and stop as soon as they intersect at the nearest common ancestor of u and v. The general strategy to find the path from node u to the root of T is to locate u in V, and follow successor pointers until reaching the rightmost node r in the solid subtree containing u. Then we jump to the subtree root t via *succ* (r) and follow the dashed edge *vparent* (t) to the next node on the path. This next node lies in a new solid subtree. We repeat this procedure until encountering a null *vparent* (t) pointer. Then r is the real tree root.

Unfortunately, this strategy cannot be applied directly, since it is impossible to determine the reversal state of u, and hence the interpretation of *pred* (u) and *succ* (u), without walking down to u from the solid subtree root. This would increase the cost of path finding to $O(d+|P|)$, where d is the depth of u. It is possible, however, to generate a path that consistently moves in the same direction once a direction in which to start is chosen. If we follow a pointer from node x to node y, then one of *pred* (y) or *succ* (y) must point back to x. To generate the next step in the path, we can simply follow the other pointer. Furthermore, once the path encounters the leftmost or rightmost node in the solid subtree, we can jump to the subtree root and determine whether the path goes up or down the actual tree.

Given this, we can find the path P between nodes u and v using the following algorithm. We march in lock-step in both directions out of u and v, marking nodes

encountered. Thus in parallel we generate two pairs of tentative paths, one node at a time. When one tentative path encounters a solid subtree root, we determine which of the two tentative paths goes up the tree, and discard the other tentative path, after unmarking the incorrectly marked nodes. If the subtree root was encountered in the incorrect direction, then we then proceed in the correct direction (in step with the other search) until finding the subtree root again. Then we follow the parent pointer of the root into a new solid subtree, at which point ambiguity again arises. Path generation is complete when one tentative path from u intersects one tentative path from v. By marching in two directions simultaneously, we double the number of pointer steps required to generate one node of the path, but the total time to generate a path of length l remains $O(l)$.

Knowing a way to find P, we can turn to the implementation of *Condensepath* (P,A). P consists entirely of round nodes, all of which must be condensed into a single round node. Let *Condense 2path* (x,y,A) be a function that condenses a path consisting of two adjacent nodes x and y, and labels the resultant node z by A. *Condensepath* (P,A) proceeds by repeating the following *condensation step* until P consists of a single node: select a pair of adjacent nodes x,y from P. Perform *Condense 2path* (x,y,A). (Let T' be the tree resulting from this condensation.) Replace x and y in P with z.

Since path condensation preserves adjacency, it is clear that after a condensation step, P is the path between u and v in tree T'. An inductive proof using this observation shows that *Condensepath* correctly condenses a path of arbitrary length. We now turn to the implementation of *Condense 2path* (x,y,A). We distinguish two cases based on the relationship between x and y in the virtual tree: 1) x and y are in the same solid subtree; and 2) x and y are in different solid subtrees. In Case 1, at least one of *pred* (x), *succ* (x) contains a pointer to y. In case 2, neither of these fields contain a pointer to y. This fact can be used to determine which case applies.

Case 1 is shown in Figure 7a. The adjacency of x and y in the actual tree implies that one is the ancestor of the other in the solid subtree. Assume that y is the ancestor of x. For the moment, we assume that the reversal states of x and y are known. Let y be the successor of x. (The other possibility is symmetric.) This implies that the right subtree of x is empty, and that the successor pointer of x points to y. By examing the fields of x and y for this configuration, we can determine which of them is in fact the ancestor. That

is, y is the ancestor of x if $right(x)$ is null and $succ(x)$ is y, or, in the symmetric case, $left(x)$ is null and $pred(x)$ is y.)

Let $w = vparent(x)$. (It may be the case that $w = y$.) Let $u = left(x)$. To begin the condensation, node u is made a child of w in place of node x. This is done by replacing the pointer to x in w by a pointer to u and setting $vparent(u) = w$. The old value of $reverse(u)$ is replaced by the exclusive-or of $reverse(u)$ and $reverse(x)$. Nodes x and y are combined to form z. The fields in $N(z)$ are copied from $N(y)$, with the exception of $label(z)$, which is set to A, and of $pred(z)$, which is copied from $N(x)$.

Combining x and y, which unites their subnode sets, guarantees that all former pointers to x or y are now pointers to z. The former predecessor of x and the former successor of y become the predecessor and successor, respectively, of z. All former middle children of x or y become middle children of z. Hence, node adjacencies in the transformed tree are correctly preserved. In general, the reversal states of x and y are not known, but they are not actually needed. One of $left(x)$ or $right(x)$ must be null, so the pointer to u can be found in the other. Similarly, one of $pred(y)$ or $succ(y)$ points to x. The corresponding field in z is copied from whichever of $pred(x)$ or $succ(x)$ does not contain a pointer to y.

Case 2 is shown in Figure 7b. The adjacency of nodes x and y implies that one, say x, must be rightmost in its solid subtree. The root of this solid subtree is a middle child of y. The reversal state of x can be computed by determining which of $pred(x)$ or $succ(x)$ points to the root of the solid subtree. Since x is rightmost, its right subtree is empty. Let $u = left(x)$. If x is the root of its subtree, a null splice is performed to make u a middle child of x. Otherwise, let $w = vparent(x)$. Using the pointer and reversal bit updates given in case 1, u replaces x as the right child of w. This makes $pred(x)$ rightmost in the solid subtree. The leftmost field in the solid subtree root and the successor field in $pred(x)$, (i.e., the field containing a pointer to x) must be updated accordingly. Finally, x and y are condensed to give z. $N(y)$ is copied to $N(z)$, and z is labeled A.

Lemma 5:

Condense 2path (x,y,A) preserves the deferred link invariants.

Proof. If x and y initially belong to the same set S of D, then the condensation does not change the invariants for S. If, prior to the condensation, x (or, equivalently, y) is D-$root(S')$ for some set S', then all nodes of S' are descendants of one or more children of

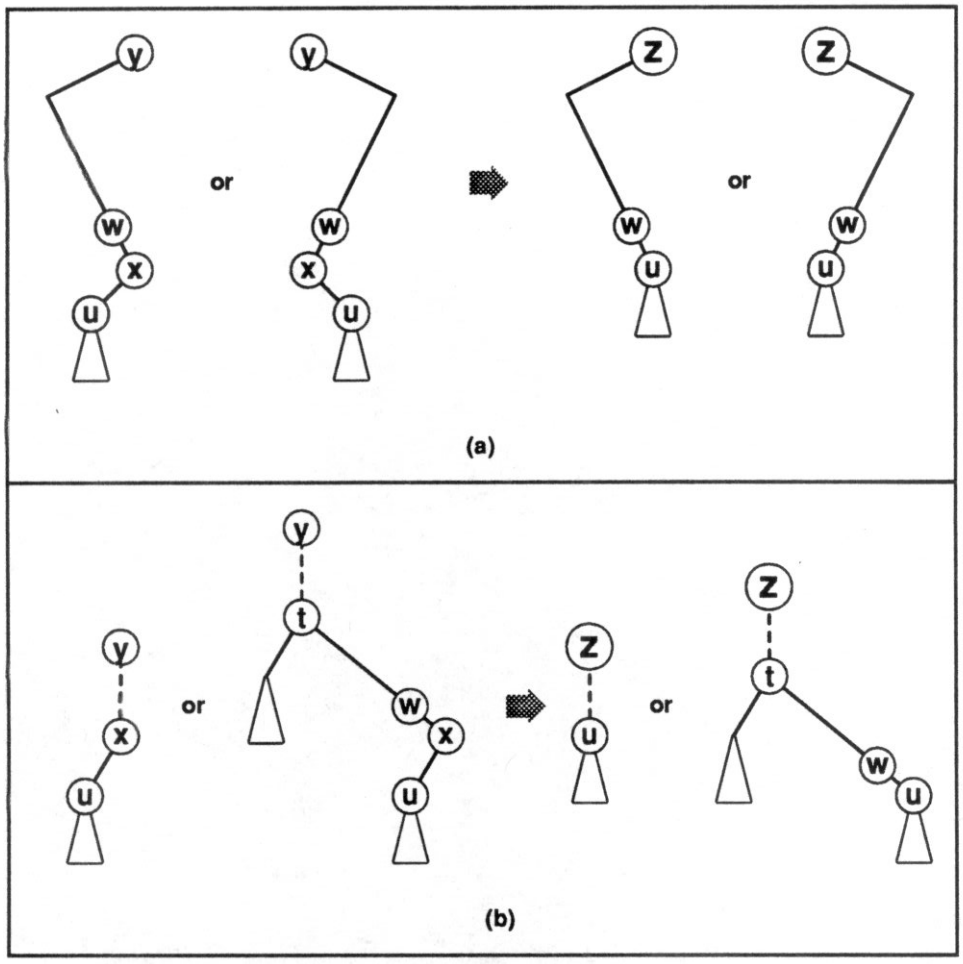


Figure 7: a) Case 1 of *Condense* 2*path* (x,y,A). $w = vparent(x)$. (We may have $w = y$.) Node z is the result of the condensation. b) Case 2 of *Condense* 2*path* (x,y,A). (We may have $t = w$.)

x. The condensation makes all children of x (and y) into children of z. Hence invariant I is preserved for S', with $D\text{-}root(S') = z$. Although the condensation decreases the actual size of S by one, its virtual size is unchanged. Therefore invariant II is preserved for all sets whose D-root is contained in S.

In case 1 of *Condense* 2*path*(x,y,A), x and y belong to the same solid subtree, and so must belong to the same set S. In case 2, however, x and y may belong to two different sets, say S_1 and S_2, respectively. This situation occurs when the root of the subtree containing x is a child of y by a deferred link. The invariants imply that y is $D\text{-}root(S_1)$ and $vsize(S_2) \geq 2vsize(S_1)$. For any descendant of x that belongs to S_1, the deferred link

between the solid subtree root and y is the first deferred link on the path to the root of the virtual tree. The condensation makes all middle children of x into middle children of z by deferred links, while the left subtree of x remains in the solid subtree that contained x. Thus after the condensation z is the parent of the first deferred link on the path from any element in S_1 to the virtual tree root. Since no virtual size changes, invariants I and II are preserved for S_1 and S_2, with $D\text{-}root(S_1) = z$. As before, invariants I and II are preserved for all sets whose D-root is contained in S_1 or S_2 prior to condensation. □

The following lemma follows from an examination of the *Condense 2path* algorithm.

Lemma 6:

During condensation, the number of virtual tree descendants of any node is non-increasing.

5. Amortized Analysis of Link/Condense Operations

To begin the running time analysis, we observe that the time to perform $Link(u,v)$ is a function of the original depth of u and the length of the path traveled up from v. We can measure this depth by the number of rotations performed during the two e-splays. The time to perform $Findpath(u,v)$ is determined by the number of times a node along the path is found, and the time to perform $Condensepath(P,A)$ is measured by the number of times a pair of nodes are combined by *Condense 2path*. We define rotation, node-finding, and pairwise combination to be basic primitives of cost 1, since each is responsible for at most a constant number of condensible node operations. The cost of a link/condense tree operation is measured by the number of primitives executed while performing the operation, and the amortized time for an operation is the product of the cost of the operation with the amortized time to perform $O(1)$ condensible node manipulations.

