Sparsification–A Technique for Speeding Up Dynamic Graph Algorithms

DAVID EPPSTEIN

University of California, Irvine, Irvine, California

ZVI GALIL

Columbia University, New York, New York

GIUSEPPE F. ITALIANO

Università "Ca' Foscari" di Venezia, Venice, Italy

AND

AMNON NISSENZWEIG

Tel-Aviv University, Tel-Aviv, Israel

Abstract. We provide data structures that maintain a graph as edges are inserted and deleted, and keep track of the following properties with the following times: minimum spanning forests, graph connectivity, graph 2-edge connectivity, and bipartiteness in time $O(n^{1/2})$ per change; 3-edge connectivity, in time $O(n^{2/3})$ per change; 4-edge connectivity, in time $O(n\alpha(n))$ per change; k-edge connectivity for constant k, in time $O(n\log n)$ per change; 2-vertex connectivity, and 3-vertex connectivity, in time $O(n)$ per change; and 4-vertex connectivity, in time $O(n\alpha(n))$ per change.

A preliminary version of this paper was presented at the 33rd Annual IEEE Symposium on Foundations of Computer Science (Pittsburgh, Pa.), 1992.

D. Eppstein was supported in part by National Science Foundation (NSF) Grant CCR 92-58355.

Z. Galil was supported in part by NSF Grants CCR 90-14605 and CDA 90-24735.

G. F. Italiano was supported in part by the CEE Project No. 20244 (ALCOM-IT) and by a research grant from Università "Ca' Foscari" di Venezia; most of his work was done while at IBM T. J. Watson Research Center.

Authors' present addresses: D. Eppstein, Department of Information and Computer Science, University of California, Irvine, CA 92717, e-mail: eppstein@ics.uci.edu, internet: http://www.ics.uci.edu/~eppstein/; Z. Galil, Department of Computer Science, Columbia University, New York, NY 10027; e-mail: galil@cs.columbia.edu, internet: http://www.cs.columbia.edu/~galil/; G. F. Italiano, Dipartimento di Matematica Applicata ed Informatica, Università "Ca' Foscari" di Venezia, Venezia, Italy, e-mail: italiano@dsi.unive.it, internet: http://www.dsi.unive.it/~italiano/; A. Nissenzweig, Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel.

Further results speed up the insertion times to match the bounds of known partially dynamic algorithms.

All our algorithms are based on a new technique that transforms an algorithm for sparse graphs into one that will work on any graph, which we call *sparsification.*

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms: Algorithms

Additional Key Words and Phrases: Dynamic graph algorithms, minimum spanning trees, edge and vertex connectivity

1. *Introduction*

Graph algorithms are fundamental in computer science, and much work has gone into the study of *dynamic* graph algorithms; that is, algorithms that maintain some property of a changing graph more efficiently than recomputation from scratch after each change. If the changes allowed include both edge insertions and deletions, the algorithm is *fully dynamic*; a *partially dynamic* algorithm only allows insertions. Among many problems that have been studied from this point of view, the minimum spanning tree[1] is perhaps the most important. Other recent work has focused on several types of connectivity.[2]

The main contribution of this paper is a new and general technique for designing dynamic graph algorithms, which we call *sparsification*. We use this technique to speed up many fully dynamic graph algorithms. Roughly speaking, when the technique is applicable it speeds up a $T(n, m)$ time bound for a graph with n vertices and m edges to $T(n, O(n))$; that is, almost to the time needed if the graph were sparse.[3] Sparsification applies to a wide variety of dynamic graph problems, including minimum spanning forests, edge and vertex connectivity. Using the data structure developed for the dynamic maintenance of a minimum spanning forest, we are able to improve previous bounds also for other problems, such as dynamic matroid intersection problems, sampling the space of spanning trees, and computing the k best spanning trees.

The technique itself is quite simple. Let G be a graph with m edges and n vertices. We partition the edges of G into a collection of $O(m/n)$ sparse subgraphs, that is, subgraphs with n vertices and $O(n)$ edges. The information relevant for each subgraph can be summarized in an even sparser subgraph, which we call a *sparse certificate*. We merge certificates in pairs, producing larger subgraphs which we make sparse by again computing their certificate. The result is a balanced binary tree in which each node is represented by a sparse certificate. Each update involves $\log(m/n)$ graphs with $O(n)$ edges each, instead of one graph with m edges. A more careful partition into subgraphs causes each update to involve a sequence of graphs with $O(n)$ edges total.

We develop three variants of our sparsification technique. We use the first variant in situations where no previous fully dynamic algorithm was known. We

[1] See, for example, Eppstein [1994a], Eppstein et al. [1992], Frederickson [1985], and Gabow and Stallman [1985].

[2] See, for example, Di Battista and Tamassia [1990], Frederickson [1997], Galil and Italiano [1992; 1991a; 1991b], Kanevsky et al [1991], La Poutré [1991], and Westbrook and Tarjan [1992].

[3] Throughout this paper log x stands for $\max(1, \log_2 x)$, so $\log(x)$ is never smaller than 1.

use a static algorithm to recompute a sparse certificate in each tree node affected by an edge update. If the certificates can be found in time $O(m + n)$, this variant gives time bounds of $O(n)$ per update.

In the second variant, we maintain certificates using a dynamic data structure. For this to work, we need a *stability* property of our certificates, to ensure that a small change in the input graph does not lead to a large change in the certificates. This variant transforms time bounds of the form $O(m^p)$ into $O(n^p)$.

In the third variant, we perform deletions as in the first variant, and insertions using a partially dynamic algorithm. This leads to insertion times often matching those of known partially dynamic algorithms, together with deletion times similar to those of the first variant.

The remainder of the paper consists of nine sections. Section 2 lists the improved bounds obtained by applying our sparsification technique. In Section 3, we describe an abstract version of sparsification. We first show in Section 4 how sparsification produces a new data structure for maintaining a minimum spanning forest and the connected components of a graph. The same data structure finds applications in other problems, such as finding the k smallest spanning trees of a graph, sampling spanning trees and maintaining a color-constrained minimum spanning tree. Next, dynamic edge and vertex connectivity problems are considered in Sections 5 and 6, respectively. We then discuss an application of sparsification to the problem of maintaining information about the bipartiteness of a graph in Section 7. In Section 8, we improve some of our bounds for edge insertions. Finally, Section 9 contains some concluding remarks, and Section 10 lists some open questions.

2. *New Results*

We describe algorithms and data structures for the following problems:

—We maintain the minimum spanning forest of an edge-weighted graph in time $O(n^{1/2})$ per edge insertion, deletion, or weight change, improving a longstanding $O(m^{1/2})$ bound [Frederickson 1985]. We can instead take time $O(\log n)$ per insertion and $O(n \log(m/n))$ per deletion, strengthening known partially dynamic algorithms.

—We give a data structure for querying membership in the connected components of a graph, in $O(n^{1/2})$ time per edge insertion or deletion, and $O(1)$ time per query. As with minimum spanning forests, the best previous bound was $O(m^{1/2})$ [Frederickson 1985].

—We give a fully persistent, fully dynamic data structure for maintaining the best swap in the minimum spanning tree of a graph, in time $O(n^{1/2})$ per update. As a consequence, we find the k smallest spanning trees of a graph in time $O(m \log\log^* n + kn^{1/2})$, or randomized expected time $O(m + kn^{1/2})$. The best previous times were $O(m^{1/2})$ per update and $O(m \log\beta(m, n) + km^{1/2})$ total [Frederickson 1997].

—We maintain a color-constrained minimum spanning tree of an edge-colored graph in $O(d^2n^{1/2} + d^{11/3}(d!)^2n^{1/3} \log n)$ time per update, where d is the total number of colors. The best previous time bound was $O(d^2m^{1/2} + d^{11/3}(d!)^2n^{1/3} \log n)$ [Frederickson and Srinivas 1989]. For fixed d, the improvement is from $O(m^{1/2})$ to $O(n^{1/2})$.

—We use a randomly weighted version of our minimum spanning forest algorithm to perform a certain random walk among the spanning trees of a graph in time $O(n^{1/2})$ per step. Feder and Mihail [1992] prove that this walk can be used to sample the space of spanning trees in time $O((n \log m + \log \epsilon^{-1})m^{1/2}n^3)$; we improve this to $O((n \log m + \log \epsilon^{-1})n^{7/2})$.

—We maintain the 2-edge-connected components of a graph, in time $O(n^{1/2})$ per update, and $O(\log n)$ per query, improving the previous $O(m^{1/2})$ bound per update [Frederickson 1997].

—We maintain the 3-edge-connected components of a graph, in $O(n^{2/3})$ time per update or query, improving the previous $O(m^{2/3})$ bound [Galil and Italiano 1991a].

—We maintain 4-edge-connected components of a graph in $O(n\alpha(n))$ time. No fully dynamic algorithm was known for this problem. The best static algorithm takes time $O(m + n\alpha(n))$ [Galil and Italiano 1991c; Kanevsky et al. 1991].

—For any fixed k, we maintain information about the k-edge-connectivity of an undirected graph in time $O(n \log n)$ per update. No dynamic algorithm for this problem was known, even for insertions only. The best known static algorithm takes time $O(m + n \log n)$ [Gabow 1991b; Gabow 1991a].

—We maintain the connected, 2- and 3-edge-connected, and 2- and 3-vertex connected components of a graph in time $O(\alpha(q, n))$ per insertion or query and time $O(n \log(m/n))$ per deletion, strengthening known partially dynamic algorithms.[4] Rauch [1995] has recently discovered a fully dynamic algorithm for 2-vertex connectivity, which takes $O(1)$ time per query and $O(m^{2/3})$ amortized time per update. Her bound and our bound are incomparable. No previous fully dynamic algorithm for 3-vertex connectivity was known. All the static problems can be solved in time $O(m + n)$ [Hopcroft and Tarjan 1973; Tarjan 1972].

—We maintain the 4-vertex-connected components of a graph in $O(n\alpha(n))$ time per update, or in time $O(\log n)$ per insertion and $O(n \log n)$ per deletion. No fully dynamic algorithm was previously known. The static problem can be solved in time $O(m + n\alpha(n))$ [Kanevsky et al. 1991].

—We maintain a bipartition of a graph, or determine that the graph is not bipartite, in $O(n^{1/2})$ time per update. Alternately, we achieve $O(\alpha(n))$ amortized time per insertion and $O(n)$ time per deletion. No dynamic algorithm was previously known. The static problem can be solved in time $O(m + n)$.