The amortized analysis uses a *potential function* [15,17] defined on the virtual tree structure. For each node x in a virtual tree, we define the *weight of node x*, $w(x)$, as the number of virtual tree descendants y of x, including x itself, such that $D\text{-}find(y) = D\text{-}find(x)$, i.e. all descendants of x belonging to the same D-set as x. We define the *logarithmic weight* $\lg w(x)$ of node x to be $\log w(x)$.† The potential Φ is defined as follows:

† By log we mean continuous binary logarithm.

$$\Phi = \sum_{nodes\ x} 2\text{lgw}(x) + \sum_{S \in D} 18(3 + \log vsize(S))$$

Let Φ_i denote the value of the potential function after the ith operation. If t_i is the actual cost of the ith operation then the amortized cost a_i of the ith operation is defined by

$$a_i = t_i + \Delta\Phi \text{ , where } \Delta\Phi = \Phi_i - \Phi_{i-1}$$

We begin by analyzing the cost of an extended splay at node v. Suppose that there are k deferred links on the path from v to the root of the virtual tree (or, in a case of a partial e-splay, to the node that is treated as the root). Then the path passes through $k+1$ sets $S_0, S_1, \cdots, S_k$ of the D-partition, where S_0 contains v and S_k contains the virtual tree root. Abbreviating $vsize(S_i)$ by N_i, let $p_i = \sum_{0 \le j \le i} |S_j|$, and $P_i = \sum_{0 \le j \le i} N_j$. Since $N_i \ge |S_i|$ for all i, we have $P_i \ge p_i$.

Lemma 7:

$P_i \le 2N_i$ for $0 \le i \le k$.

Proof. From Lemma 3, if $N_i = m$, then $i \le \log m$. By invariant II, $N_{i-1} \le N_i/2$, $N_{i-2} \le N_{i-1}/2$, etcetera. Thus

$$\sum_{0 \le j \le i} N_j \le \sum_{0 \le j \le \log N_i} \frac{N_i}{2^j} \le 2N_i - 1.$$

□

Lemma 8:

The amortized cost of an extended splay at v is at most $24\log N_k + O(1) + \sum_{0 \le i \le k} [(12\log N_i + 4) - 18(3 + \log N_i)]$, where k and N_i are defined as above.

Proof. Passes one, two, three, and five do not affect the partition defined by D, so any changes to the potential in these passes occur only in the term involving $\text{lgw}(x)$. In reference [15], Sleator and Tarjan use this reduced potential function in analyzing their splay and extended splay algorithms. From their paper we draw two facts: first, the cost of the splay in pass five is at most $6\log p_k + 1$; second, the cost of passes one through three in the contiguous section of path between the i^{th} and $i+1^{st}$ deferred links is at most $12\text{lgw}(x_i)+2$, where x_i is the i^{th} node on the path after pass three. Thus the cost of passes

one, two, three, and five is

$$6\log p_k + 1 + \sum_{0 \le i \le k} [\,12\lg w(x_i) + 2\,].$$

After pass three, the path from v to the root consists of the $k+1$ nodes x_i. Pass four does no rotations so it has zero actual cost, but the unions done in pass four change the potential. The $k+1$ sets S_i, $0 \le i \le k$, are replaced by a single set S' with virtual size P_k. The unions also increase the weight of each node x_i on the path, since the number of descendants of x_i that belong to the same set as x_i increases. By invariant I, node x_i is *D-root* (S_{i-1}) for $1 \le i \le k$. This implies that only the nodes on the path increase in weight as a result of the unions. During pass four node x_i increases in weight by at most $p_i - w(x_i)$, where (x_i) is the weight of x_i prior to pass four. Thus the cost of pass four is at most

$$18(\log P_k + 3) + \sum_{0 \le i \le k} [\,2\log(p_i) - 2\lg w(x_i) - 18(3 + \log N_i)\,]$$

We have $P_i \ge p_i \ge w(x_i)$ and $N_i \ge w(x_i)$. From Lemma 7, $2N_i \ge P_i$. Combining terms, and using these observations, the lemma follows. □

Lemma 9:

Let u be contained in virtual tree V. The amortized cost of *Link* (u,v) is $O(\log n)$, where $n = vsize(V)$.

Proof. Using Lemma 8, it is straightforward to show that the extended splay at u has cost $O(\log n)$, since each term of the sum in Lemma 8 is less than zero, and $N_k \le n$. If no e-splay at v occurs, this is the entire cost of the link. To analyze the cost of an e-splay at v, we divide the sum of Lemma 8 into two parts. Let l be minimal such that $N_l \ge n$. We rewrite the sum of Lemma 8 as $A + B$, where

$$A = 12\log N_l + 4 + \sum_{0 \le i < l} [(12\log N_i + 4) - 18(3 + \log N_i)]$$

and

$$B = 6\log N_k + O(1) + \sum_{l < i \le k} [(12\log N_i + 4) - 18(3 + \log N_{i-1})].$$

To bound these sums, we observe that since the e-splay at v did not terminate upon reaching the i^{th} set, it must be the case that $n + P_{i-1} > N_i/2$. Each term of the sum in A

is at most zero, so A is $O(\log N_l)$. We bound N_l as follows: by Lemma 7, $P_{l-1} \leq 2N_{l-1}$. By the definition of l, $n > N_{l-1}$. Together with our initial observation, this implies that $N_l < 6n$, and hence $A = O(\log n)$.

To bound B, we observe that by the definition of l, $N_i \geq 2n$ for $i > l$. Together with Lemma 7, this implies that $5N_{i-1} \geq 2(n + P_{i-1}) > N_i$. Using this inequality, we find that each term of the sum in B is at most $-(6\log N_i - 8)$, and hence B is $O(1)$. Therefore, the extended splay at v has amortized cost at most $O(\log n)$. Finally, making u a middle child of v, and uniting the two sets containing u and v, increases the potential of v by at most $O(\log n)$. Thus *Link* (u,v) has amortized cost $O(\log n)$. □

Theorem 3:

A sequence of m bridge-connected component operations can be performed in worst-case time $O(m\alpha(m,n))$.

Proof. As before, each condensible node operation takes amortized time $O(\alpha(m,n))$. A call to *Maketree* costs $O(1)$, since it increases the potential by a constant amount. Path finding does not modify the tree, so the potential is unchanged. Since a *find block* operation requires the finding of a constant-length path, the total time spent in *find block* operations is $O(m\alpha(m,n))$. Lemmas 5 and 6 imply that *Condense 2path* does not increase the potential and so has amortized cost at most 1. Thus the total cost of condensing edge insertions is $O(n)$, and the worst-case time is $O(n\alpha(m,n))$.

To determine the total cost of component links, we note that since the number of square nodes bounds the number of round nodes, the virtual size of a bridge-block tree is at most twice the number of vertices in the corresponding component of the graph. Combining this observation with Lemmas 7 and 8, we find that the amortized cost of a component link is $O(\log n)$, where n is the number of vertices in the smaller component. This gives the following recurrence for the total cost of component links:

$$T(n) \leq \max_{1 \leq j \leq n/2} \{T(j) + T(n-j) + c(1 + \log j)\}$$

This implies $T(n) = O(n)$. There is at most one call to *D-find* or *D-union* per pointer step, and hence the total time spent manipulating the D partition is $O(n\alpha(T(n),n))$, which is $O(m\alpha(m,n))$. Thus the total cost of processing component links is $O(m\alpha(m,n))$. □

In conclusion, we remark that all data structures are size $O(n)$, and thus the space required by the algorithm is $O(n)$.

6. A Data Structure for an Optimal Block Algorithm

The link/condense tree data structure of the previous section can be modified to produce an efficient data structure for maintaining the block structure forest. Unfortunately, these modifications are not trivial. The complications arise from the fact that a block tree path consists of alternating round and square nodes, and during path condensation only the round nodes are combined. Care must be taken in restructuring the virtual tree so that movement of the square nodes does not increase the value of the potential function (which would cause the amortized time per update to increase).

We take as our starting point the complete link/condense tree data structure of the previous section. On this data structure, we impose the following *solid subtree restrictions:*

(i). *Each solid path in the actual tree either is a single round node or terminates in a square node (i.e., a square node is rightmost in the corresponding solid subtree of the virtual tree.)*

(ii): *All square nodes are leaf nodes of their solid subtrees.*

The algorithm for *Findpath*(u,v) given in Section 4 can be used here without changes. To implement *Condensepath*(P,A), we use the function *Condense 3path*(r,u,s,A), which transforms a three-node path consisting of round node r, square node u, and round node s, and returns the round node t, labeled A, resulting from the condensation of r and s. Path condensation proceeds by repeatedly selecting a three-node path from P and replacing it with the single node returned by *Condense 3path*. The process terminates when P consists of a single round node. There are four cases for *Condense 3path*(r,u,s,A):

1) all three nodes are in the same solid subtree;

2) u and one round node are in one subtree whose root is a middle child of the other round node;

3) u and one round node are in one subtree, and the other round node is a middle child of u;

4) all three nodes are in different solid subtrees.

Which case applies can be determined by examining *vparent* pointers.

In cases 1 and 2, the relationship between virtual tree and actual tree implies that in the actual block tree, one round node is an ancestor of u and the other is a descendant of u. In case 1, restriction (i) implies that in the solid subtree one of r and s is a descendant of the other, and u is a child of the descendant round node. This fact can be used to determine which of r and s is the ancestor. Let r be the descendant. (The subcase with r the ancestor is analogous.) Node u is either the left or right child of r. The other child of r is made a child of $vparent(r)$ using the pointer and reverse bit updates described in case 1 of the *Condense 2path* routine of Section 4. Then nodes r and s are condensed together to give node t, with the fields of t being copied from the fields of s. This makes u a middle child of the condensed node.

In case 2, assume that r is in the same solid subtree as u. (The case of s in the same solid subtree is analogous.) Restriction (i) implies that u is the right child of r and that u is rightmost in its solid subtree. As in case 2 of *Condense 2path*, the left child of r is made into the right child of $vparent(r)$. Then nodes r and s are condensed together to give node t, with the fields of t being copied from the fields of s. This makes u a middle child of the condensed node.

Cases 3 and 4 are simpler. At least one of the round nodes r and s is a middle child of u. By restriction (ii) this round node must be a singleton solid subtree. The two round nodes are simply condensed to give t. The fields of t are copied from whichever round node is not a descendant of u, or, if both r and s are descendants, from either one.

The four cases are shown in Figures 8a through 8d. From examination of the cases, it is clear that *Condense 3path* does not violate the solid subtree restrictions. The proofs of Lemmas 5 and 6 in Section 4 can be adapted to show that *Condense 3path* preserves the deferred link invariants, and does not increase the weight of any node.

We now turn to the implementation of $Link(u,v)$. Although *Link* remains unchanged in basic design, the existence of solid subtree restrictions necessitates changes to the extended splay procedure, and this in turn causes slight modifications to *Link*.

Recall that the extended splay is based on *splaying*. A splay at node x moves x to the root of its solid subtree by rotating every edge along the path from x to the root. Splaying does not rearrange the subtrees rooted at nodes off the path from x to the root.

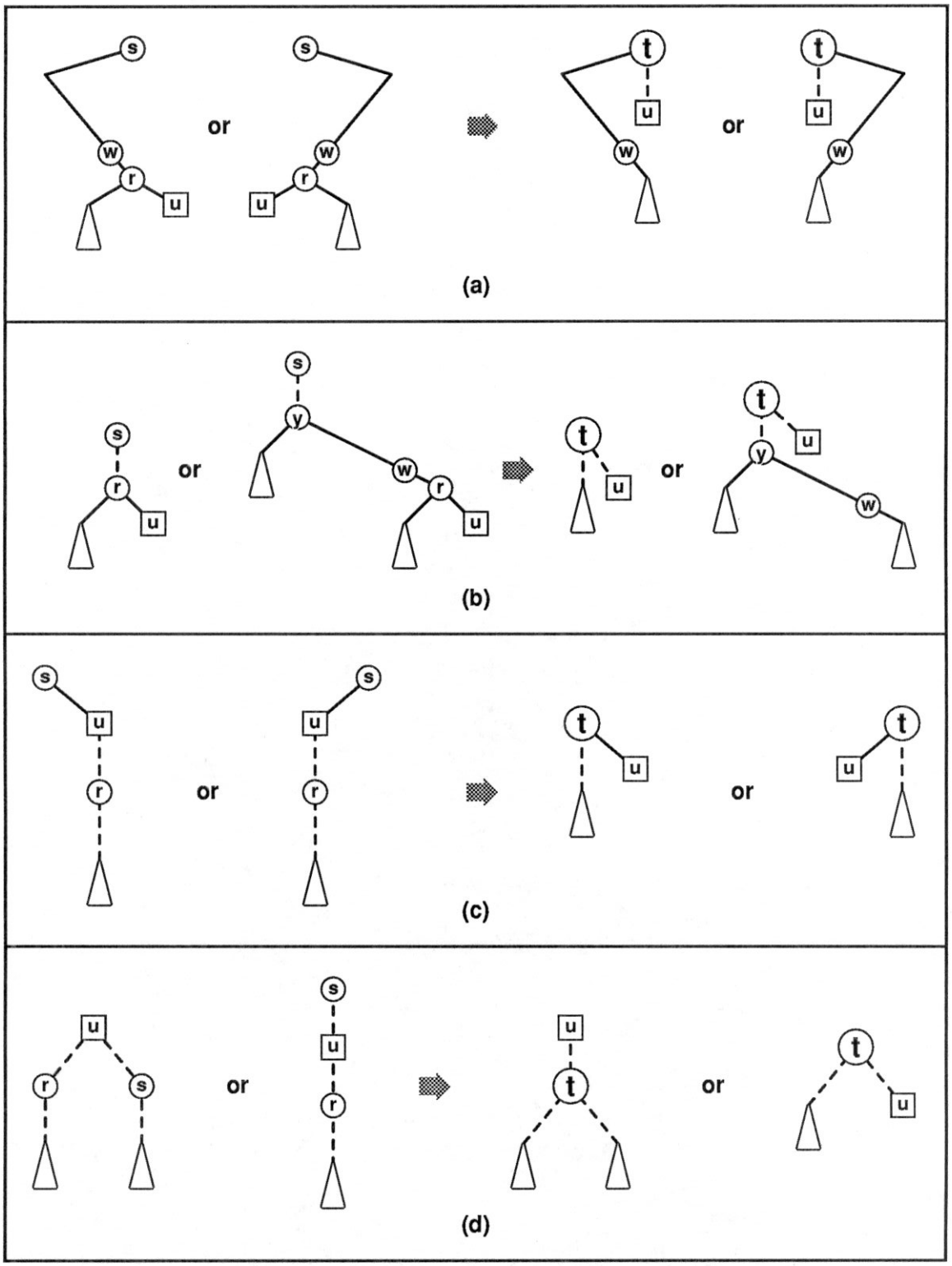


Figure 8: Cases 1 through 4 of *Condense* 3*path* (r,u,s,A).
Node t is the result of condensing nodes r and s.

This implies that if x is round, a splay at x cannot violate restrictions (i) or (ii). Splaying a square node to the root, however, will violate restriction (ii) except in the trivial case that the square node is a singleton solid subtree. Therefore, we define the function *square-splay* (v), which moves a square node to within one step of the root.

```
square-splay (v) begin
    splay at pred (v); splay at succ (v);
    rotate the edge between v and pred (v);
    perform a null splice at pred (v);
    perform a null splice at v;
end
```

In the above, by *pred* (v) and *succ* (v) we mean the actual predecessor and successor, respectively, of v. The fields containing these values may be switched if node v is reversed. (The reversal states of all nodes along the splay path are computed by an initial walk down the path.) The behavior of splaying is such that after the first two splays, v is a right child of *pred* (v) and *pred* (v) is a left child of *succ* (v) (see [15]). If v has no predecessor or successor, then the code involving the missing node can simply be ignored. For example, if v has no predecessor, then we need only splay at the successor. At the conclusion of *square-splay* (v), node v remains a leaf; either it is a singleton solid subtree or it is the left child of the root. Thus *square-splay* does not violate the solid subtree restrictions.

We must also be careful when splicing. An attempt to splice a round child to a square parent may violate restriction (ii), while an attempt to splice any child into a round node that forms a single-node subtree may violate restriction (i). To handle the first case, we introduce a new splice function, *square-splice* (r), where r is a round child of a square parent. Note that by restriction (i), r is a single-node solid subtree.

```
square-splice (r) begin
    let v = vparent (r); splice r to v.
    rotate the edge between r and v;
end
```

At the conclusion of *square-splice* (r), square node v is the right child of r. Since r had no right child initially, v remains a leaf node after the rotation. Thus *square-splice* does not violate the solid subtree restrictions.

In the second case, we can allow the splicing of a left subtree to a round node r that forms a single-node subtree as long as r has a parent. Since this parent must be a square node, r will immediately participate in a square-splice that will give it a square right child, guaranteeing that restriction (i) is not violated. The splice is disallowed if r is the root of the virtual tree (and hence the root of the actual tree), or if there is a deferred link from r to its parent.

As before, an extended splay at v is a five-pass process. (In the block algorithm we only e-splay at square nodes.) We use square-splays and square-splices as needed, and do not splice at singleton round nodes from which there are deferred links. After the first pass, the path from v to the root consists of dashed edges, deferred links, and solid edges between a square node and its parent. After pass three, the path from v to the root consists primarily of deferred links. Between each pair of deferred links there is a section of path that consists of either a single round node, or a solid edge between a square node and its round parent, or a solid edge between a square node and its round parent followed by a dashed edge leading into a single-round-node round subtree. After passes four and five v is either at the root, is a left or middle child of the root, or is the left child of a middle child of the root. Figure 9 gives an example of the first three passes of an extended splay, showing square-splays and square-splices.

The *Link* algorithm is essentially the same as that of Section 4. The only difference is in the way a tree is everted at square node v. Consider the possible cases after the initial e-splay at v. If v is the root of its virtual tree, no processing is needed to do the eversion. If v is the left child of the virtual tree root, then we simply toggle the reverse bit in the root. If v is a middle child of the root, we perform an ordinary splice at v and toggle the reverse bit in the root. If v is a left child, and $vparent(v)$ is a middle child of the root, then we perform an ordinary splice to make $vparent(v)$ a left child of the root, and toggle the reversal bit in the root. In the latter three cases, v becomes the end of a solid subpath, so restriction (i) is not violated.

The modifications to splaying and splicing do not affect the way the deferred link partition of the virtual tree is maintained. We may therefore use the proof of Lemma 4 in Section 4 to conclude that e-splay and *Link* preserve the deferred link invariants. With our description of the block algorithms complete, we turn to the analysis of their running times.

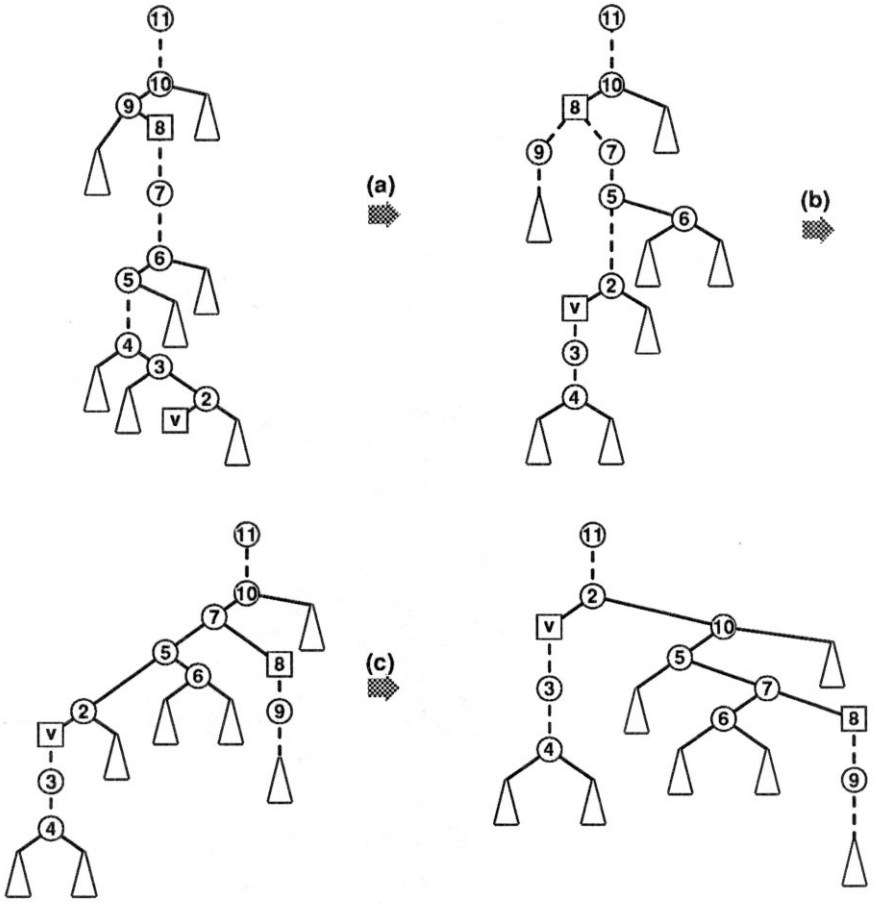


Figure 9: Passes 1-3 of an extended splay at node v. The path contains no deferred links. Triangles represent subtrees with square rightmost nodes. (a) First pass: splaying and square-splaying inside solid subtrees. (b) Second pass: splicing and square-splicing dashed edges. (c) Third pass: square-splay along final solid path.

7. Amortized Analysis of the Block Link/Condense Tree

In general our analysis here closely follows that of Section 5. We define the potential Φ to be:

$$\sum_{\text{nodes } x} 5 \lg w(x) + \sum_{S \in D} 85 (3 + \log vsize(S))$$

It is straightforward to adapt the analysis of Section 5 to show that the total time spent in queries and condensing edge additions is $O(m\alpha(m,n))$. To achieve the same bound for

component links, it suffices to demonstrate that the amortized cost of *Link* (u,v) is still $O(\log n)$. As before we measure cost by number of rotations. We will be fairly terse in the analysis, relying on results proved in reference [15]. The reader is advised to examine that paper for further details and explanation.

We begin by analyzing the costs of *square-splay* (v) and *square-splice* (r). From Lemma 1 of reference [15] we draw the following fact: let x be a left or right child of y. A single rotation of the edge between x and y has amortized cost $1 + 3(5\lg w(y) - 5\lg w(x))$. This bounds the cost of a square-splice.

From the same source, we draw a second fact: let y be the root of a solid subtree that contains node x. The amortized cost of a splay at x is at most $1 + 3(5\lg w(y) - 5\lg w(x))$. In executing *square-splice* (v) we perform two splays and a single rotation. The properties of symmetric order imply that, since v is a leaf, both the predecessor and the successor of v must at all times be ancestors of v, and hence their logarithmic sizes are always greater than that of v. This implies that the amortized cost of *square-splay* (v) in a solid subtree rooted at t is at most $3 + 45(\lg w(t) - \lg w(v))$.

We now bound the cost of the first three passes of an extended splay at node v.

Lemma 10:

Consider a section of the path from v to the root that is bounded by deferred links, i.e. all nodes on the path are in the same set of the D-partition. Let x be the lowest node on the section of path and let t be the highest node. (Thus x is a parent of a deferred link and t is a middle child of a deferred link.) The cost of passes one, two, and three in this section of path is is at most $75\log n + 13$, where $n = w(t)$.