3. *Sparsification*

We first describe an abstract version of our sparsification technique. Our technique is based on the concept of a *certificate:*

Definition 3.1. For any graph property $\mathcal{P}$, and graph G, a certificate for G is a graph G' such that G has property $\mathcal{P}$ if and only if G' has the property.

[4] See for example, DiBattista and Tamassia [1990], Galil and Italiano [1993], La Poutré [1991], and Westbrook and Tarjan [1992].

Cheriyan et al. [1993] use a similar concept, however they require G' to be a subgraph of G. We do not need this restriction. However, Definition 3.1 allows trivial certificates: G' could be chosen from two graphs of constant complexity, one with property $\mathcal{P}$ and one without it.

Definition 3.2. For any graph property $\mathcal{P}$, and graph G, a *strong certificate* for G is a graph G' on the same vertex set such that, for any H, $G \cup H$ has property $\mathcal{P}$ if and only if $G' \cup H$ has the property.

In all our uses of this definition, G and H will have the same vertex set and disjoint edge sets. A strong certificate need not be a subgraph of G, but it must have a structure closely related to that of G. The following facts follow immediately from Definition 3.2.

Fact 1. Let G' be a strong certificate of property $\mathcal{P}$ for graph G, and let G'' be a strong certificate for G'. Then G'' is a strong certificate for G.

Fact 2. Let G' and H' be strong certificates of $\mathcal{P}$ for G and H. Then $G' \cup H'$ is a strong certificate for $G \cup H$.

A property is said to have *sparse certificates* if there is some constant c such that for every graph G on an n-vertex set, we can find a strong certificate for G with at most cn edges.

3.1. SPARSIFICATION TREE. The other ingredient of our algorithm is a *sparsification tree*. In the conference version of our paper, this was simply a balanced tree with $O(m/n)$ leaves, each holding an arbitrary $O(n)$-edge subgraph of the original graph. Each internal node corresponds to a subgraph formed by the union of edges at descendant leaves. We improve that method somewhat here, by using a technique similar to the *2-dimensional topology tree* of Frederickson [1985; 1997] to partition the edges in a way that induces subgraphs with few vertices.

We start with a partition of the vertices of the graph, as follows: we split the vertices evenly in two halves, and recursively partition each half. Thus, we end up with a complete binary tree in which nodes at distance i from the root have $n/2^i$ vertices.

We then use the structure of this tree to partition the edges of the graph. For any two nodes α and β of the vertex partition tree at the same level i, containing vertex sets V_α and V_β, we create a node $E_{\alpha\beta}$ in the edge partition tree, containing all edges in $V_\alpha \times V_\beta$. The parent of $E_{\alpha\beta}$ is $E_{\gamma\delta}$, where γ and δ are the parents of α and β respectively in the vertex partition tree. Each node $E_{\alpha\beta}$ in the edge partition tree has either three or four children (three if $\alpha = \beta$, four otherwise).

We use a slightly modified version of this edge partition tree as our sparsification tree. The modification is that we only construct those nodes $E_{\alpha\beta}$ for which there is at least one edge in $V_\alpha \times V_\beta$. If a new edge is inserted new nodes are created as necessary, and if an edge is deleted, those nodes for which it was the only edge are deleted.

LEMMA 3.1.1. *In the sparsification tree described above, each node $E_{\alpha\beta}$ at level i contains edges inducing a graph with at most $n/2^{i-1}$ vertices.*

PROOF. There can be at most $n/2^i$ vertices in each of V_α and V_β. □

3.2. Basic Sparsification. We say a time bound $T(n)$ is *well behaved* if for some $c < 1$, $T(n/2) < cT(n)$. We assume well-behavedness to eliminate strange situations in which a time bound fluctuates wildly with n. All polynomials are well behaved. Polylogarithms and other slowly growing functions are not well behaved, but since sparsification typically causes little improvement for such functions, we will generally assume all time bounds to be well behaved.

Our main result is the following:

Theorem 3.2.1. *Let $\mathcal{P}$ be a property for which we can find sparse certificates in time $f(n, m)$ for some well behaved f, and such that we can construct a data structure for testing property $\mathcal{P}$ in time $g(n, m)$ which can answer queries in time $q(n, m)$. Then there is a fully dynamic data structure for testing whether a graph has property $\mathcal{P}$, for which edge insertions and deletions can be performed in time $O(f(n, O(n))) + g(n, O(n))$, and for which the query time is $q(n, O(n))$.*

Proof. We maintain a sparse certificate for the graph corresponding to each node of the sparsification tree. The certificate at a given node is found by forming the union of the certificates at the three or four child nodes, and running the sparse certificate algorithm on this union. As shown in Lemmas 3.1.1 and 4.1 the certificate of a union of certificates is itself a certificate of the union, so this gives a sparse certificate for the subgraph at the node. Each certificate at level i can be computed in time $f(n/2^{i-1}, O(n/2^i))$. Each update will change the certificates of at most one node at each level of the tree. The time to recompute certificates at each such node adds in a geometric series to $f(n, O(n))$.

This process results in a sparse certificate for the whole graph at the root of the tree. We update the data structure for property $\mathcal{P}$, on the graph formed by the sparse certificate at the root of the tree, in time $g(n, O(n))$. The total time per update is thus $O(f(n, O(n))) + g(n, cn)$. $\square$

This technique is very effective at producing dynamic graph data structures for a multitude of problems, in which the update time is $O(n \log^{O(1)} n)$ instead of the static time bounds of $O(m + n \log^{O(1)} n)$.

3.3. Stable Sparsification. To achieve sublinear update times, we further refine our sparsification idea.

Definition 3.3.1. Let A be a function mapping graphs to strong certificates. Then A is stable if it has the following two properties:

(1) For any graphs G and H, $A(G \cup H) = A(A(G) \cup H)$.
(2) For any graph G and edge e in G, $A(G - e)$ differs from $A(G)$ by $O(1)$ edges.

Informally, we refer to a certificate as stable if it is the certificate produced by a stable mapping. The certificate consisting of the whole graph is stable, but not sparse.

Theorem 3.3.2. *Let $\mathcal{P}$ be a property for which stable sparse certificates can be maintained in time $f(n, m)$ per update, where f is well behaved, and for which there is a data structure for property $\mathcal{P}$ with update time $g(n, m)$ and query time $q(n, m)$. Then $\mathcal{P}$ can be maintained in time $O(f(n, O(n))) + g(n, O(n))$ per update, with query time $q(n, O(n))$.*

PROOF. As before, we use the sparsification tree described above. After each update, we propagate the changes up the sparsification tree, using the data structure for maintaining certificates. We then update the data structure for property $\mathcal{P}$ which is defined on the graph formed by the sparse certificate at the tree root.

At each node of the tree, we maintain a stable certificate on the graph formed as the union of the certificates in the three or four child nodes. The first part of the definition of stability implies that this certificate will also be a stable certificate that could have been selected by the mapping A starting on the subgraph of all edges in groups descending from the node. The second part of the definition of stability then bounds the number of changes in the certificate by some constant s, since the subgraph is changing only by a single edge. Thus, at each level of the sparsification tree, there is a constant amount of change.

When we perform an update, we find these s changes at each successive level of the sparsification tree, using the data structure for stable certificates. We perform, at most, s data structure operations, one for each change in the certificate at the next lower level. Each operation produces at most s changes to be made to the certificate at the present level, so we would expect a total of s^2 changes. However, we can cancel many of these changes since, as described above, the net effect of the update will be at most s changes in the certificate.

In order to prevent the number of data structure operations from becoming larger and larger at higher levels of the sparsification tree, we perform this cancellation before passing the changes in the certificate up to the next level of the tree. Cancellation can be detected by leaving a marker on each edge, to keep track of whether it is in or out of the certificate. Only after all s^2 changes have been processed do we pass the at most s uncanceled changes up to the next level.

Each change takes time $f(n, O(n))$, and the times to change each certificate then add in a geometric series to give the stated bound. $\square$

3.4. ASYMMETRIC SPARSIFICATION. Theorem 3.2.1 can be used to dynamize static algorithms, while Theorem 3.3.2 can be used to speed up existing fully dynamic algorithms. In order to apply effectively Theorem 3.2.1, we only need to *compute* efficiently *sparse* certificates, while, for Theorem 3.3.2, we need to *maintain* efficiently *stable sparse* certificates. Indeed stability plays an important role in the proof of Theorem 3.3.2. In each level of the update path in the sparsification tree, we compute s^2 changes resulting from the s changes in the previous level, and then by stability obtain only s changes after eliminating repetitions and canceling changes that require no update. Although in most of the applications we consider stability can be used directly in a much simpler way, we describe it in this way here for sake of generality.

In some applications, we may expect to perform many more insertions than deletions, so it might be appropriate to speed up insertion time even at some expense in the deletion time. One way of doing this would be to use a partially dynamic data structure, that only allows for insertions, and rebuild the data structure from scratch after each deletion. As we now show, sparsification can provide an improvement in this situation as well.

For this method, we define our sparsification tree differently: we group the edges arbitrarily into subgraphs of exactly n edges each, with one *small subgraph* of at most n edges. We then form a balanced binary tree with these subgraphs as leaves. Each internal node corresponds as before to a subgraph formed as the union of its descendant leaves. We can maintain such a sparsification tree as edges are inserted or deleted, as follows: to delete an edge, remove it from its subgraph and replace it with an edge from the small subgraph. To insert an edge, place it in the small subgraph. If a deletion removes the last edge from the small subgraph, another subgraph (one at the greatest depth in the balanced tree) can be designated as small. If an insertion causes the small subgraph to have $n + 1$ edges, it can be split into two subgraphs and some minor modification to the tree will cause it to remain balanced. In this way, each update affects $O(\log(m/n))$ nodes of the tree.

This alternate sparsification tree is simpler than the one defined earlier, but has the disadvantage that certificates at lower levels of the tree do not decrease in size, so using this tree creates an additional $O(\log(m/n))$ factor in time. However, as we now describe, we can use this structure to perform edge deletions in a similar amount of time to Theorem 3.1.1, while allowing insertions to be performed much more quickly.