Proof. Let k be the number of solid subtrees on the initial path from x to t. The amortized cost of the first pass is at most $3k + 45\lg w(t)$. This bound follows from summing the splay or square-splay costs within each solid subtree. The amortized cost of the second pass is at most $k + 15\lg w(t)$, since there are most k square-splices and the sum again telescopes. Thus the amortized cost of first two passes is $4k + 60\lg w(t)$.

If x is square, then after the second pass, x is at depth k, and in the third pass, *square-splay* (x) will perform exactly one splay, moving the parent (successor) of x to the root (or to within one step from the root, if t forms a round single-node subtree). If x is round, x is at depth $k-1$ and is itself splayed to the root in the third pass. Therefore, at least $k-2$ rotations occur in the third pass. We charge 5 for each of these $k-2$ rotations;

the additional charge accounts for $4k-8$ of the rotations left over from the first two passes. From the discussion of e-splay in reference [15], we find that even with this additional charge, the amortized cost of the final splay is at most $5 + 15\lg w(t)$. Summing over the three passes, we obtain the desired result. The constant factor can be reduced to 45 by noticing that single-round-node subtrees and square nodes come in pairs along the path, and j such pairs cause at most $2j$ extra rotations in passes one and two. □

The D-unions that occur in pass four are more expensive here than in the bridge-block algorithm, because there may be as many as three nodes between each deferred link, each of which has its weight increased by the D-unions. Using the definitions of Lemma 8, Section 5, and letting x_i denote the highest node on the path belonging to set S_i, we find that the D-unions change the potential by at most

$$85(\log P_k + 3) + \sum_{0 \le i \le k} [\,15\log(p_i) - 5\lg w(x_i) - 85(3 + \log N_i)\,]$$

After the D-unions in pass four, the path from v to the root contains no deferred links, so we can use Lemma 10 to bound the cost of the remaining splices in pass four and the splay in pass five. Therefore, a bound on the amortized cost of an extended splay is given by

$$160\log N_k + O(1) + \sum_{0 \le i \le k} [(85\log N_i + 28) - 85(3 + \log N_i)]$$

As in the proof of Lemma 9, Section 5, this bound can be used to show that $Link(u,v)$ has amortized cost $O(\log n)$, where n is the virtual size of the tree containing u.

Theorem 4:

Any sequence of m biconnected component operations can be performed in worst-case time $O(m\alpha(m,n))$.

8. General Lower Bounds

Lower bounds for the on-line block and bridge-block problems can be obtained using simple reductions from the disjoint set union problem. Let n be the number of elements and m the number of operations in an instance of disjoint set union. Tarjan [16] gave a lower bound of $\Omega(\alpha(m,n))$ on the amortized time per operation and Blum [2] gave a lower bound of $\Omega(\log n / \log\log n)$ on the worst-case time of a single operation. Both

these lower bounds apply to the class of *separable pointer algorithms* for set union. Fredman and Saks [6] have given an $\Omega(\alpha(m,n))$ bound on the amortized cost per operation in the cell probe model of Yao [24], and is also possible to show an $\Omega(\log n / \log\log n)$ bound on the worst-case cost per operation [M. Fredman, private communication, 1989]. We have discussed both these models in chapter 2.

The reduction from disjoint set union to the on-line block and bridge-block problems is straightforward. For each set element a we create a pair of vertices a_0 and a_1 connected by an edge e_a. To answer a find query, we execute *find block* (a_0) or *find block* (a_0,a_1), depending on whether the reduction is to the bridge-block or block problem, respectively. To unite the sets containing elements a and b, we add an edge between a_0 and b_0 and an edge between a_1 and b_1. Thus each set corresponds to a component that is both bridge-connected and biconnected. We define a separable pointer algorithm for bridge-connectivity or biconnectivity to be an algorithm that uses a linked data structure in which no pointer connects the subgraphs representing each graph component. If we begin with a separable algorithm for bridge-connectivity or biconnectivity, the above reduction gives a separable pointer algorithm for disjoint set union. Similarly, the reduction can be used to give a cell probe algorithm for disjoint set union.

We can also give a reduction of disjoint set union to the variant of the block problem in which the graph is initially connected. The initial graph resembles a wheel with hub vertex h. For each element a there is a vertex v_a and an edge connecting v_a to h. Queries are answered by executing *find block* (h,v_a). To unite the sets containing a and b, an edge is added between v_a and v_b. The algorithm given by this reduction does not fit the separable pointer machine model, but the cell probe lower bounds apply. For the variant of the bridge-block problem in which G is initially connected, we know only the trivial lower bound of $\Omega(1)$ on the time per operation.

9. Remarks.

One major difference between the dynamic tree data structure of Sleator and Tarjan and the link/condense tree data structure presented here is that in the latter the time to link together two trees is dependent only on the size of the child tree. If condensation is not required, the tree can be modified to implement all the standard dynamic tree operations, such as *find min* and *add cost*, in time $O(\log n)$, while still allowing fast linking.

Such a tree would be suitable for any tree-based algorithm in which a recurrence relation similar to that of the bridge-block or block algorithm arises.

This paper leaves open the problem of maintaining the triconnected components of a graph undergoing edge insertion, and it leaves a gap in the table of Section 1 between the upper and lower bounds for the bridge-block problem on an initially connected graph. We also leave open the problem of implementing any kind of edge deletion in time $o(n)$ per operation. Reif [12] introduces the notion of *complete dynamic problems*, i.e., a collection of problems with the property that if one problem can be solved in $o(n)$ time per operation, then all the problems can be solved in $o(n)$ time per operation. Examples of these problems include acceptance by a linear-time Turing machine, the Boolean circuit evaluation problem, and the depth-first search numbering of a graph. These problems seem to have no better on-line solution than simply running a linear-time off-line algorithm whenever the input instance changes. One can ask whether bridge-connectivity and biconnectivity are complete dynamic problems.

References for Chapter 4

[1] B. Awerbuch and Y. Shiloach, "New connectivity and MSF algorithms for shuffle-exchange network and PRAM," *IEEE Trans. on Computers* C-36 (1987), pp. 1258-1263.

[2] N. Blum, "On the single-operation worst-case time complexity of the disjoint set union problem," *SIAM J. Comput.* 15 (1986), pp. 1021-1024.

[3] G. A. Cheston, "Incremental algorithms in graph theory," Tech. Rep. No. 91 (PhD. Diss.), Dept. of Computer Science, University of Toronto, (1976).

[4] G. Di Battista and R. Tamassia, "On-Line Planarity Testing," Tech. Rep. No. CS-89-31, Dept. of Computer Science, Brown University (1989).

[5] S. Even and Y. Shiloach, "On-line edge deletion," *J. Assoc. Comp. Mach.* 28 (1981), pp. 1-4.

[6] M. L. Fredman and M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proc. 21st ACM Symposium on Theory of Computing,* Seattle, Washington (May 1989), pp. 345-354.

[7] G. N. Frederickson, "On-line updating of minimum spanning trees," *SIAM J. Computing* 14 (1985), pp. 781-798.

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[9] J. Hopcroft and R. E. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Comm. ACM* 16 (1973), 372-378.

[10] J. Hopcroft and R. E. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Computing* 2:3 (1973).

[11] R. Karp and V. Ramachandran, "Parallel Algorithms for Shared Memory Machines," Tech. Rep. No. CSD 88/408, Dept. of Computer Science, U.C. Berkeley (1988). (To appear in *Handbook of Theoretical Computer Science,* North-Holland.)

[12] J. H. Reif, "A topological approach to dynamic graph connectivity," *Inform. Process. Lett.* 25 (1987), pp. 65-70.

[13] A. Schonhage, "Storage modification machines," *SIAM J. Comput.* 9 (1980), pp. 490-508.

[14] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Comput. Sys. Sci.* 26 (1983), pp. 362-391.

[15] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc. Comput. Mach.* 32 (1985), pp. 652-686.

[16] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. Sys. Sci.* 18 (1979), pp. 110-127

[17] _____, "Amortized computational complexity," *SIAM J. Alg. Disc. Meth.* 6 (1985), pp. 306-318.

[18] _____, *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[19] _____, "Depth first search and linear graph algorithms," *SIAM J. Computing* 1 (1972), pp. 146-160.

[20] _____, "Efficiency of a good but not linear set union algorithm," *J. Assoc. Comput. Mach.* 22 (1975), 215-225.

[21] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. Assoc. Comput. Mach.* 31 (1984), pp. 245-281.

[22] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Computing* 14 (1985), pp. 862-874.

[23] J. Westbrook and R. E. Tarjan, "Amortized analysis of algorithms for set union with backtracking," *SIAM J. Computing* 18 (1989) pp 1-11.

[24] A. C. Yao, "Should tables be sorted?" *J. Assoc. Comput. Mach.* 28 (1981), 615-628.

5. *Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph*

1. Introduction

Let $G = (V,E)$ be an undirected planar graph with $n = |V|$ and $m = |E|$. Let $w(\mathbf{e})$ be a real-valued weight for each edge **e**. A minimum spanning forest for G is a spanning forest that minimizes the sum of the weights of the spanning edges. A maximum spanning forest is defined analogously. We wish to maintain a representation of the minimum or maximum spanning forest for G while processing on-line a sequence of *change weight* $(\mathbf{e},\Delta x)$ operations. This operation adds real number Δx to the weight of the graph edge **e**. In addition, we wish to support operations that change the structure of G, such as the insertion and deletion of edges and vertices.

Dynamic planar graphs of this kind arise in communication networks, graphics, and VLSI design, and they occur in algorithms that build planar subdivisions such as Voronoi diagrams. The dynamic minimum spanning tree problem has been considered by Spira and Pan [15], Chin and Houck [4], and Frederickson [7]. The best result is that of Frederickson, who gave an algorithm based on ''topology trees'' that runs in $O(\sqrt{m})$ time per operation on general graphs, and $O((\log n)^2)$ time on planar graphs. Frederickson's work also implies that the minimum spanning tree for a general graph being modified on-line by edge additions alone can be maintained in $O(\log n)$ amortized time per operation, although he does not explicitly state this.

In this chapter we present data structures and algorithms that maintain a minimum spanning tree of an edge-weighted subdivision of the plane subject to on-line modifications of the kind listed above. The subdivision is allowed to contain loop edges or multiple edges. Our algorithms run in $O(m)$ space and $O(\log m)$ amortized time per operation, where m is the number of edges in the subdivision. We can maintain a minimum spanning forest of an n-vertex planar graph G in time $O(\log n)$ by using our subdivision algorithms on an embedding of G in the plane.

Our algorithms use the topological properties of the subdivision. To modify the subdivision structure we use a pair of simple primitives from which more complicated operations such as the insertion or deletion of edges can be built. Each minimum spanning tree is maintained with a variant of the dynamic tree data structure of Sleator and Tarjan [13,14] called an *edge-ordered dynamic tree*. This data structure is used to represent free trees in which for each vertex there is a total ordering of the incident edges. It can support much the same variety of operations as Sleator-Tarjan dynamic trees, with the addition of operations to split and condense vertices while preserving the edge ordering. Depending upon the application, this repertoire of operations can be used to test membership of an edge in the spanning forest, determine the spanning tree containing a given vertex, or find the edge of maximum or minimum weight on the tree path between two vertices. The edge-ordered tree also finds use in the on-line planarity testing algorithm of Di Battista and Tamassia [5]. Thus our data structure is fairly general and powerful. The algorithms can be made to run in worst-case time $O(\log m)$ with the biased tree implementation of dynamic trees.