THEOREM 3.4.1. *Let $\mathcal{P}$ be a property for which we can find sparse certificates in time $f(n, m)$, and such that we can construct a partially dynamic data structure for testing property $\mathcal{P}$ in time $g(n, m)$ which can handle edge insertions in time $p(n, m)$ and answer queries in time $q(n, m)$. Then there is a fully dynamic data structure for testing whether a graph has property $\mathcal{P}$, for which edge insertions can be performed in time $O(f(n, O(n)))/n + g(n, O(n))/n + p(n, O(n))$, edge deletions can be performed in time $f(n, O(n))O(log(m/n)) + g(n, O(n))$, and for which the query time is $q(n, O(n))$.*

PROOF. We construct a sparsification tree as described above. The root of the tree corresponds to a sparse certificate for the entire graph. We maintain a partially dynamic data structure on the graph formed by that certificate. Each inserted edge is not immediately added to the sparsification tree; instead it is inserted in the partially dynamic data structure and added to a list of edges not yet in the tree. If there have been k insertions, the time for this part of the algorithm is $p(n, O(n) + k)$ per insertion. However, k might be larger than n, so we must keep the graph sparse. After each sequence of n consecutive insertions, we perform a sparsification step in the graph formed by the union of the root certificate and the list of inserted edges, and reconstruct the partially dynamic data structure, in time $f(n, O(n)) + g(n, O(n))$, which can be amortized to $f(n, O(n))/n + g(n, O(n))/n$ per insertion.

Whenever we perform a deletion, we also add to the sparsification tree each edge that has been inserted since the previous deletion, emptying the list of edges not yet in the tree. If k such insertions have been made, we will need at most $(k - 1)/n$ new leaves in the sparsification tree, and the number of tree nodes involved as ancestors to those leaves will be $O(k/n + \log(m/n))$. At each node, we compute a sparse certificate in time $f(n, O(n))$. At the root of the tree, we reconstruct the partially dynamic data structure in time $g(n, O(n))$. Thus, the time per deletion is at most $f(n, O(n))O(k/n + \log(m/n)) + g(n, O(n))$. The $f(n, O(n))O(k/n)$ term can be charged as an amortized $O(f(n/O(n))/n)$ time per insertion. □

The amortized bounds can be made worst case by the use of standard techniques, such as performing the sparsification and data structure reconstruction gradually over the course of many insertions.

3.5. BETTER SPACE AND PREPROCESSING. We have seen that sparsification can lead to good time bounds per operation. We now consider the amount of space used by this method. For simplicity, we assume that the certificate computation algorithm takes linear space. Our analysis will apply equally well to the preprocessing time needed to create the data structure.

In the method presented in Theorems 3.1.1 and 3.3.2, each edge is involved in data structures in $O(\log n)$ nodes, so the total space is $O(m \log n)$. However, in Theorem 3.4.1, the space is $O(m)$: this is the number of edges in certificates at the bottom level of nodes of the sparsification tree, but at higher levels the sparsification causes the number of edges to decrease geometrically. As we now show, this space savings can be applied to the other two sparsification methods.

The idea is simple: we use the sparsification tree we described for basic and stable sparsification (which subdivides the graph by its vertices), only down to the level of nodes with $O(n^2/m)$ vertices each. Above that level, the space and preprocessing are proportional to the total number of vertices in all nodes, which sums to $O(m)$. Below that level, we use the other sparsification tree, described in Theorem 3.4.1, for which the space is linear.

THEOREM 3.5.1. *The time bound of Theorem* 3.1.1 *can be achieved with* $O(m)$ *space, plus any additional space needed to construct certificates or compute the graph property on the root certificate. For preprocessing time in Theorem* 3.1.1, *or either space or preprocessing in Theorem* 3.3.6, *if the individual processing or space at a node of the sparsification tree is bounded by* $h(n)$, *the total bound will be* $O(m/n\ h(O(n)))$.

PROOF. The time for computing or updating certificates in the upper levels of the sparsification tree is $O(f(n))$ by Theorem 3.1.1 or 3.3.2. The time in lower levels is $O(f(n^2/m)\log(m^2/n^2))$; with the assumption of well-behavedness this can be rewritten $O(f(n)(n/m)^{\log c} \log(m/n)) = O(f(n))$.

If m changes by more than a constant factor as a result of many insertions or deletions, the boundary between the lower and upper parts of the sparsification tree may shift; we can recompute the new tree in this event without changing the asymptotic amortized time per operation, or perform this reconstruction gradually as in Theorem 3.4.1 to achieve a similar worst case bound.

The bound for space and preprocessing comes from the fact that there are $O(m/n)$ nodes in the tree, with $O(n)$ edges each. □

4. *Minimum Spanning Trees and Connectivity*

Given an undirected graph with nonnegative edge weights, the *minimum spanning forest* is the subgraph of minimum total weight that has the same connected components as the original graph. This subgraph will always be a forest, hence the name. We are interested in maintaining a minimum spanning forest during updates, such as edge insertions, edge deletions, and edge-weight changes. We assume that a minimum spanning forest is given as a list of edges: after each update, we would like to return the new list together with a pointer to the first edge in the list. We should be able to quickly test whether any given edge of the graph is in or out of the list.

A minimum spanning forest is not a graph property, since it is a subgraph rather than a Boolean function. However, sparsification still applies to this problem. Alternately, our data structure maintains any property defined on the minimum spanning trees of graphs. This data structure will also be an important subroutine in some of our later results.

In the static case, it was well known that a minimum spanning forest can be computed by either of two dual greedy algorithms:

Blue rule: Add edges one at a time to the spanning forest until it spans the graph. At each step, find a cut in the graph that contains no edges of the current forest, and add the edge with lowest weight crossing the cut.

Red rule: Remove edges one at a time from the graph until only a forest is left. At each step, find a cycle in the remaining graph and remove the edge with the highest weight in the cycle.

Most classical static minimum spanning tree algorithms use the blue rule, but the recent breakthrough linear time randomized algorithm of Karger et al. [1995] uses a combination of both rules.

We will need the following analogue of strong certificates for minimum spanning trees:

LEMMA 4.1. *Let T be a minimum spanning forest of graph G. Then, for any H, there is some minimum spanning forest of $G \cup H$, which does not use any edges in $G - T$.*

PROOF. If we use the red rule on graph $G \cup H$, we can eliminate first any cycle in G (removing all edges in $G - T$) before dealing with cycles involving edges in H. □

Thus, we can take the strong certificate of any minimum spanning forest property to be the minimum spanning forest itself. Minimum spanning forests also have a well-known property that, together with Lemma 4.1, proves that they satisfy our definition of stability:

LEMMA 4.2. *Let T be a minimum spanning forest of graph G, and let e be an edge of T. Then either $T - e$ is a minimum spanning forest of $G - e$, or there is a minimum spanning forest of the form $T - e + f$ for some edge f.*

This fact is implicit, for instance, in Frederickson's dynamic minimum spanning tree algorithm [1985].

If we modify the weights of the edges, so that no two are equal, we can guarantee that there will be exactly one minimum spanning forest. For each vertex v in the graph, let $i(v)$ be an identifying number chosen as an integer between 0 and $n - 1$. Let ϵ be the minimum difference between any two distinct weights of the graph. Then, for any edge $e = (u, v)$ with $i(u) < i(v)$, we replace $w(e)$ by $w(e) + \epsilon i(u)/n + \epsilon i(v)/n^2$. The resulting MSF will also be a minimum spanning forest for the unmodified weights, since for any two edges originally having distinct weights the ordering between those weights will be unchanged. This modification need not be performed explicitly—the only operations our algorithm performs on edge weights are comparisons of pairs of weights, and this can be done by combining the original weights with the numbers of the vertices involved taken in lexicographic order. The mapping from graphs to unique minimum spanning forests is stable, since part (1) of the definition of stability follows from Lemma 4.1 and part (2) follows from Lemma 4.2.

We base our algorithm on the result of Frederickson [1985], that minimum spanning trees can be maintained in time $O(m^{1/2})$. We improve this bound by combining Frederickson's algorithm with our sparsification technique.

THEOREM 4.3. *The minimum spanning forest of an undirected graph can be maintained in time $O(n^{1/2})$ per update.*

PROOF. Apply the stable sparsification technique of Theorem 3.3.2, with $f(n, m) = g(n, m) = O(m^{1/2})$ by Frederickson's algorithm. □

A more direct and simpler use of stability is possible as the following argument shows. Consider the insertion of a new edge e that is added to a leaf subgraph G_i of the sparsification tree. The constant change in all ancestors of G_i is the following. Let G_j be an ancestor of G_i. Add e to G_j and update the minimum spanning forest of G_j. If there is no change in the minimum spanning forest, then the update caused by e is not propagated up in the sparsification tree. Otherwise, the change is that e enters the minimum spanning forest of G_j, and at most one edge, say f, leaves it. In this case, we insert e into the parent of G_j and update its minimum spanning forest accordingly. So the net update that is percolating in the path from G_i to the root of sparsification tree is given by edge e only. The reason why we do not need to propagate to the parent of G_j the update given by the deletion of f is the following. Let G_k be the parent of G_j. If f was not in the minimum spanning forest of G_k, then deleting f from G_k does not change its minimum spanning forest. If f was in the minimum spanning tree of G_k, then by the red rule inserting e into G_k will cause again a swap with f into its minimum spanning forest. In both cases, propagating the deletion of f from G_j to G_k is superfluous. A similar argument applies in case of edge deletions or edge weight changes rather than edge insertions.

Most minimum spanning forest properties can be maintained in the bounds given by Theorem 4.3 by making use of the dynamic tree data structure of Sleator and Tarjan [1983], with a query time of $O(\log n)$. One such property, namely testing whether vertices are in the same tree of the forest (which is well known to be equivalent to testing connected components of the graph), can be maintained even with an $O(1)$ query time.

COROLLARY 4.4. *The connected components of an undirected graph can be maintained in time $O(n^{1/2})$ per update, and $O(1)$ time per query.*

PROOF. Again, use Theorem 3.3.2 with Frederickson's algorithm. As noted by Frederickson [1997], connectivity queries in his data structure can be performed in constant time. □

Corollary 4.4 shows how our data structure for maintaining a minimum spanning forest of a graph can be used for maintaining information about the connected components of a graph. We next list three other applications of the same data structure.

4.1. BEST SWAPS AND PERSISTENCE.

In any connected graph, the *minimum spanning tree* (in short, *MST*) differs from the second-best spanning tree by the addition of an edge not already in the MST, and the removal of an MST edge. Frederickson [1997] described a *fully persistent* algorithm for maintaining this *best swap*: each update operation is performed in one of a number of *versions* of the data structure, and creates a new such version without modifying existing versions. Frederickson's data structure allows two operations: (1) perform the best swap, and delete the removed MST edge from the graph; and (2) change to $-\infty$ the weight of the MST edge that would be removed by the best swap (forcing that edge to remain in the MST, so that some other swap becomes best). Frederickson uses these operations to find the k smallest spanning trees of a graph in time $O(m \log \beta(m, n) + k \min(k, m)^{1/2})$. We refer the interested reader to reference [Frederickson 1997] for the details of the method. We only mention here that full persistence is crucial for the algorithm, as the ith smallest spanning (i.e., the new version to be created) tree is derived via best swaps from one of the jth smallest spanning trees, $1 \leq j \leq i$ (i.e., one of the previous versions).