The remainder of this chapter is organized into four sections. In section 2 we discuss the subdivision representation scheme of Guibas and Stolfi [9]. In section 3 we consider the case of a subdivision with fixed structure that is undergoing edge weight changes on-line. We give an $O(\log m)$ time algorithm for this restricted situation. In section 4 we introduce the two primitives used by Guibas and Stolfi to modify planar subdivisions; these primitives provide a conceptually simple way to describe more complicated modification operations. Finally, in section 5 we develop the edge-ordered dynamic tree, and use it to extend the algorithm of section 3 to run in a fully dynamic setting.

2. Planar Subdivisions and Their Representation

A subdivision of the plane S is a connected set of vertices and edges that partition the plane into a collection of faces. S may have loop edges or multiple edges between vertices. We are interested only in the topology of S, i.e., the incidence relations between vertices, edges, and faces, and do not consider the actual geometric positions of the vertices and edges. Let G be a planar graph of n vertices. An embedding of G generates a collection of subdivisions, one for each connected component of the graph. If G is triconnected then the topological structure of its embedding is unique up to mirror image

[10], but in general there are multiple embeddings possible for a given planar graph. The edges and vertices of a subdivision, however, constitute a unique graph or multigraph. Our algorithms maintain subdivisions, and they take advantage of the topological relationships among the subdivision's vertices, edges, and faces. In this section we summarize the concepts and notation that we will use in dealing with subdivisions. They are drawn primarily from the work of Guibas and Stolfi in reference [9].

Each undirected edge $\mathbf{e} = \{u,v\}$ of the subdivision S can be directed in two ways. If e is the directed version of $\mathbf{e}$ originating in u and terminating in v, then $sym(e)$ is the version of $\mathbf{e}$ directed from v to u. Note that if $\mathbf{e}$ is a loop edge u and v are identical, but we may still define e and $sym(e)$ as oppositely directed versions of the same undirected edge. The operator $orig(e)$ gives the vertex at which directed edge e originates, and makes it convenient to use directed edges to specify vertices of G. Note that since the plane is orientable, it is possible to define in a consistent way the left and right faces to which a directed edge e is adjacent.

Using the topological incidence relationship between edges and faces of S, we define the **dual graph** $G^* = (F,E^*)$ [6,9,12]. Each face of S gives rise to a vertex in F. Dual vertices f_1 and f_2 are connected by a dual edge $\mathbf{e}^*$ whenever primal edge $\mathbf{e}$ is adjacent to the faces of S corresponding to f_1 and f_2. Note that G^* can be embedded in the plane by placing each dual vertex inside the corresponding face of S, and placing dual edges so that each one crosses only its corresponding primal edge. This embedding is called the dual subdivision S^*. In simpler terms, the dual of a subdivision is given by exchanging the roles of faces and vertices, and S and S^* are each other's dual. Figure 1 gives an example of a subdivision and its dual.

As in the primal subdivision, each undirected dual edge generates two directed edges of S^*; the *sym* and *orig* operators are extended to these dual directed edges. The operator $rot(e)$ gives the dual directed edge that originates in the right face of e and terminates in the left face, i.e. it is e rotated 90° counterclockwise. Similarly $rot^{-1}(e)$ is the directed dual edge from the left face of e to the right face of e, i.e. edge e rotated 90° clockwise. Note that $rot(rot(e)) = sym(e)$ and $(sym(rot(e)) = rot^{-1}(e)$. For a given undirected edge $\mathbf{e}$ in the primal subdivision S, we specify the two pairs of primal and dual directed edges as e_0, e_1, e_2, e_3, where e_0 is a primal directed edge and $e_{i+1 \bmod 4} = rot(e_i)$, $0 \le i \le 3$.

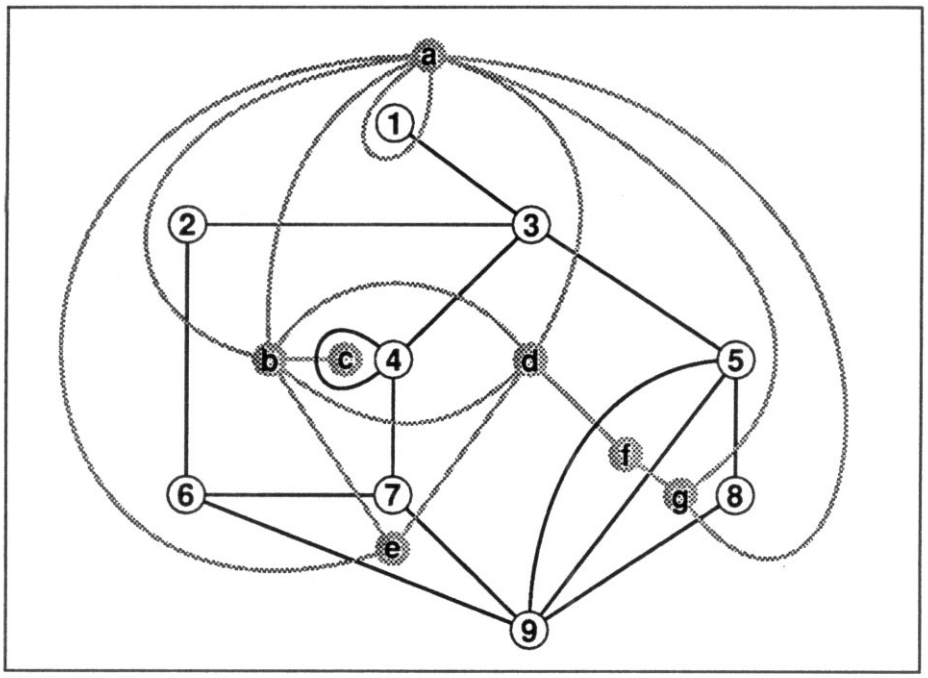


Figure 1: A subdivision (black) and its dual (grey).

Within S and S^* we can unambiguously establish a sense of counterclockwise rotation around a vertex. The notation *next* (e) refers to the edge following e in counterclockwise order around *orig* (e). The **edge ring** of a vertex v is a circular linked list of the directed edges originating at v, organized in counterclockwise order so that *next* (e) is the successor of e in the edge ring. If a vertex v is adjacent to only one undirected edge **e**, then v's edge ring contains the single directed edge e originating at v, and *next* (e) is e. On the other hand, for a loop edge **e** both e and *sym* (e) belong to the edge ring of vertex *orig* (e).

In reference [9], Guibas and Stolfi generalize these concepts to arbitrary two dimensional manifolds, define the notion of an edge algebra on the set of edges and operators, and prove many interesting theorems about such edge algebras. The reader is referred to their paper for more details.

3. Changing Edge Weights Only

In this restricted version of the problem, the structure of S is fixed, but each edge has an associated weight $w(\mathbf{e})$ that is varying as a result of *change weight*$(\mathbf{e},\Delta x)$ operations. We give data structures and algorithms to maintain a minimum spanning tree in the graph induced by the vertices and edges of S. We then apply our algorithms to the maintenance of a minimum spanning forest for a planar graph G undergoing changes in edge weight, using the one-to-one correspondence between the minimum spanning trees for an embedding of G and for G itself.

We work with both S and its dual S^*. For each dual edge we define $w(\mathbf{e}^*) = w(\mathbf{e})$. The following lemma is the basis for the algorithm.

Lemma 1:

Given a spanning tree T for S, let T^* be the set of dual edges $\{\mathbf{e}^* \mid \mathbf{e}$ *is not in* $T\}$. Then T^* is a spanning tree for S^*.

Proof. Given a face f, there must be an edge in T^* that connects it to an adjacent face g. If not, then all the primal edges that bound the face f must be spanned in the primal subdivision. But this implies a cycle in T, which is impossible. This argument can be extended to connected regions of the plane corresponding to subgraphs of T^*, Thus the graph induced by the edges in T^* must be connected. Furthermore, we can apply this argument in the dual to establish that since T is connected by assumption, there cannot be a cycle in T^*. □

Corollary:

T is a minimum spanning tree for S if and only if T^* is a maximum spanning tree for S^*.

Proof. If $w(T)$ is the sum of the weights of the edges in T, and W is the sum of the weights of all edges in S, we have that $W = w(T) + w(T^*)$. Thus $w(T^*)$ is maximized when $w(T)$ is minimized. □

Figure 2 gives an example of the primal and dual spanning trees for the subdivision of Figure 1.

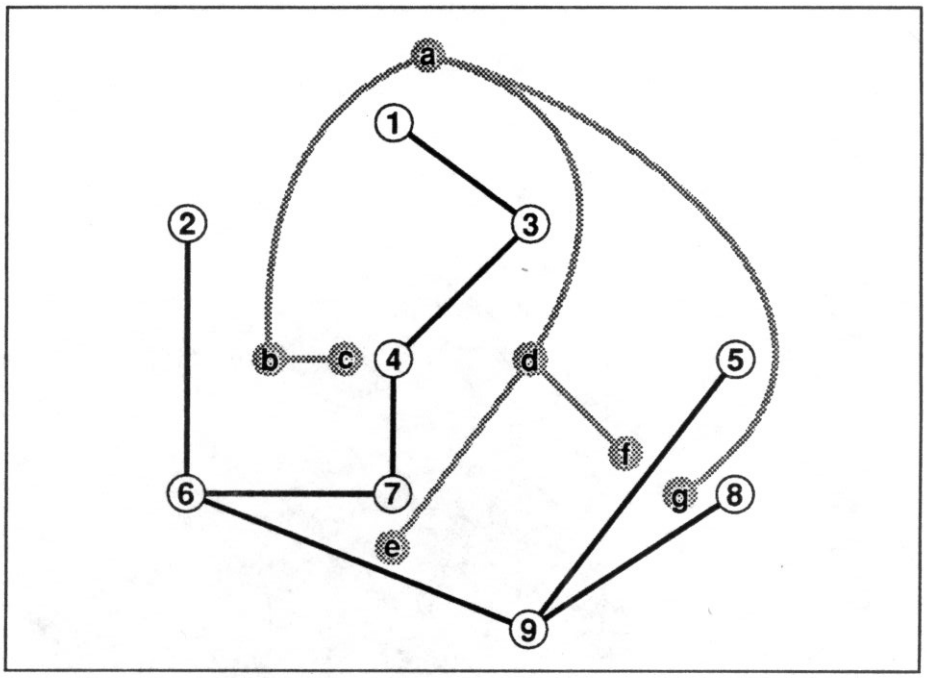


Figure 2: Primal and dual spanning trees for the subdivision of figure 1.

The algorithm maintains T and T^* in tandem. In accordance with lemma 1, an edge **e** is either a spanning edge of T, or its dual $\mathbf{e}^*$ is a spanning edge of T^*. In general, as edge weights change edges are driven out of one tree and their duals are driven into the other. Lemma 1 implies that after a change in edge weight, correct updating of the primal spanning tree automatically results in correct updating of the dual, and vice versa.

To perform the updates efficiently, we utilize the dynamic tree data structure of Sleator and Tarjan [13,14]. Dynamic trees are designed to represent a forest of rooted trees, each node of which has a real-valued cost, under the following operations:

find cost (v): Return the cost of node v.

find root (v): Return the root of the tree containing node v.

find min (v) (*find max* (v)): Return the node of minimum (maximum) cost on the path from v to r, the root of the tree containing v.

add cost (v,x): Add real number x to the cost of every node on the path from v to the root of the tree containing v.

link (v,w): Add an edge from v to w, thereby making v a child of w in the forest. This operation assumes that v is the root of one tree and w is in another.

cut (v): Delete the edge from v to its parent, thereby dividing the tree containing v into two trees.

evert (v): Make v the root of its tree by reversing the path from v to the original root.

find parent (v): Return the parent of v, or null if v is the root of its tree.

find lca (u,v): Return the least common ancestor of nodes u and v.

All the above operations can be performed in $O(\log n)$ amortized time per operation and $O(n)$ space, where n is the number of nodes in the tree.

The vertices of S are represented by dynamic tree nodes with cost $-\infty$. Similarly, the vertices of S^* are represented by dynamic tree nodes with cost $+\infty$. In T the operation *find max* is used, while in T^* the operation *find min* is used. For every edge **e** there is a dynamic tree node $\hat{e}$ of cost $w(\mathbf{e})$. If **e** is a spanning edge of T then there is an edge between the tree node representing the vertex $orig(e_0)$ and $\hat{e}$, and an edge between $\hat{e}$ and the tree node for $orig(e_2)$. Conversely, if $\mathbf{e}^*$ is a spanning edge of T^*, then tree edges join $orig(e_1)$ to $\hat{e}$, and $\hat{e}$ to $orig(e_3)$. Thus **e** is represented by two edges connected through the degree-two node $\hat{e}$. This representation allows *find max* and *find min* on T and T^* respectively to return edges rather than vertices.

For each edge **e**, the five values of $\hat{e}$ and $orig(e_i)$, $0 \le i \le 3$, are stored in the form of pointers to the corresponding dynamic tree nodes. If the subdivision has $O(m)$ edges the number of vertices and faces is also $O(m)$, and so the total space required for the trees is $O(m)$

Now consider the processing of a *change weight* $(\mathbf{e},\Delta x)$ operation. Four cases can occur:

1) **e** is in T and Δx is negative.
2) **e** is not in T ($\mathbf{e}^*$ is in T^*) and Δx is positive.
3) **e** is not in T and Δx is negative.
4) **e** is in T and Δx is positive.

Clearly, cases 1 and 2 have no effect on the spanning trees. Now consider case 3. Since edge **e** has decreased in cost it may now enter the minimum spanning tree T, forcing some other edge out. It is well-known (e.g. see [7,19]) that T is updated by replacing with e a maximum-cost edge **d** in the cycle formed by adding e to T. We can find **d** by executing *evert* (*orig* (e_0)) followed by *find max* (*orig* (e_2). In the special case where the *find max* operation returns $-\infty$, processing terminates immediately. This case occurs when *orig* (e_0) $\equiv$ *orig* (e_2); that is, **e** is a loop edge that can never be a spanning edge, while the dual edge $\mathbf{e}^*$ is a bridge of G^* that must always be a spanning edge.

If $w(\mathbf{e}) \geq w(\mathbf{d})$, no action need be taken. If not, however, then the new minimum spanning tree T is given by deleting edge **d** and inserting edge **e**. Simultaneously, the new maximum spanning tree T^*, is given by deleting edge **e** and inserting edge **d**. This is done by executing the operations:

evert (*orig* (d_0));*cut* ($\hat{d}$);*cut* (*orig* (d_2));

evert (*orig* (e_1));*cut* ($\hat{e}$);*cut* (*orig* (e_3));

followed by:

evert (*orig* (e_0));*link* ($\hat{e}$,*orig* (e_0));*link* (*orig* (e_2),$\hat{e}$);

evert (*orig* (d_1));*link* ($\hat{d}$,*orig* (d_1));*link* (*orig* (d_3),$\hat{d}$);

Since only a constant number of executions of *link*, *cut* and *evert* are required to implemented these modifications, the time for the *change weight* operation is $O(\log n)$ amortized.

Now we consider case 4. Let (V_1,V_2) be a partition of the vertices of G. The cut induced by (V_1,V_2) is the set of edges of G with one endpoint in V_1 and the other in V_2. Again, it is well-known that in case 4, T is updated by replacing **e** with a minimum-cost edge in the cut induced by the partition (V_1,V_2) where V_1 and V_2 are the vertices of the connected components of T created by the removal of edge **e**. Given only the primal tree, this cut edge is hard to find. The utility of the dual spanning tree becomes clear, however, when it is observed that case 4 is the equivalent in the dual tree of case 3 in the primal tree. A dual edge not in T^* has increased in cost, and may therefore force a dual edge out of T^*. The same processing as in case 3 can be applied, interchanging the role of dual and primal tree, and using *find min* rather than *find max*. Thus case 4 can also be handled in time $O(\log n)$.

Theorem 1:

Let S be a subdivision of the plane undergoing on-line changes in edge weight. The minimum spanning tree of S can be maintained in $O(\log m)$ amortized time per operation and $O(m)$ space, where m is the number of edges.

Let G be a planar graph of n vertices undergoing changes in edge weight. An embedding can be generated in $O(n)$ time using one of the algorithms of Hopcroft and Tarjan [11] or Booth and Lueker [1] (see Chiba, Nishizeki, Abe, and Ozawa [3]). Each connected component gives rise to a planar subdivision. The initial spanning trees can be found in $O(n)$ time with the algorithm of Cheriton and Tarjan [2]. Thus, given $O(n)$ preprocessing time, the result of Theorem 1 can be used to maintain the minimum spanning forest of G in $O(\log n)$ amortized time per operation and $O(n)$ space.

4. Subdivisions Undergoing Structure Modifications

In this section we discuss the implementation of dynamic operations that affect the topological structure of planar subdivisions. Following Guibas and Stolfi [9], we supply two modification primitives, *make-edge* and *splice*, that can be used to build more complicated dynamic operations. Typical choices for such operations might be insertion and deletion of edges and vertices, but the primitives are very flexible and allow other possibilities. As before, we may choose to use our subdivision algorithms to represent planar graphs, but since embeddings are not unique for a given graph G, a situation may occur in which an edge cannot legally be inserted into the subdivision, even though G with the new edge remains planar. Therefore our algorithms do not give full generality in the dynamic operations permitted on planar graphs.

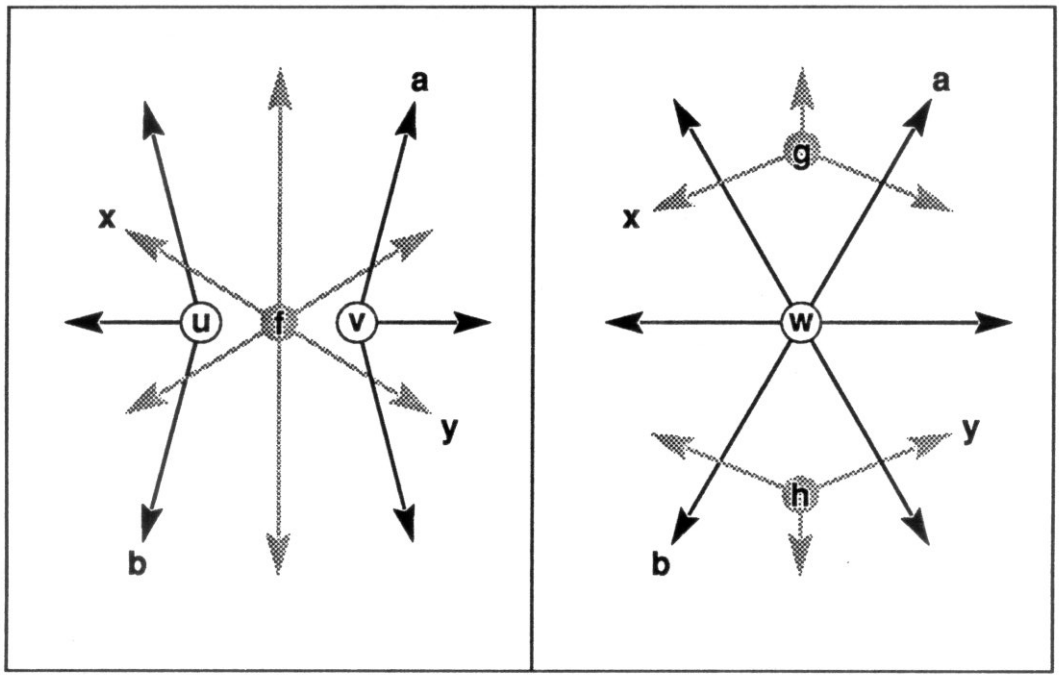
In general, we maintain a collection of subdivisions, each of which is thought of as lying in a distinct plane. We simultaneously maintain each subdivision's dual. The directed edges of the subdivisions are the basic unit for specifying modifications. The edge rings are explicitly maintained, and a vertex or face is referenced by a directed edge in its edge ring. The *make-edge* primitive, which takes no parameter, returns the directed version of the new single edge. The edge and its endpoints form a new subdivision that is embedded along with its dual in a new plane. The *make-edge* primitive has an inverse, *destroy-edge* (e), which takes as an argument an edge that is guaranteed to be

disconnected. The edge is destroyed and the storage is released.

The second primitive is *splice* (e_1,e_2), where e_1 and e_2 are directed edges of the primal subdivision. Splice operates on the vertices *orig* (e_1) and *orig* (e_2), and independently on the dual vertices corresponding to the left faces of e_1 and e_2, which are given by *orig* $(rot^{-1}(e_1))$ and *orig* $(rot^{-1}(e_2))$. If the edges originate in the same vertex then the splice operation splits that vertex in two, with the edges clockwise from e_1 to e_2 going to one of the halves, while the remaining edges go to the other. If the edges have different origins, then the two vertices are combined into one by inserting the edge ring of one vertex into the edge ring of the other. Figure 3 gives an example. Let $\varepsilon_1 = rot\,(next(e_1))$ and $\varepsilon_2 = rot\,(next(e_2))$. The splice is performed by simply exchanging the values of *next* (e_1) and *next* (e_2), while simultaneously exchanging the values of *next* (ε_1) and *next* (ε_2)

The values given by the *next*, *orig*, and *rot* operators determine incidence relations between the faces, edges, and vertices of S. In turn, these incidence relations determine the topology of the surface that S subdivides. Since *splice* (e_1,e_2) changes the values of *next* (e_1) and *next* (e_2), the choice of e_1 and e_2 is restricted by the requirement that the result of the splice remain a subdivision of the plane. Any splice is allowed in which e_1 and e_2 have the same origin or left face, because the splitting of a vertex in either the primal or dual preserves planarity, and if one subdivision remains planar then its dual must also remain planar. If both the origins and the left faces differ, however, and the two edges are contained in the same subdivision, then the splice is disallowed. Such a splice increases by one the genus of the surface that S subdivides. On the other hand, if the edges lie in different subdivisions, i.e. different planes, the splice is allowed. In this case, the splice merges the two subdivisions so that they are contained in a single surface. Given S, it is always possible to draw a subdivision that is topologically equivalent to S but in which some specified edge or vertex is adjacent to the exterior face. Thus the splice of edges contained in different subdivisions can be thought of as redrawing the subdivisions to place the edges on the exteriors, and then plugging the subdivisions together at these edges' origins. The validity of a *splice* or *destroy-edge* operation can be tested using the data structure we present in the next section.