THEOREM 4.1. *The best swap of an undirected graph can be maintained with full persistence in time $O(n^{1/2})$ per update.*

PROOF. Theorem 3.3.2 may be easily modified to support full persistence, as long as one knows a persistent version of the data structure used at each node of the sparsification tree. We augment the sparsification tree so that each such node exists in multiple versions. Each version of a node points to a persistent data structure for maintaining the certificate at that node, to another simple persistent data structure for determining which of its edges are in which child, and to appropriate versions of its two child nodes. When we perform an update, we first find the leaf node containing the updated edge by following pointers from the appropriate version of the root of the sparsification tree. We update the certificate data structures at each node, and make new versions of the nodes along the update path pointing to the new versions of the certificate data structures.

Frederickson's data structure can support edge insertions and deletions in $O(m^{1/2})$ time, and the two update types listed above may be simulated using $O(1)$ insertions and deletions. The MST and best swap edge do not form a strong certificate for the best swap, but they are a subgraph of another graph, the union of the MST T_1 of G and the MST T_2 of $G - T_1$, which forms a stable certificate for the best swap problem. These two MST's can be maintained by Frederickson's MST algorithm, which can easily be made persistent using the same techniques as Frederickson's best swap data structure.

Thus, a combination of a persistent version of Frederickson's data structure with a persistent version of sparsification gives the result. □

THEOREM 4.2. *The k best spanning trees of an undirected graph can be found in a total of $O(m \log \log^* n + k \min(n, k)^{1/2})$ time, or by a randomized algorithm taking $O(m + k \min(n, k)^{1/2})$ expected time.*

PROOF. As Frederickson [1997] shows, this problem can be solved using $O(k)$ operations in the persistent best swap data structure. The data structure can be constructed in time $O(m \log \log^* n)$, the bottleneck constructing a minimum spanning tree in each node of the sparsification tree. This can be improved to $O(m)$ expected time by using the recent results of Karger et al. [1995]. A technique of the first author [Eppstein 1992] combined with recent results on sensitivity analysis [Dixon et al. 1992; King 1995] allows us to replace n by $\min(n, k)$ and m by $\min(m, k)$ in the time bounds. □

COROLLARY 4.3. *The k best spanning trees of a Euclidean planar point set can be found in time $O(n \log n \log k + k \min(k, n)^{1/2})$. For a point set with the rectilinear metric, the time is $O(n \log n + n \log \log n \log k + k \min(k, n)^{1/2})$, and for a point set in higher dimensions the time is $O(n^{2-2/(\lceil d/2 \rceil+1)+\epsilon} + kn^{1/2})$.*

PROOF. In an earlier work [Eppstein 1994b], we reduced these geometric problems to the general graph problem in these time bounds. □

4.2. SAMPLING SPANNING TREES. Consider the random walk used by Feder and Mihail [1992] to randomly sample spanning trees of a graph G. The state at each point in the walk consists of a set of edges, which is either a spanning tree of G, or a forest with $n - 2$ edges (i.e., a spanning tree with an edge removed). At each step, we stay with the current edge set with probability 1/2. If we do not stay with the current step, we either remove a randomly chosen edge from the edge set if it is already a spanning tree, or we add an edge chosen randomly among those edges that augment the set to become a spanning tree. Thus, whenever the edge set changes, it alternates between being a spanning tree and a separated forest.

We can implement this random walk if we can perform the following *combined step* efficiently: given a spanning tree of the graph, remove a randomly chosen edge, and then augment the resulting forest to be a spanning tree again by choosing another edge randomly among the edges that cross the cut spanned by the removed edge.

THEOREM 4.2.1. *We can implement the random walk of Feder and Mihail* [1992] *on the spanning forests of a given graph, in time $O(n^{1/2})$ per step.*

PROOF. The input is an unweighted graph, so we are free to assign weights arbitrarily. We do this in such a way that, at each point in time, the edges in the current spanning tree are given weight zero. The edges outside the tree are given weights corresponding to their positions in a permutation chosen uniformly at random among all possible permutations of these edges.

When we remove an edge from the spanning tree, we insert it into this random permutation by choosing its position as a randomly chosen integer from 0 to $m - n$, preserving the relative positions in the permutation of all other edges. We then use our data structure to find a new minimum spanning tree with the weights described above. In this new tree, some edge will have been added to replace the removed edge. The random permutation on the edge weights causes this newly added edge to be selected uniformly at random among the edges spanning the cut. We remove this selected edge from the random permutation and give it weight zero. The removal of this edge will not bias the distribution of permutations of the remaining edges, so the overall permutation will continue to be chosen uniformly at random.

To implement this, we need to compute minimum spanning trees in a graph with weights that may change as edges are inserted to and deleted from the random permutation. Our minimum spanning tree algorithm, and Frederickson's algorithm on which ours is based, do not perform arithmetic on edge weights;

they can operate in the given time bounds if they can merely compare any two weights in constant time. These comparisons can be performed using a data structure of Dietz and Sleator [1987] to find which of the two edges occurs earlier in the permutation. This structure maintains a list of elements, and allows insertions and deletions within the list; we use an auxiliary data structure to determine in $O(\log n)$ time where in the list to perform each insertion. □

Feder and Mihail show that this walk converges to the uniform distribution in a total of $O((n \log m + \epsilon^{-1})n^3)$ steps; thus, we can select a spanning tree from the uniform distribution in time $O((n \log m + \epsilon^{-1})n^{7/2})$.

4.3. Matroid Intersection Problems. Frederickson and Srinivas [1989] considered the problem of maintaining dynamically a solution to a class of matroid intersection problems, including the problem of maintaining a colored-constrained minimum spanning tree of a graph. Before defining the problem, we need a little terminology on matroids [Lawler 1976].

A matroid M consists of a finite set E of elements, plus certain rules describing some properties of a family of subsets of E [Lawler 1976]. For our purposes, it is enough to say that these rules define the notion of *independence* in subsets of E. A matroid can be defined as a family of subsets of E, known as *independent sets*, satisfying the following two axioms:

(1) Any subset of an independent set is independent.
(2) If A and B are both independent, and $|B| > |A|$, there is some $b \in B - A$ such that $A \cup \{b\}$ is independent.

(In Section 7, we will see an example of a matroid defined using these axioms.) A subset of E that is not independent is called *dependent*. A *base* B is a maximal independent subset of E, and the *rank*(A) of a subset $A \subseteq E$ is the cardinality of a maximal independent subset of A. Let B be a base, $e \in B$, and $f \in E - B$. The *circuit* $C(f, B)$ is the set containing all the elements that can be deleted from $B \cup \{f\}$ to restore independence. The *cocircuit* $\bar{C}(e, B)$ is the set containing all the elements that restore rank when added to $B - \{e\}$.

The matroid that we will be using in this section is the graphic matroid that defines minimum spanning trees. Let G be a weighted undirected graph. E is the set of edges in the graph. A base B of the matroid is a spanning tree of G, an independent set is a forest, and the rank is the number of edges in a spanning tree. A circuit is a cycle: $C(f, B)$ is the cycle induced by adding edge f to the spanning tree B (namely, $C(f, B)$ contains all the edges that f can replace in B). A cocircuit is a cut: $\bar{C}(e, B)$ contains all the edges that can replace e in B. A minimum spanning tree is a minimum weight base in a graphic matroid.

Assume that each edge of a graph is labeled with one out of d different colors. We would like to maintain a minimum spanning tree that contains a certain number of edges of each color, as edges are inserted, deleted, or have their weight changed. We call this the *color-constrained minimum spanning tree* problem. Frederickson and Srinivas [1989] gave a very sophisticated algorithm to update a solution of a family of matroid intersection problems, which include the dynamic maintenance of color-constrained minimum spanning trees. Their algorithm makes use of the following three low-level primitives:

maxcirc(f, B): find the maximum weight element in the circuit $C(f, B)$ (assuming that $f \notin B$).
mincocirc(e, B): find the minimum weight element in the cocircuit $\bar{C}(e, B)$ (assuming that $e \in B$).
swap(e, f, B): change B into $B - e + f$ (assuming that $f \notin B$, $e \in C(f, B)$).

Note that when the base B is a spanning tree of a graph, these primitives correspond to the operations used to update a minimum spanning tree in an uncolored graph. Namely, the primitives are finding the maximum weight edge in the cycle induced by edge $f \notin B$ (used when a new edge is inserted or when the weight of nontree edge is decreased), finding the minimum weight edge in the cut obtained after removing $e \in B$ (used when a tree edge is deleted or has its weight increased), and swapping edges e and f (used to update the solution after a change).

THEOREM 4.3.1. *Let G be a graph with n vertices and m edges of d colors. A solution to the color-constrained minimum spanning tree problem can be maintained in* $O(d^2n^{1/2} + d^{11/3}(d!)^2n^{1/3} \log n)$ *time per update, and* $O(dm + d^3n)$ *space.*

PROOF. Frederickson and Srinivas [1989] showed that, if $U(m, n)$ denotes the time needed to perform *maxcirc, mincocirc*, and *swap* on spanning trees in uncolored graphs, and $U(m, n) = O(m^{1/2})$, then a solution to the color-constrained minimum spanning tree problem can be maintained in $O(d^2U(m, n) + d^{11/3}(d!)^2n^{1/3} \log n)$ time per update, and $O(dm + d^3n)$ space.

Our data structure for updating a minimum spanning tree lets us solve *maxcirc*, *mincocirc*, and *swap* in time $U(m, n) = O(n^{1/2}) = O(m^{1/2})$. Thus, the bounds in Frederickson and Srinivas' algorithm become the ones claimed in the theorem. □

5. *Edge Connectivity*

In this section, we consider the problem of maintaining information about the k-edge connectivity of an undirected graph during edge insertions and deletions. Typical query operations we would like to perform include checking if the graph is k-edge-connected, or checking whether any two given vertices are k-edge-connected. The best previous data structure for 2-edge connectivity is due to Frederickson [1997], who improved a previous $O(m^{2/3})$ time bound per update [Galil and Italiano 1992] to $O(m^{1/2})$. Galil and Italiano [1993] and La Poutré [1991] gave algorithms for maintaining 3-edge connectivity with insertions only, in amortized time $O(\alpha(q, n))$ per insertion. Galil and Italiano [1991a] discovered a fully dynamic algorithm for 3-edge connectivity, with an $O(m^{2/3})$ time bound. No nontrivial dynamic algorithm was known for any higher order of connectivity.

If a graph is k-edge-connected, we could use a minimal k-edge-connected subgraph as a sparse certificate. However, it has only been recently that algorithms were developed for finding such subgraphs in linear time, even for 2-edge connectivity [Han et al. 1995]. Further, no sublinear-time algorithms are known for maintaining such graphs, let alone with any sort of stability properties.

Instead we use the following technique, which produces certificates guaranteed to be both sparse and stable. Given a graph G, let $T_1 = U_1$ be a spanning forest of G. Let T_2 be a spanning forest of $G - U_1$, and let $U_2 = U_1 \cup T_2$. In general, let T_i be a spanning forest of $G - U_{i-1}$, and let U_i be $U_{i-1} \cup T_i$. Both Thurimella and Nagamochi and Ibaraki noted the following connection between these graphs and edge connectivity:

LEMMA 5.1 ([THURIMELLA 1989; NAGAMOCHI AND IBARAKI 1992]). *U_k is a certificate of k-edge connectivity for G.*

More generally, U_k and G have the same set of $(k - 1)$-cuts, and the same set of k-edge-connected components. We need the following stronger result:

LEMMA 5.2. *U_k is a strong certificate of k-edge connectivity for G.*

PROOF. Consider any k-edge-connected graph $G \cup H$, and build a certificate in the following way. Start with T_1, and augment it with edges of H to be a spanning forest for $G \cup H$. Then start with T_2, and augment it with some of the remaining edges of H to be a spanning forest for $G \cup H$. In this way, we obtain a collection of forests, which by Lemma 5.3 is a certificate for $G \cup H$, and which is a subgraph of $U_k \cup H$. Therefore, if $G \cup H$ is k-edge-connected, so is $U_k \cup H$. On the other hand, if $G \cup H$ is not k-edge-connected, then neither can its subgraph $U_k \cup H$. So U_k is a strong certificate for G. □

For any fixed k, we can maintain a graph U_k in time $O(km^{1/2})$ per update, using k copies of Frederickson's minimum spanning forest algorithm. The resulting certificate is stable, since each update involves at most a single edge flip in each of k minimum spanning trees.

We are now ready to state our results for edge connectivity.

THEOREM 5.3. *The 2-edge-connected components of an undirected graph can be maintained in time $O(n^{1/2})$ per update. Queries asking whether two vertices are in the same component can be answered in $O(\log n)$ time.*

PROOF. We use Frederickson's minimum spanning tree algorithm [Frederickson 1985] to maintain the stable strong certificate U_2, giving $f(n, m) = O(m^{1/2})$. Each update to the graph translates to $O(1)$ changes to U_2. The 2-edge-connected components of U_2 can be maintained in time $g(n, m) = O(m^{1/2})$, with queries whether two vertices are in the same component answered in time $q(n, m) = O(\log n)$ [Frederickson 1997]. Applying Theorem 3.3.2 gives our result. □

THEOREM 5.4. *The 3-edge-connected components of an undirected graph can be maintained in time $O(n^{2/3})$ per update and $O(n^{2/3})$ per query.*

PROOF. We use the same technique, maintaining the 3-edge-connected components in U_3 with the $O(m^{2/3})$-time algorithm of Galil and Italiano [1991a]. $f(n, m) = O(m^{1/2})$ as before, and $g(n, m) = q(n, m) = O(m^{2/3})$, so the total time is $O(n^{1/2} + n^{2/3}) = O(n^{2/3})$. □

THEOREM 5.5. *The 4-edge-connected components of an undirected graph can be maintained in time $O(n\alpha(n))$ per update.*

PROOF. We maintain U_4, and after each update recompute whether the resulting graph is 4-edge-connected. We transform the problem into one of testing 4-vertex connectivity by replacing every vertex by a 2-hub wheel [Galil and

Italiano 1991c], and then use the $O(m + n\alpha(n))$-time 4-vertex connectivity algorithm of Kanevsky et al. [1991]. □

THEOREM 5.6. *For any fixed constant k, information on whether an undirected graph is k-edge-connected can be maintained in time $O(n \log n)$ per update.*

PROOF. We maintain the stable strong certificate U_k using our dynamic minimum spanning tree techniques, and check if U_k is k-edge-connected after each update using Gabow's $O(m + n \log n)$-time algorithm [Gabow 1991a]. □

6. *Vertex Connectivity*

Vertex connectivity seems to be harder in general than edge connectivity. The best connectivity testing algorithms are significantly faster for edge connectivity than for vertex connectivity. This relative hardness was made explicit by Galil and Italiano [1991c], who showed how computations of edge connectivity could be reduced to computations of vertex connectivity; no reduction in the reverse direction is known.

Nevertheless, our sparsification technique is still useful here, and provides the first nontrivial fully dynamic algorithms for vertex connectivity. As usual, we must first define our certificates; they will turn out to be a restricted form of the certificates U_k used for edge connectivity.

Given a graph G, let $B_1 = C_1$ be a breadth-first spanning forest of G; that is, we start from any vertex and perform a breadth-first search until all edges are exhausted; then we continue by restarting at any unvisited node until all connected components of the graph are covered. Such a search is well known to take linear total time. Similarly, let B_2 be a breadth first forest of $G - C_1$, and let $C_2 = C_1 \cup B_2$. In general, let B_i be a breadth first forest of $G - C_{i-1}$, and let $C_i = C_{i-1} \cup B_i$. Cheriyan and Thurimella [1991] prove that C_k is a certificate for the k-vertex-connectedness of G. Nagamochi and Ibaraki [1992] showed that a sparse certificate for k-vertex connectivity can be found in time $O(m + n)$. As before, we need a strengthening of these results.

LEMMA 6.1. *C_k is a strong certificate of k-vertex connectivity for G.*

PROOF. Consider a $(k - 1)$-vertex cut S in any graph $C_k \cup H$, partitioning the remaining vertices into two components A and B. Obviously, H can contain no edges from A to B. Let x be the vertex of S first visited by the breadth first search defining forest B_1. Suppose x was visited from a vertex in A. Then, all the edges from x to B in G will be visited before any edge from another vertex of S to B. So all such edges will be in B_1 instead of $C_k - B_1$, and $S - x$ forms a $(k - 2)$-vertex cut in $C_k - B_1$. Continuing in the same way, we see that each breadth-first search eliminates a vertex, and so B_k must be disconnected. But this can only happen if S is also a cutset in G, and hence in $G \cup H$.

Thus, we see that any cut in $C_k \cup H$ is also a cut in $G \cup H$. The converse follows immediately from the fact that C_k is a subgraph of G. Hence, C_k is a strong certificate. □

THEOREM 6.2. *The 2- and 3-vertex connected components of an undirected graph can be maintained in time $O(n)$ per update. The 4-vertex connectivity of an undirected graph can be maintained in time $O(n\alpha(n))$ per update.*

PROOF. Each certificate C_k can be found in time $f(n, m) = O(km)$ by repeated application of breadth-first search. At the root of the sparsification tree, k-vertex connectivity can be tested, and k-vertex-connected components identified, in linear time for $k = 2$ and $k = 3$ [Hopcroft and Tarjan 1973; Tarjan 1972]. 4-vertex connectivity can be tested in time $g(n, m) = O(m + n\alpha(n))$ [Kanevsky et al. 1991]. Applying the basic sparsification technique of Theorem 3.1.1 proves the results. $\square$

7. *Bipartiteness*

In this section, we show how to maintain efficiently information on whether a graph is bipartite, during insertions and deletions of edges. We first give a simple algorithm based on Theorem 3.1.1, which supports each update in $O(n)$ time.

THEOREM 7.1. *We can perform edge insertions or deletions in a graph, and compute whether the graph is bipartite after each update, in time $O(n)$ per update.*

PROOF. Recall that a graph G is bipartite if and only if G does not contain an odd cycle. Therefore, a spanning forest of G together with one more edge inducing an odd cycle, or just a spanning forest if no such edge exists, is a sparse strong certificate of bipartiteness. Since this certificate can be found in linear time, Theorem 3.1.1 proves the result. $\square$

Next, we show how to improve this bound to $O(n^{1/2})$, by using more sophisticated data structures combined with the results of Theorem 3.3.2. Note that in order to apply Theorem 3.3.2, we need a stable sparse certificate and a fully dynamic algorithm to be sped up. We use the same certificate as before: namely, we use a spanning forest of G plus one additional edge inducing an odd cycle (if there is such an edge).

LEMMA 7.2. *The certificates described above form the bases of a matroid.*

PROOF. We described in Section 4.3 one of a number of standard definitions of a matroid, in terms of certain axioms on a family of *independent sets*; here we need only verify those axioms for these certificates. What we want to prove is that the subgraphs of G formed either as forests or as forests together with a single edge spanning an odd cycle form the independent sets of a matroid. Clearly, a subgraph of such a subgraph is itself of the same form, so the key property we need to show is the other axiom defining a matroid: if we have two such subgraphs C_1 and C_2, with $|C_1| > |C_2|$, then there is some edge $e \in C_1$ such that $C_2 \cup \{e\}$ is also of the above type.

If C_1 spans a larger portion of the graph than C_2, we can add an edge from C_2 to C_1 and span a larger graph than C_2 alone (this is simply the matroid basis exchange rule applied to the standard spanning forest matroid). Otherwise, C_1 and C_2 span the same portion of G, but C_1 contains an odd cycle whereas C_2 is a forest. Two-color the connected component of C_2 containing the odd cycle in C_1. Since the cycle is odd, some edge of it must have both endpoints the same color. But then that edge spans an odd cycle in C_2, and can be added to C_2 to form a larger independent set. $\square$

COROLLARY 7.3. *If we arbitrarily supply distinct edge weights to a graph, then the subgraph formed by using the minimum spanning forest of a graph together with* *the minimum weight additional edge forming an odd cycle, is a stable certificate for bipartiteness.*

PROOF. The red and blue rules described for minimum spanning trees work for any matroid—the generalized version of the red rule removes the minimum element in any circuit of the matroid, and the generalized blue rule adds the minimum element in each cocircuit of the matroid. The correctness of the blue rule can easily be seen from the more common greedy algorithm for finding the minimum weight base: If we add elements one by one in sorted order, the minimum element of the cocircuit will be examined first, at which point it will certainly be independent of the previously added elements, and so must be part of the minimum weight base. The red rule is just the matroid dual of the blue rule.