Let S be a subdivision containing m edges. Any undirected edge **e** can be removed from S by taking one of its directed versions e and executing *splice* $(e,next^{-1}(e))$ and



(a) (b)

Figure 3: a) Example of edge rings. Primal vertices u and v lie on the boundary of face f. b) Edge rings and topology produced by executing *splice* (a,b) (or equivalently, *splice* (x,y)) on edge rings of 3a.

splice $(sym(e), next^{-1}(sym(e)))$. Thus a sequence of $O(m)$ splices breaks S into m single edges. Since splice is reversible (in fact, splice is its own inverse), we may conclude that the operations *make-edge* and *splice* are sufficient to generate any planar subdivision. Furthermore, we see how to use *make-edge* and *splice* to implement more complicated dynamic operations. For example, *insert-edge* (e_1,e_2), which inserts an edge between *orig* (e_1) and *orig* (e_2), dividing the face to the left of e_1 and e_2, can be implemented by $e =$ *make-edge* followed by *splice* (e_1, e) and *splice* $(e_2, sym(e))$. Let G denote the planar multigraph induced by the vertices and edges of a collection of subdivisions. Each subdivision induces a connected component of G. We may use *make-edge* and *splice* to generate any planar graph G.

5. Edge-ordered Trees and the Fully Dynamic Algorithm.

In this section we develop the edge-ordered dynamic tree, a data structure designed to handle efficiently splices and the resultant cutting and linking of edge rings. An edge-ordered tree is a general rooted tree in which a total order is imposed on the edges adjacent to a given node (including the parent edge). The ordered set of edges adjacent to node v is called the **edge list** for v. For example, in our application we will use the counterclockwise ordering of the edges around the vertex in the current graph embedding. Each node v in the tree has a real-valued cost, $cost(v)$. The edge-ordered tree supports the following collection of operations:

Link (v,w): Add an edge e from v to w, thereby making v a child of w in the forest. The new edge is inserted at the end of v's edge list and at the front of w's. Return e.

Split (v,e): Split node v into two nodes. If $\alpha e \beta$ is the ordered list of edges adjacent to v then αe becomes the ordered list of edges adjacent to one new node while β becomes the ordered list adjacent to the other. The two nodes have the same cost as v.

Merge (u,v): Merge nodes u and v into a single node. If α is the ordered list of edges for u and β is the ordered list of edges for v then $\alpha\beta$ is the ordered list of edges for the merged node. Nodes u and v must have the same initial cost.

Cycle (v,e): Cyclically permute the order of edges adjacent to v so that e is the last edge in the order. The initial ordered list $\alpha e \beta$ becomes $\beta \alpha e$.

Evert (v): Make v the root of its tree by reversing the path from v to the original root.

Find cost (v): Return the cost of node v.

Find root (v): Return the root of the tree containing node v.

Add cost (v,x): Add real number x to the cost of node v. (Note: does **not** add cost to nodes on the path to the root).

Cut (v): Break the tree into two fragments by deleting the edge between v and its parent.

Find min(v)(*Findmax*(v)): Return the node of minimum (maximum) cost on the path from v to r, the root of the tree containing v.

Find parent (v): Return the parent of v, or null if v is the root of its tree.

Find lca (u,v): Return the least common ancestor of nodes u and v.

Find nodes (e): Return the nodes to which edge e is incident.

To implement the edge-ordered tree we do not create a completely new data structure; rather we show how to transform any given tree T into a new tree T'. Each node v of T is expanded into a collection of subnodes called a **node path**. Each subnode has a cost that is always set equal to *cost* (v). There is one subnode in the node path v for every edge e in the edge list of v. The subnode for e is connected by a tree edge to the subnodes of its predecessor and successor in the edge list. The subnodes for the first and last edges in the list are connected only to their successor and predecessor respectively. Pointers to the first and last subnodes are stored in *first* [v] and *last* [v]. Whenever an edge e connects nodes v_1 and v_2 in T, there is an edge in T' between the two subnodes generated by e in the node paths of v_1 and v_2. Pointers to the two subnodes are stored in *endpoints* [e]. If T has n nodes and hence $n-1$ edges, then T' has $2n-2$ nodes. Note that every node in T' has degree at most three.

The transformed tree T' is maintained with a standard Sleator-Tarjan dynamic tree. The node path for node v has the property that if *evert* (*last* [v]) is performed, then the ordered sequence of nodes on the tree path between *first* [v] and *last* [v] corresponds exactly to the ordered sequence of edges in the edge list from first to last. This property allows the processing of all the edge-ordered tree operations with only a constant number of dynamic tree operations. If we only need to perform the operations *Link* through *Add cost*, the dynamic tree suffices. To perform *Cut*, the node paths must also be threaded into a doubly-linked list, and to perform *Find min*, *Find parent*, *Find lca*, and *Find nodes* auxiliary balanced trees are required. We begin by giving implementations of the edge-ordered tree operations in the first group. For convenience, we will use the notation e to represent both an edge and the corresponding tree subnodes.

Link (u,v) **begin**
    create a new edge e and two new subnodes e_1 and e_2; set *endpoints* [e] = $\{e_1, e_2\}$.
    evert (*last* [u]);
    link (u,e_1); *link* (e_1,e_2); *link* (e_2,*first* [v]);
    set *last* [u] = e_1; set *first* [v] = e_2;
end

Split (v,e) **begin**
Create two new nodes v_1 and v_2;
evert (*last* $[v]$);
Let e_1 and e_2 be the endpoints of e.
If *find parent* $(e_1) = e_2$ set $x = e_2$ else set $x = e_1$
set y = *find parent* (x);
cut (x);
set *first* $[v_1]$ = *first* $[v]$; set *last* $[v_1] = x$;
set *first* $[v_2] = y$; set *last* $[v_2]$ = *last* $[v]$;
destroy node v;
end

Merge (u,v) **begin**
Create new node $\overline{v}$
evert (*last* $[u]$);
link (*last* $[u]$, *first* $[v]$)
set *first* $[\overline{v}]$ = *first* $[u]$; set *last* $[\overline{v}]$ = *last* $[v]$;
destroy nodes u and v;
end

Cycle (v,e) **begin**
evert (*last* $[v]$);
Let x be the subnode of e in v's edge list (as in Split);
set y = *find parent* (x);
cut (x);
link (*last* $[v]$, *first* $[v]$);
set *first* $[v] = y$; set *last* $[v] = x$;
end

Addcost (v, x) **begin**
evert (*last* $[v]$);
add cost (*first* $[v]$, x)
end

The remaining operations Evert, Find cost, and Find root are simply implemented by a call to the matching dynamic tree operation, replacing v by *last* $[v]$ in the call. If the tree is to be rooted at node r, then each of the above operations must be followed by a final *evert* (r) to restore the tree to its proper state.

If the operation *Cut* (v) is needed, we thread the node paths into a circular doubly-linked list, storing pointers to the predecessor and successor of subnode e in *pred*$[e]$ and *succ*$[e]$.

Cut (v) **begin**
set $x =$ *find lca* (*first* $[v]$, *last* $[v]$);
set $y =$ *find parent* (x);
cut (x);
evert (*succ* (x)); *cut* (x);
cut (*pred* (x); *link* (*pred* (x), *succ* (x)));
set *succ* (*pred* (x)) = *succ* (x) and *pred* (*succ* (x)) = *pred* (x);
repeat the above three steps for y.
end

We note that in order to maintain the node path linked lists, each *link* or *cut* that occurred in the implementations of the first group of operations must be followed by the appropriate operation on the linked list. In addition, if we wish to reclaim the storage used by deleted edges, the subnodes must contain a pointer to the edge for which they are endpoints.

To include *Find min*(v), *Find parent* (v), *Find lca* (u,v), and *Find nodes* (e) in the repertoire of edge-ordered tree operations, we need the operation *find node* (x), which given subnode x returns the node whose node path contains x. By maintaining each node path in an auxiliary balanced binary tree such as a red-black tree or splay tree (see [18]), *find node* (x) can be answered in $O(\log n)$ time worst-case or amortized depending on the choice of data structure. Again, appropriate insertions, deletion, splits and concatenates must be done in the auxiliary data structure when operations such as link or cut occur in the implementations of the first group of tree operations. The balanced trees mentioned above also support insertions, deletions, splits, and concatenations in $O(\log n)$ time. With *find node*, we implement the latter four operations as follows.

Find min(v) **begin**
return *find node* (*find min*(*first* $[u]$))
end

Find parent (v) **begin**
return *find node* (*find parent* (*find lca* (*first* $[v]$, *last* $[v]$)))
end

Find lca (u,v) **begin**
return *find node* (*find lca* (*first* $[u]$, *first* $[v]$)))
end

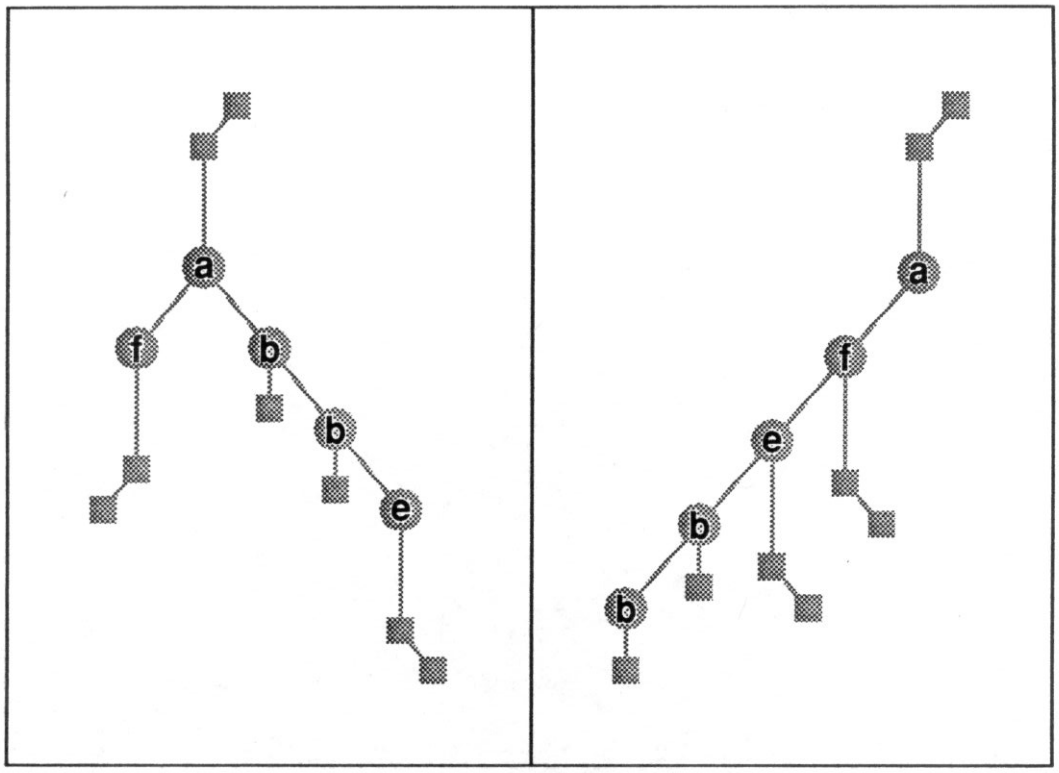
Find nodes (e) **begin**
    let $\{e_1, e_2\}$ be *endpoints* $[e]$;
    return $\{$*find node* (e_1), *find node* $(e_2)\}$;
end

Since each edge-ordered tree operation is implemented using a constant number of dynamic tree operations, the overall amortized running time per operation remains $O(\log n)$.