The certificate used here is the minimum weight base in the matroid of Lemma 7.2: By the blue rule, we can add the edges of the minimum spanning forest first, at which point the only edge to add is the minimum weight edge inducing an odd cycle. Lemmas 4.1 and 4.2 were proved using only the correctness of the red and blue rules, so their analogues hold and show that the minimum base for any matroid is stable. $\square$

A form of stability would be preserved if we used any additional edge inducing an odd cycle, not necessarily that of minimum weight, since an $O(1)$ bound on the amount of change per update would follow from the minimum spanning forest's stability. However, there would not then be a unique certificate for each graph, so the mapping required in the definition of stability would not exist. Stable sparsification could still be made to work in this case, but using the minimum weight certificate saves us from having to complicate the definition of stability. The use of matroids in Corollary 7.3 also shows that as in the case of minimum spanning forests stability can be used directly in a simpler way than our general stable sparsification technique.

We next describe a fully dynamic algorithm that maintains the certificates described above in $O(m^{1/2})$ time per update. This algorithm is an application of the topology tree of Frederickson [1997], and can be viewed as a version of Frederickson's *ambivalent data structures*. We refer the reader to Frederickson [1997] for the details of the method. For our purposes, it is enough to recall from Frederickson [1997] some definitions on restricted multi-level partitions, topology trees, and 2-dimensional topology trees.

We first perform a standard transformation to convert the graph G into a graph with maximum vertex degree 3 [Harary 1969]: Suppose $v \in V$ has degree $d(v) > 3$, and is adjacent to vertices $u_1, u_2, \ldots, u_d$. In the transformed graph, v is replaced by a chain of $d - 1$ *dashed* edges: namely, we substitute v by d vertices $v_1, v_2, \ldots, v_d$. For each edge (v, u) of the original graph, in position i among the list of edges adjacent to v and position j among the edges adjacent to u, we create an *actual* edge (v_i, u_j). We also create *dashed* edges (v_i, v_{i+1}) for $1 \leq i \leq d - 1$. As a result of this transformation, the graph keeps its actual edges, and has an additional $O(m)$ dashed edges.

Throughout the sequence of updates, a spanning tree T of G containing all the dashed edges is maintained. A *vertex cluster* with respect to T is a set of vertices that induces a connected subgraph on T. An edge is *incident* to a cluster if exactly one of its endpoints is inside the cluster. Two clusters are *adjacent* if there is a tree edge that is incident to both. A *boundary vertex* of a cluster is a vertex that is adjacent in T to some vertex not in the cluster. The *external degree* of a cluster is the number of tree edges incident to it.

A *restricted partition of order* z of G is a partition of its vertex set V into $O(m/z)$ vertex clusters such that:

(1) Each set in the partition yields a vertex cluster of external degree at most 3.
(2) Each cluster of external degree 3 is of cardinality 1.
(3) Each cluster of external degree less than 3 is of cardinality less than or equal to z.
(4) No two adjacent clusters can be combined and still satisfy the above.

Any cluster in such a partition with more than one vertex will have maximum external degree 2, and therefore all clusters will have at most two boundary vertices.

A *restricted multi-level partition* consists of a collection of restricted partitions of V satisfying the following:

(1) The clusters at level 0 (known as *basic clusters*) form a restricted partition of order z.
(2) The clusters at level $l \geq 1$ form a restricted partition of order 2 with respect to the tree obtained after shrinking all the clusters at level $l - 1$.
(3) There is exactly one vertex cluster at the topmost level.

Frederickson showed that the number of levels in a restricted multi-level partition is $\Theta(\log n)$ [Frederickson 1997].

The *topology tree* is a hierarchical representation of G based on T. Each level of the topology tree partitions the vertices of G into connected subsets called *clusters*. More precisely, given a restricted multi-level partition for T, a *topology tree* for T is a tree satisfying the following:

(1) A topology tree node at level l represents a vertex cluster at level l in the restricted multi-level partition.
(2) A node at level $l \geq 1$ has at most two children, representing the vertex clusters at level $l - 1$ whose union gives the vertex cluster the node represents.

As shown in Frederickson [1997], the update of a topology tree because of an edge swap in T consists of two subtasks. First, a constant number of basic clusters (corresponding to leaves in the topology tree) have to be examined and possibly updated. Since each basic cluster has size $O(z)$, this can be supported in $O(z)$ time. Second, the changes in these basic clusters percolate up in the topology tree, possibly causing vertex clusters in the multilevel partition to be regrouped in different ways. This is handled by rebuilding portions of the topology tree in a bottom-up fashion, and involves a constant amount of work to be done on at most $O(\log n)$ topology tree nodes. Consequently, the update of a topology tree because of an edge swap can be supported in time $O(z + \log n)$ [Frederickson 1997].

A *2-dimensional topology tree* for a topology tree is defined as follows: For every pair of nodes V_α and V_β at the same level in the topology tree there is a node labeled $V_\alpha \times V_\beta$ in the 2-dimensional topology tree. Let E_T be the tree edges of G (i.e., the edges in the spanning tree T): node $V_\alpha \times V_\beta$ represents all the non-tree edges of G (i.e., the edges of $E - E_T$) having one endpoint in V_α and the other in V_β. The root of the 2-dimensional topology tree is labeled $V \times V$ and represents all the non-tree edges of G. If a node is labeled $V_\alpha \times V_\beta$, and V_α has children $V_{\alpha i}$, $1 \le i \le p$, and V_β has children V_{β_j}, $1 \le j \le q$, in the topology tree, then $V_\alpha \times V_\beta$ has children $V_{\alpha_i} \times V_{\beta_j}$, $1 \le i \le p$, $1 \le j \le q$, in the 2-dimensional topology tree.

The update of a 2-dimensional topology tree during a swap in its corresponding topology tree can be performed in $O(m/z)$ time [Frederickson 1997]. The crucial point of this analysis is that only $O(m/z)$ nodes in the 2-dimensional topology tree need to be looked at and eventually updated during a swap.

The last concept needed from Frederickson [1997] is the concept of *ambivalent data structure*. This is based upon the idea of maintaining a small number of possible alternatives in a data structure, even though only one of them can be valid at any time. The ambivalence in our application comes from not knowing whether a given edge induces an even or odd cycle in the MST; we solve this difficulty by storing two edges at each node of the two-dimensional topology tree, one inducing a cycle of each parity.

We now describe the details of our application of Frederickson's clustering method to bipartiteness. Throughout the sequence of operations, we maintain a spanning tree T containing all of the dashed edges, and we make use of a 2-dimensional topology tree to keep track of the parity of the cycles induced by non-tree edges of G in this spanning forest. Note that non-tree edges are actual edges, that is, edges of the original graph (before applying the transformation to have vertex degrees no larger than three). Given a non-tree edge $e \in E - E_T$, we denote by λ_e the cycle induced in the spanning tree T by e.

Our data structure will be a topology tree for T, and the corresponding 2-dimensional topology tree, both augmented with the following additional information. For each node $V_j \times V_r$ in the 2-dimensional topology tree, we maintain at most two non-tree edges of G between V_j and V_r, one for each parity of their induced cycle in T.

The other information we maintain is the following: For each selected edge, we maintain in node $V_j \times V_r$ of the 2-dimensional topology tree the distances between its endpoint in a cluster (either V_j or V_r) and all boundary vertices in the same cluster. By distance between two vertices u and v, we mean the number of *actual* edges (i.e., not dashed) in the spanning tree path between u and v. Finally, for each cluster V_j we maintain in the corresponding node of the topology tree the distances between all pairs of boundary vertices of V_j. Given two vertices v_i and v_j, we denote their distance by $d(v_i, v_j)$. Since each cluster contains at most three boundary vertices, there will be at most $O(1)$ distances to be stored at each node of the topology tree and of the 2-dimensional topology tree. As seen before, in order to decide which non-tree edge is to be maintained between V_j and V_r, we must be able to check whether two cycles λ_{f_1} and λ_{f_2} have the same parity. The following lemma shows that in order to accomplish this task it is enough to know the distances between the endpoints of f_1 and f_2, and the boundary vertices in V_j and V_r.

LEMMA 7.4. *Let V_j and V_r be any two clusters at the same level of the topology tree, and let f_1 and f_2 be any two non-tree edges between V_j and V_r. Let w_j be a* *boundary vertex of V_j, and let w_r be a boundary vertex of V_r. Let j_1 and j_2 be respectively the endpoints of f_1 and f_2 in V_j and let r_1 and r_2 be respectively the endpoints of f_1 and f_2 in V_r. The two cycles λ_{f_1} and λ_{f_2} have the same parity if and only if the quantity $d(j_1, w_j) + d(j_2, w_j) + d(r_1, w_r) - d(r_2, w_r)$ is even.*

PROOF. Let π_j be the tree path (which is entirely contained in cluster V_j) between j_1 and j_2, and let π_r be the tree path (which is entirely contained in V_r) between r_1 and r_2. Note that π_j, f_1, π_r and f_2 form a cycle $C(f_1, f_2)$ in G. We prove the lemma by proving the following two claims:

(1) λ_{f_1} and λ_{f_2} have the same parity if and only if $C(f_1, f_2)$ has an even number of edges.

(2) $C(f_1, f_2)$ has an even number of edges if and only if $d(j_1, w_j) + d(j_2, w_j) + d(r_1, w_r) + d(r_2, w_r)$ is even.

We first prove claim (1). Consider the subgraph formed by the symmetric difference of λ_{f_1} and λ_{f_2}. It is easy to see that it consists of π_j, π_r, f_1, and f_2 and therefore (1) follows.