We now discuss the application of edge-ordered trees to the maintenance of the minimum spanning tree problem. Since we need the *Find min*(v) operation, we use the auxiliary balanced trees for the node paths. Let G denote the multigraph induced by the vertices and edges of a collection of subdivisions, and let G^* denote the graph given by their duals. As in section 3, the vertices of G are represented by tree nodes of cost $-\infty$ and the vertices of G^* by nodes of cost $+\infty$. For each directed version e_i of edge **e** there is a tree node $\hat{e}_i$ of cost $w(\mathbf{e})$. Node $\hat{e}_i$ is made a child of the node corresponding to the vertex *orig* (e_i). The counterclockwise order of the edges around vertex v is used to determine the linear order for the children of the corresponding tree node. If **e** is a spanning edge of G then $\hat{e}_0$ and $\hat{e}_2$ are merged to give a degree-two node connecting the tree nodes for *orig* (e_0) and *orig* (e_1). Similarly, if **e** is a spanning edge of G^*, then $\hat{e}_1$ and $\hat{e}_3$ are merged. There are $m+n$ tree nodes; since the graph is planar the total space required is $O(n)$. Note that loop edges give rise to two tree nodes, one for each directed version of the loop, that are children of the same node. Figure 4 gives an example of a node path.

It is easy to see that the algorithm described above to process *change weight* operations can be slightly modified to apply to the spanning trees with vertex paths. The transfer edge of **e** from one spanning tree to the other is accomplished by splitting the merged vertex in the first tree and merging the two directed edge nodes in the other tree.

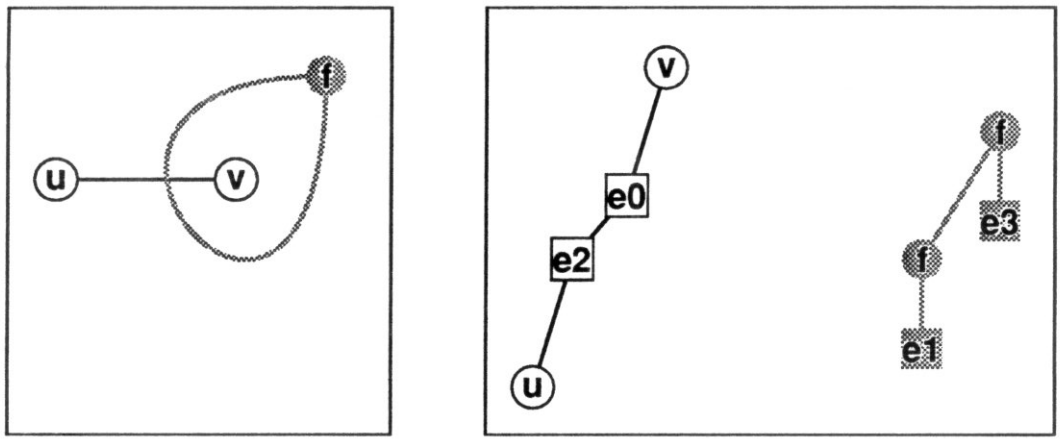
A *make-edge* request creates two new vertices in the primal graph, connected by the new edge **e**. Simultaneously, the dual graph is augmented by a single vertex with the incident loop edge $\mathbf{e}^*$. The primal edge **e** is automatically a spanning edge of G. To satisfy the request, the algorithm allocates storage for a new primal/dual spanning tree pair. The primal tree T consists of two singleton node paths connected through a node that is the merge of $\hat{e}_0$ and $\hat{e}_2$. The dual tree T^* consists of a node path containing two subnodes, with children $\hat{e}_1$ and $\hat{e}_3$. (See figure 5.)



(a) (b)

Figure 4: a) Node path for vertex d of the subdivision of figure 1 and the spanning tree of figure 2. Each subnode is labelled by the vertex to which it is adjacent. $\hat{e}_i$ nodes are shown as unlabelled squares. b) Node path for d after executing *Cycle* $(d, \mathbf{e})$, where $\mathbf{e} = \{d,a\}$.

A *splice* (d,e) operation has more complicated behavior. The most complex situation occurs when d and e have the same origin but distinct left faces (or symmetrically, the same left face but distinct origins). Let δ and ε be the dual directed edges given by $rot(next(d))$ and $rot(next(e))$ respectively. Combining the faces *orig* (δ) and *orig* (ε) into a single face will create a cycle in the spanning tree for the dual graph. This cycle is broken by removing the edge $\mathbf{x}^*$ of minimum weight on the cycle. The algorithm of section 2 for processing a *change weight* request can be used to find $\mathbf{x}^*$. Splitting of the vertex *orig* (d) breaks the primal spanning tree into two fragments. They are joined together by linking in the edge $\mathbf{x}$. Thus the tree modifications caused by the splice are similar to those



(a) (b)

Figure 5: a) Primal (black) and dual (grey) subdivisions produced by $e = make\text{-}edge()$. b) Primal and dual edge-ordered trees for subdivisions of 4a.

occuring if the two halves of the split vertex had been joined by an edge that changed weight from $-\infty$ to $+\infty$. The specific processing follows:

1) Find $\mathbf{x}^*$ using *Evert* and *Find min* in T^*. The node for $\mathbf{x}^*$ is the merge of nodes $\hat{x}_1$ and $\hat{x}_3$. Execute $Split(\mathbf{x}, x_1)$. This breaks T^* into two fragments.

2) Using $Find\ nodes(d)$, determine u such that $u = orig(d)$ and $orig(e)$. Perform $Cycle(u,d)$ and $Split(u,e)$.

3) Using $Find\ nodes(\delta)$, find $f = orig(\delta)$. Using $Find\ nodes(\varepsilon)$, find $g = orig(\varepsilon)$. Perform $Cycle(f,\delta)$, $Cycle(g,\varepsilon)$, and $Merge(f,g)$. (Note: the nodes f and g are initially in different tree fragments.) This produces a single spanning tree T^* for the dual graph.

4) Reconnect the two fragments of T with $Merge(\hat{x}_0,\hat{x}_2)$. This gives a correct tree for the primal graph.

The processing for the other cases of $splice(d,e)$ is simpler. If the edges have the same origin and left faces, then the vertex $orig(d)$ is an articulation point of G, and the splice breaks one subdivision into two subdivisions of distinct surfaces and

correspondingly breaks one component of G into two components. The two fragments into which T is broken by the splice remain valid minimum spanning trees for the new components, since T previously spanned the entire graph, and the fragments were connected only through $orig(d)$. Therefore we need only execute the processing in step 2) above on the primal and dual vertices.

Similarly, if the edges belong to different components, and hence different subdivisions of distinct surfaces, then the splice operation joins the components through a new articulation vertex, $orig(d)$ merged with $orig(e)$. By assumption, the two initial components are correctly spanned, so by combining the two vertices a valid minimum spanning tree for the unified graph is created. Therefore, in this case we need only execute the processing in step 3) above on the primal and dual vertices.

The *make-edge* operation requires constant time, while each splice performs a constant number of edge-ordered tree operations, each of which requires $O(\log n)$ amortized time per operation.

Theorem 2:

The minimum spanning tree of a planar subdivision undergoing both changes in edge weight and changes to its structure can be maintained in $O(\log n)$ amortized time per operation and $O(n)$ space.

We note that given a minimum spanning tree, we can answer connectivity queries such as $find(u,v)$, which asks if vertices u and v are in the same component of G by taking representative subnodes in the vertex paths for u and v and finding the roots of the spanning trees containing then. (This query can be used to check the validity of splice operations.) In fact, the data structure we have presented encodes the entire structure of the subdivisions The entire range of dynamic tree operations described above and in references [13,14] is available for use with the spanning trees, making the overall data structure quite poweful and flexible.

6. Remarks

In implementing edge-ordered tree operations we used balanced trees as auxiliary data structures to maintain the node paths while performing splits and merges. These auxiliary data structures are used primarily to answer *find node* queries in logarithmic time. In fact, Sleator-Tarjan dynamic trees may also be used as the auxiliary data

structures, with the edge lists maintained as a linear branch always rooted at the head node. This suggests that it may be possible to combine the auxiliary functions into the primary dynamic tree and eliminate the auxiliary data structures entirely. We are currently unable to do so, however.

We have assumed that all modification operations are specified by edges. Tamassia [16] gives a data structure for maintaining a dynamic biconnected planar embedding that can test in $O(\log n)$ time whether two vertices u and v lie on a common face. With this auxiliary data structure we can allow modifications to be specified in terms of vertices. For example, we can support *insert_edge* (u,v), which inserts an edge between vertices u and v if they lie on a common face, by using Tamassia's data structure to find the two edges that are adjacent to a common face and have as origins u and v respectively. These edges can then be used as input to *splice*.

Our planar subdivision algorithms can be used to maintain planar graphs, but they have the shortcoming that the modifications permitted are limited by the embedding. Even if one planar graph G_1 can be derived from another G_2 by a single edge addition, a large number of modifications to the subdivision that embeds G_1 may be required to build a subdivision that embeds G_2. In applications of dynamic planar graphs such as vision or chip design, a subdivision of the plane is the basis for the generation of all operations, so a subdivision-based algorithm is not a liability. From a theoretical point of view, however, it would be more satisfying to have an algorithm that allowed the following operations: insert a new vertex; delete a disconnected vertex; delete an edge; and insert an edge if the resultant graph remains planar. If such an algorithm were based on the primal/dual spanning tree relationship, however, then it would need to move quickly (i.e., $O(\log n)$ amortized time) between topologically distinct embeddings. In recent work Di Battisti and Tamassia [5] give data structures and algorithms that can do this in $O(\log n)$ time in the restricted case that only edge insertions are allowed. If a modification primitive powerful enough to allow edge deletions is allowed, however, the problem becomes significantly more difficult, and currently no solution better than repeated application of a static planarity testing tree algorithm is known.

References for chapter 5.

[1] K. Booth and G. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-Tree algorithms," *J. Computer and System Sciences* 13 (1976), 335-379.

[2] D. Cheriton and R. E. Tarjan, "Finding minimum spanning trees," *SIAM J. Computing* 5 (1976), pp. 724-742.

[3] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, "A linear algorithm for embedding planar graphs using PQ-trees," *J. Computer and System Sciences* 30 (1985), 54-76.

[4] F. Chin and D. Houck, "Algorithms for updating minimum spanning trees," *J. Comput. Sys. Sci.* 16 (1978), pp. 333-344.

[5] G. Di Battista and R. Tamassia, "On-line planarity testing," Technical Report No. CS-89-31, Department of Computer Science, Brown University.

[6] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, "Optimal point location in a monotone subdivision," *SIAM J. Computing* 15 (1986), 317-340.

[7] G. N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications", *SIAM J. Computing* 14 (1985), 781-798.

[8] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica* 6 (1986), 109-122

[9] L. J. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams," *ACM Trans. on Graphics* 4 (1985), 74-123.

[10] F. Harary, *Graph Theory,* Addison-Wesley, Reading, MA., 1972, 105.

[11] J. Hopcroft and R. E. Tarjan, "Efficient planarity testing," *J. ACM* 21 (1974), 549-568.

[12] F. P. Preparata and R. Tamassia, "Fully dynamic techniques for point location and transitive closure in planar structures," *Proc. 29th IEEE Symposium on Foundations of Computer Science,* (1988), 558-567.

[13] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Comput. Sys. Sci.* 26 (1983), 362-391.

[14] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc. Comput. Mach.* 32 (1985), 652-686.

[15] P. M. Spira and A. Pan, "On finding and updating spanning trees and shortest paths," *SIAM J. Computing* 4 (1975), pp. 375-380.

[16] R. Tamassia, ''A dynamic data structure for planar graph embedding'' *Proc. 15th Int. Conf. on Automata, Languages, and Programming* (1988), 576-590.

[17] R. E. Tarjan, ''Amortized computational complexity,'' *SIAM J. Alg. Disc. Meth.* 6 (1985), 306-318.

[18] _____, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, 1983, 45-53.

[19] _____, ''Sensitivity analysis of minimum spanning trees and shortest path trees,'' *Information Processing Letters* 14 (1982), pp 30-33.