We now turn to claim (2). The length of $C(f_1, f_2)$ is $(|\pi_j| + |\pi_r| + 2)$. The tree path between j_1 and w_j unioned with the path between j_2 and w_j gives a path (not necessarily simple) between j_1 and j_2. Consequently, it must contain π_j as a subpath and $d(j_1, w_j) + d(j_2, w_j)$ must be equal to $(|\pi_j| + 2q_j)$ for some $q_j \geq 0$ ($q_j = 0$ if and only if w_j is in π_j). Similarly, $d(r_1, v_r) + d(r_2, v_r) = (|p_r| + 2q_r)$ for some $q_r \geq 0$ ($q_r = 0$ if and only if w_r is in π_r). So, $d(j_1, w_j) + d(j_2, w_j) + d(r_1, w_r) + d(r_2, w_r) = (|p_j| + |p_r| + 2q_j + 2q_r)$, which has the same parity as $(|p_j| + |p_r| + 2) = |C(f_1, f_2)|$. □

Since the cycles λ_{f_1} and λ_{f_2} depend on how clusters V_j and V_r are connected in the spanning tree T, we cannot determine the parity of either λ_{f_1} or λ_{f_2} by looking only at V_j and V_r. However, Lemma 7.4 tells us that knowing the distances between the endpoints of f_1 and f_2 and the boundary vertices of V_j and V_r is enough to decide whether these two cycles have the same parity or not. This is particularly important, since (contrary to the parity of a cycle) the information about these distances is local to a cluster and needs no update as long as the cluster does not change.

We now show how to maintain the extra information stored in the nodes of the augmented topology tree and of the augmented 2-dimensional topology tree whenever these trees undergo an update such as an edge swap.

The extra information stored in the augmented topology tree can be updated as follows. If the swap causes a basic cluster to be split or merged, we recompute the new cluster V_j, together with all the edges incident to it, and their distance from the boundary vertices of V_j. While percolating this update up in the topology tree, we follow the algorithm of Frederickson [1997]. The only difference is in how we update the information about the distance of the boundary vertices. Assume we are handling an affected cluster V_j, that consists of two children clusters $V_{j'}$ and $V_{j''}$ joined by edge (w', w''). Let w' be a boundary vertex of $V_{j'}$, and w'' be a boundary vertex of $V_{j''}$. The distance $d(v', v'')$ in V_j between a boundary vertex v' previously in $V_{j'}$ and a boundary vertex v'' previously in $V_{j''}$ can be computed in $O(1)$ time by considering $d(v', w')$ (available in $V_{j'}$), $d(w'', v'')$ (available in $V_{j''}$) and the edge (w', w''). This adds only extra $O(1)$ time per node considered, and consequently keeps the bound required to update the augmented topology tree $O(z + \log n)$.

We now turn to the update of the augmented 2-dimensional topology tree.

LEMMA 7.5. *The augmented topology tree and 2-dimensional topology tree can be initialized in $O(m)$ time and space, and updated in $O(m^{1/2})$ time.*

PROOF. Recall that as shown in Frederickson [1985; 1997] updating a 2-dimensional topology tree can be done by performing a constant amount of work on $O(m/z)$ nodes. To generate the new augmented 2-dimensional topology tree, we follow the same ideas as in Frederickson [1985; 1997]. We only sketch here the differences with that algorithm. For each basic cluster V_j that has changed, do the following. For each other basic cluster V_r and for each set of nontree edges between V_j and V_r, find the edges (at most two) that will be stored in $V_j \times V_r$. By Lemma 7.4, this can be done by using the distances computed for clusters V_j and V_r and stored in the topology tree. We now describe how to update the selected edges for an internal node $V_j \times V_r$ of the 2-dimensional topology tree. Denote by $V_{j'}$ and $V_{j''}$ the subclusters children of V_j in the topology tree, and by $V_{r'}$ and $V_{r''}$ the subclusters children of V_r. (This is the more general case, since either V_j or V_r can consist of only one subcluster). We compute the selected edges of $V_j \times V_r$ starting from the selected edges of the four nodes $V_{j'} \times V_{r'}$, $V_{j'} \times V_{r''}$, $V_{j''} \times V_{r'}$, and $V_{j''} \times V_{r''}$, as follows.

The minimum weight edge between V_j and V_r of a given parity must come from some combination of $V_{j'} \times V_{r'}$, $V_{j'} \times V_{r''}$, $V_{j''} \times V_{r'}$, and $V_{j''} \times V_{r''}$, and will be the minimum weight edge of its parity in that combination. We compare the parities of the (at most eight) such edges, using Lemma 7.4, to determine the selected edges for $V_j \times V_r$. Again, this shows how the augmented 2-dimensional topology tree can be updated in $O(m/z)$ time. □

THEOREM 7.6. *We can perform edge insertions or deletions in a graph, and compute whether the graph is bipartite after each update, in time $O(n^{1/2})$ per update.*

PROOF. After performing any update on the augmented 2-dimensional topology tree, the minimum weight edge inducing an odd cycle in the minimum spanning forest must be one of the two selected edges for the node $V \times V$ at the root of the 2-dimensional topology tree. We can test one or both such edges by computing the length of the induced cycle, using Sleator and Tarjan's dynamic tree data structure. If no such edge is selected, or one edge is selected but yields an even cycle, then the graph is bipartite and no odd cycle exists. Thus, the augmented topology tree and 2-dimensional topology tree yields an $O(m^{1/2})$ update algorithm for maintaining our certificate for bipartiteness, as well as $O(m^{1/2})$ fully dynamic algorithm for maintaining information about the bipartiteness of a graph. Applying Theorem 3.3.2 gives the result. □

Note that we can also answer queries, specifying the colors of two vertices of the graph and asking whether there exists a two-coloring of the graph with those colors, in the same time bounds, simply by examining the parity of the length of the path between the two vertices in the minimum spanning forest.

8. *Faster Insertions*

We saw earlier that the property of bipartiteness can be maintained in $O(n)$ time per update. Using Theorem 3.4.1, we can improve this result by combining sparsification with a partially dynamic algorithm for the same problem:

LEMMA 8.1. *In linear time, we can construct a data structure for a bipartite graph such that each additional edge can be added, and the bipartiteness of the resulting graph tested, in amortized time $O(\alpha(n))$.*

PROOF. We modify the well-known union–find data structure. We represent each vertex by a set element, and each connected component of the graph by a set. Sets in the union–find data structure are represented by trees; we augment the data structure so that each tree edge stores a bit of information, denoting the parity of the paths between the two elements it connects. When a new graph edge is added between points in different sets, we connect the roots of the two sets with an edge of the appropriate parity. When a new edge connects points in the same set, we must test whether it forms an odd cycle, by compressing the paths from those points to the root of the set, and examining the parity of the resulting edges. If an odd cycle is found, all further insertions can be ignored. All operations can be performed in the same time as the basic union–find algorithm, which is well known to have an $O(\alpha(n))$ amortized time bound. □

THEOREM 8.2. *We can construct a data structure for testing bipartiteness for which each insertion can be performed in time $O(\alpha(n))$ and each deletion can be performed in time $O(n \log(m/n))$.*

PROOF. We combine the data structure above with Theorem 3.4.1. □

We next apply this technique to the spanning tree and connectivity problems we solved using the other versions of our sparsification technique.

THEOREM 8.3. *The minimum spanning forest of a graph can be maintained in time $O(\log n)$ per insertion and $O(n \log(m/n))$ per deletion.*

PROOF. We apply Theorem 3.4.1 using a partially dynamic algorithm based on the dynamic tree data structure of Sleator and Tarjan [1983]. □

THEOREM 8.4. *The connected, 2- and 3-edge-connected, and 2- and 3-vertex-connected components of a graph can be maintained in amortized time $O(\alpha(q, n))$ per insertion or query and $O(n \log(m/n))$ per deletion, where q is the total number of queries made.*

PROOF. For connected components, we use the well known union–find algorithm as our partially dynamic data structure. For 2-connected components of either type, we use the partially dynamic algorithms of Westbrook and Tarjan [1992]. For 3-edge-connected components, we use the algorithm of La Poutré [1991]. For 3-vertex-connected components we use an algorithm of La Poutré, van Leeuwen, and Overmars (personal communication cited in Di Battista and Tamassia [1990]). □

THEOREM 8.5. *The 4-vertex connectivity of a graph can be maintained in time $O(\log n)$ per insertion and $O(n \log n)$ per deletion.*

PROOF. We use a data structure of Kanevsky et al. [1991], for which a sequence of l insertions takes time $O(n \log n + l)$. Thus, $p(n, m) = O(1)$, $f(n, m) = O(m + n \log n)$, and the result follows from Theorem 3.4.1. □

9. *Conclusions and Recent Work*

We have presented a new technique, called sparsification, that proves to be very useful in the design of fully dynamic graph algorithms. As an application, we have presented new improved algorithms for dynamic minimum spanning forests, and for several types of dynamic edge and vertex connectivity. Sparsification can also be applied to static problems such as computing the k smallest spanning trees of a graph. The technique is quite general and uses previous algorithms as subroutines. Subsequent research has expanded our results and confirmed the importance of the sparsification technique.

While sparsification has many applications in algorithms for arbitrary graphs, it seemed unlikely that it could be used to speed up algorithms for families of graphs that are already sparse, such as planar graphs. However, in recent work [Eppstein et al. 1996; 1998], we applied sparsification to dynamic planar graph problems, by expanding the notion of strong certificate introduced in this paper to graphs in which a subset of the vertices are denoted as *interesting*; these certificates may reduce the size of the graph by removing uninteresting vertices. Using this notion, we defined a type of sparsification based on *separators*, small sets of vertices the removal of which splits the graph into roughly equal size components. Repeatedly finding separators in these components gives a *separator tree*, which we also use as our sparsification tree; the interesting vertices in each certificate will be those vertices used in separators at higher levels of the tree.

Fernández-Baca et al. [1996] have applied the sparsification technique described here to an alternate type of dynamic graph problem: one in which edge weights do not change by discrete update events, but rather in which the edge weights are linear functions of a time parameter. They showed how to use these techniques to quickly construct the sequence of minimum spanning trees produced by such an input, or to find values of the time parameter satisfying certain criteria.

Henzinger and La Poutré [1995] used sparsification to solve fully dynamic 2-vertex-connectivity in time $O(\sqrt{n \log n} \log(m/n))$, improving the bounds given here. Henzinger and King have given randomized algorithms for some of the problems described in this paper, including connectivity, 2-edge connectivity, 2-vertex connectivity, and bipartiteness improving their expected time bounds to polylogarithmic [Henzinger and King 1995a; 1995b]. Very recently, Henzinger and King [1997] presented an $O(m^{1/3} \log n)$ algorithm for the dynamic minimum spanning tree problem: combined with sparsification, this improves the $O(n^{1/2})$ bound given in this paper.

Sparsification is very simple to implement and likely to be used in practice, as shown in some recent work. Experimental comparison of some of the dynamic connectivity algorithms has been performed by Alberts et al. [1997]. The method compared the basic sparsification algorithm (having a worst-case update bound of $O(n \log(m/n))$) with a randomized method based on that of Henzinger and King (having an expected amortized bound of $O(\log^3 n)$). The sparsification method worked well for small update sequences, but the other method was faster on longer sequences. It remains to be seen how well stable sparsification would perform in similar experiments. Alberts et al. [1997] also showed that in the average case for sufficiently random inputs, a simple sparsification tree based on edge subdivision performs as well as the vertex-subdivision method we described in Theorem 3.1.1. Very recently, Amato et al. [1997] performed an extensive experimental study of dynamic algorithms for minimum spanning tree problems, and showed that stable sparsification has small constants and yields fast algorithms even on small graphs.

10. *Open Problems*

Beyond the question of general improvement in any of our bounds, some specific open questions remain to be solved.

First, to what extent can we take advantage of possible imbalances in insertion and deletions? Theorem 3.4.1 provides algorithms with extremely fast insertion times, and relatively slow deletion times. Theorem 3.3.2 on the other hand provides algorithms in which the insertion and deletion times are approximately equal. Can we combine these results to trade off the two time bounds against each other, for situations in which insertions should be faster than deletions by less than a linear factor? Recent work of Henzinger and King [1995b; 1995a; 1997] seems to shed some light on this direction.

Second, for minimum spanning forests, bipartiteness, connectivity, and 2- and 3-edge connectivity, we can achieve sublinear update times. As noted above, this was also achieved by Henzinger and La Poutré [1995] for 2-vertex connectivity. Can these results be extended to higher orders of connectivity?

Third, can we allow other update operations besides edge insertion and deletion? Vertices can be inserted, and isolated vertices deleted, in the same times as edge insertion and deletion, if we maintain the size of each group of edges to be some number N near n that changes less frequently than n itself. Is there some way of allowing rapid deletion of vertices that may still be connected to many edges?

Fourth, lower bounds on the problems we solve deserve further study. Berman et al. [1990] claim such results for dynamic minimum spanning trees and various connectivity problems, but their bounds only hold if unreasonably small limits are placed on preprocessing time. Recently, Fredman and Henzinger [1997] proved an $\Omega(\log n/\log \log n)$ lower bound per operation for fully dynamic k-edge connectivity in the cell-probe model of computation. Can the gap between upper and lower bounds for these problems be tightened? Furthermore, can we prove nontrivial lower bounds for the other problems? We can prove that sparsification does not work for certain problems; for instance if $\mathcal{P}$ is the property of having a perfect matching, the only strong certificate for any graph is the graph itself. Further work on the inability of sparsification to solve certain problems was recently presented by Khanna et al. [1996].

ACKNOWLEDGMENTS. We are grateful to Yefim Dinitz and Tom Spencer for useful comments, and to Greg Frederickson for suggesting the application of sparsification to color-constrained minimum spanning trees.

REFERENCES

ALBERTS, D., CATTANEO, G., AND ITALIANO, G. F. 1997. An empirical study of dynamic graph algorithms. *ACM J. Exper. Algorithmics*, to appear.

AMATO, G., CATTANEO, G., AND ITALIANO, G. F. 1997. Experimental analysis of dynamic minimum spanning tree algorithms. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, pp. 314–323.

BERMAN, A. M., PAULL, M. C., AND RYDER, B. G. 1990. Proving relative lower bounds for incremental algorithms. *Acta Inf. 27*, 665–683.

CHERIYAN, J., KAO, M. Y., AND THURIMELLA, R. 1993. Scan-first search and sparse certificates–An improved parallel algorithm for k-vertex connectivity. *SIAM J. Comput. 22*, 157–174.

CHERIYAN, J., AND THURIMELLA, R. 1991. Algorithms for parallel k-vertex connectivity and sparse certificates. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing* (New Orleans, La., May 6–8). ACM, New York, pp. 391–401.

DI BATTISTA, G., AND TAMASSIA, R. 1990. On-line graph algorithms with SPQR-trees. In *Proceedings of the 17th International Collquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 443. Springer-Verlag, Berlin, Germany, pp. 598–611.

DIETZ, P., AND SLEATOR, D. 1987. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing* (New York, N.Y., May 25–27). ACM, New York, pp. 365–372.

DIXON, B., RAUCH, M., AND TARJAN, R. E. 1992. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 1184–1192.

EPPSTEIN, D. 1992. Finding the k smallest spanning trees. *BIT 32*, 237–248.

EPPSTEIN, D. 1994a. Offline algorithms for dynamic minimum spanning tree problems. *J. Algor. 17*, 237–250.

EPPSTEIN, D. 1994b. Tree-weighted neighbors and geometric k smallest spanning trees. *Int. J. Comput. Geom. Appl. 4*, 229–238.

EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND SPENCER, T. H. 1996. Separator based sparsification I. Planarity testing and minimum spanning trees. *J. Comput. Syst. Sci, 52*, 1 (Feb.), 3–27.

EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND SPENCER, T. H. 1998. Separator based sparsification II. Planarity testing and minimum spanning trees. *SIAM J. Comput.*, to appear.

EPPSTEIN, D., ITALIANO, G. F., TAMASSIA, R., TARJAN, R. E., WESTBROOK, J., AND YUNG, M. 1992. Maintenance of a minimum spanning forest in a dynamic plane growth. *J. Algor. 13*, 33–54.

FEDER, T., AND MIHAIL, M. 1992. Balanced matroids. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing* (Victoria, B.C., Canada, May 4–6). ACM, New York, pp. 26–38.

FERNANDEZ-BACA, D., SLUTZKI, G., AND EPPSTEIN, D. 1996. Using sparsification for parametric minimum spanning tree problems. *Nordic J. Comput. 3*, 4 (Winter), 352–366 (Special issue for the 5th SWAT).

FREDERICKSON, G. N. 1985. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput. 14*, 781–798.

FREDERICKSON, G. N. 1997. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput. 26*, 2 (Apr.), 484–538.

FREDERICKSON, G. N., AND SRINIVAS, M. A. 1989. Algorithms and data structures for an expanded family of matroid intersection problems. *SIAM J. Comput. 18*, 112–138.

FREDMAN, M. L., AND HENZIGER, M. R. 1997. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, to appear.

GABOW, H. N. 1991a. Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 812–821.

GABOW, H. N. 1991b. A matroid approach to finding edge connectivity and packing arborescenes. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing* (New Orleans, La., May 6–8). ACM, New York, pp. 112–122.

GABOW, H. N., AND STALLMAN, M. 1985. Efficient algorithms for graphic matroid intersection and parity. In *Proceedings of the 12th International Colloquium of Automata, Languages, and Programming*. Lecture Notes in Computer Science, vol. 194. Springer-Verlag, Berlin, Germany, pp. 339–350.

GALIL, Z., AND ITALIANO, G. F. 1991a. Fully dynamic algorithms for 3-edge-connectivity. Unpublished manuscript.

GALIL, Z., AND ITALIANO, G. F. 1991b. Maintaining biconnected components of dynamic planar graphs. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science*, vol. 510. Springer-Verlag, Berlin, Germany, pp. 339–350.

GALIL, Z., AND ITALIANO, G. F. 1991c. Reducing edge connectivity to vertex connectivity. *Sigact News 22*, 57–61.

GALIL, Z., AND ITALIANO, G. F. 1992. Fully dynamic algorithms for 2-edge-connectivity. *SIAM J. Comput. 21*, 1047–1069.

GALIL, Z., AND ITALIANO, G. F. 1993. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput. 22*, 11–28.

HAN, X., KELSEN, P., RAMACHANDRAN, V., AND TARJAN, R. E. 1995. Computing minimal spanning subgraphs in linear time. *SIAM J. Comput. 24*, 1332–1358.

HARARY, F. 1969. *Graph Theory*. Addison-Wesley, Reading, Mass.

HENZINGER, M. R., AND KING, V. 1995a. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*. IEEE, New York.

HENZINGER, M. R., AND KING, V. 1995b. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing* (Las Vegas, Nev., May 29–June 1). ACM, New York, pp. 519–527.

HENZINGER, M. R., AND KING, V. 1997. Maintaining minimum spanning trees in dynamic graphs. In *Proceedings of the 24th International Colloquium Automata, Languages, and Programming*. Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany.

HENZINGER, M. R., AND LA POUTRÉ, J. A. 1995. Certificates and fast algorithms for biconnectivity in fully dynamic graphs. In *Proceedings of the 3rd European Symposium on Algorithms*.

HOPCROFT, J., AND TARJAN, R. E. 1973. Dividing a graph into triconnected components. *SIAM J. Comput. 2*, 135–158.

KANEVSKY, A., TAMASSIA, R., DI BATTISTA, G., AND CHEN, J. 1991. On-line maintenance of the four-connected components of a graph. In *Proceedings of the 32nd IEEE Symposium Foundations of Computer Science*. IEEE, New York, pp. 793–801.

KARGER, D. R., KLEIN, P. N., AND TARJAN, R. E. 1995. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM 42*, 321–328.

KHANNA, S., MOTWANI, R., AND WILSON, R. H. 1996. On certificates and lookahead on dynamic graph problems. In *Proceedings of the 7th ACM–SIAM Symposium on Discrete Algorithms* (Atlanta, Ga., Jan. 28–30). ACM, New York, pp. 222–231.

KING, V. 1995. A simpler minimum spanning tree verification algorithm. In *Proceedings of the 4th Workshop Algorithms and Data Structures*. pp. 440–448.

LA POUTRÉ, J. A. 1991. Dynamic graph algorithms and data structures. Ph.D. Dissertation. Department of Computer Science, Utrecht Univ.

LAWLER, E. L. 1976. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart and Winston, New York.

NAGAMOCHI, H., AND IBARAKI, T. 1992. Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica 7*, 583–596.

RAUCH, M. H. 1995. Fully dynamic biconnectivity in graphs. *Algorithmica 13*, 6 (June), 503–538.

SLEATOR, D. D., AND TARJAN, R. E. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci. 24*, 362–381.

TARJAN, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*, 146–160.

THURIMELLA, R. 1989. Techniques for the design of parallel graph algorithms. Ph.D. Dissertation. Univ. Texas, Austin, Austin, Tex.

WESTBROOK, J., AND TARJAN, R. E. 1992. Maintaining bridge-connected and biconnected components on-line. *Algorithmica 7*, 433–464.

RECEIVED JANUARY 1993; REVISED FEBRUARY 1996 AND MAY 1996; ACCEPTED JULY 1997