Improved Sparsification

David Eppstein* Zvi Galil† Giuseppe F. Italiano‡

Tech. Report 93-20

Department of Information and Computer Science
University of California, Irvine, CA 92717

April 15, 1993

Abstract

In previous work we introduced *sparsification*, a technique that transforms fully dynamic algorithms for sparse graphs into ones that work on any graph, with a logarithmic increase in complexity. In this work we describe an improvement on this technique that avoids the logarithmic overhead. Using our improved sparsification technique, we keep track of the following properties: minimum spanning forest, best swap, connectivity, 2-edge-connectivity, and bipartiteness, in time $O(n^{1/2})$ per edge insertion or deletion; 2-vertex-connectivity and 3-vertex-connectivity, in time $O(n)$ per update; and 4-vertex-connectivity, in time $O(n\alpha(n))$ per update.

*Department of Information and Computer Science, University of California, Irvine, CA 92717. Work supported in part by NSF grant CCR-9258355.

†Department of Computer Science, Columbia University, New York, NY 10027 and Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel. Work supported in part by NSF Grants CCR-9014605 and CDA-9024735.

‡IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598. On leave from Università di Roma.

1 Introduction

A *fully dynamic* graph algorithm is one that allows edge insertions and deletions, and recomputes some desired graph property quickly after each such update. In a previous paper [4], we introduced *sparsification*, a general technique that improves many such algorithms. Using this technique we were able to improve time bounds of the form $O(m^c)$ for such problems, into the form $O(n^c \log(m/n))$. We used sparsification to improve algorithms for many problems related to minimum spanning forests, edge and vertex connectivity, and bipartiteness. Apart from the logarithmic term $O(\log(m/n))$, the resulting time bounds for general graphs matched the best that could previously have been achieved only for sparse graphs.

An open problem remaining from that work was to close the remaining gap between sparse and general graphs, and eliminate the $O(\log(m/n))$ term from the time bound. In this paper we provide a method of doing so. Using this idea we can further improve the time bounds of algorithms for many of the same minimum spanning forest, connectivity, and bipartiteness problems previously solved by sparsification.

Sparsification itself worked by recursively partitioning the edges of the graph into a tree of subgraphs, and representing each node of the tree by a sparse *certificate*. The partition itself was quite arbitrary. For our new results, we use the same edge partition idea of our original sparsification paper, but we guide the edge partition by a corresponding partition of the vertices. In this way we can guarantee that the sizes of the certificates involved in any single update decrease in a geometric progression, and eliminate the $O(\log(m/n))$ overhead term found in our original sparsification results. We improve any sparsification based dynamic graph algorithm previously running in time $O(n^c \log(m/n))$, for some $c > 0$, to $O(n^c)$ per update.

Related ideas have been used in two special cases of sparsification. First, in an application of general graph sparsification to a geometric minimum spanning tree problem [1], we were able to eliminate the logarithmic overhead, by partitioning vertices as well as edges. In this case the vertex partition occurred naturally as part of the recursive definition of the graph on which sparsification was applied. Second, we later used an idea similar to sparsification, but based on partitions by separators, to achieve improvements in many dynamic algorithms for planar graphs [5]. In that paper as well, the partition of vertices arose naturally, in this case from the use of known separator theorems. Again the number of vertices in each partition decreased appropriately and we were able to avoid the logarithmic

factor. Our results in this paper can be seen as a generalization of the vertex partition techniques used in those two situations, to cover all possible input graphs. However we do not require any particularly natural vertex partition: we show that, as in our previous paper where any edge partition performed well, here any vertex partition suffices in our improved sparsification technique.

2 New Results

We describe fully dynamic data structures for the following problems.

1. We maintain the minimum spanning forest of an edge-weighted graph in $O(n^{1/2})$ time per edge insertion, deletion, or weight change, improving a previous $O(n^{1/2}\log(m/n))$ bound [4]. Applications of this result include algorithms for selecting random spanning trees [6] and maintaining color-constrained minimum spanning trees [9].

2. 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 $O(m\log\beta(m,n)+kn^{1/2})$. The best previous times were $O(n^{1/2}\log(m/n))$ and $O(m\log\beta(m,n)+kn^{1/2}\log(m/n))$ respectively [4].

3. 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 $O(n^{1/2}\log(m/n))$ [4].

4. 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 $O(n^{1/2}\log(m/n))$ bound per update [4].

5. We maintain the 2- and 3-vertex connected components of a graph in time $O(n)$ per update, improving a previous bound of $O(n\log(m/n))$ [4]. Rauch [12] proved an incomparible time bound of $O(m^{2/3})$ per update and $O(1)$ per query for fully dynamic 2-vertex connectivity.

6. We maintain the 4-vertex-connected components of a graph in time $O(n\alpha(n))$ per update, improving a previous bound of $O(n\log(m/n)+n\alpha(n))$ [4].

7. We maintain a bipartition of a graph, or determine that the graph is not bipartite, in $O(n^{1/2})$ time per update, improving a previous bound of $O(n^{1/2}\log(m/n))$ [4].

3 Sparsification

In this section we review the definition of sparsification from our earlier paper. Sparsification is based on the notion of a *strong certificate*:

Definition 1. *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.*

A graph 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.

The original sparsification technique maintains a partition of the edges of the graph into $\lceil m/n \rceil$ groups, all but one of which contain exactly n edges. We form a complete binary tree with leaves corresponding to the groups. Each node in this *sparsification tree* corresponds to a subgraph formed by the edges in the groups at the leaves of the tree that are descendants of the given node. For each such subgraph, we maintain a sparse certificate of $\mathcal{P}$. The certificate at a given node is found by forming the union of the certificates at the two child nodes, and running the sparse certificate algorithm on this union. Each update will cause $O(1)$ changes in the structure of the tree, so when an edge is inserted or deleted, we change $O(1)$ groups. For each of these groups, and for their $\log(m/n)$ ancestors in the sparsification tree, we recompute a new sparse certificate. This results in a sparse certificate for the whole graph at the root of the tree. We then update the data structure for property $\mathcal{P}$, on the graph formed by the sparse certificate at the root of the tree. If the certificates and the data structure can both be constructed in linear time, the result is a fully dynamic algorithm with update time $O(n\log(m/n))$. In this paper we show that with the same assumptions this result can be improved to $O(n)$ time per update.

In a variant of our original sparsification technique, we update each certificate quickly using a dynamic data structure instead of a static computation, by taking advantage of a *stability* property of certain certificates.

Definition 2. *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 stable sparsification technique assumes the existence of a fully dynamic data structure for maintaining the desired stable certificates. As before, we group the edges into $\lceil m/n \rceil$ groups of n edges each, arranged in a complete binary tree. We propagate changes up the tree, using the data structure for maintaining certificates, and update the data structure for property $\mathcal{P}$, 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 two certificates in the two child nodes. It follows from stability that at each level of the sparsification tree there is a constant amount of change, and so the update to the certificate at each level can be performed with $O(1)$ data structure operations. If the certificate, and a data structure for maintaining property $\mathcal{P}$ itself, can be maintained in time $O(m^c)$, this technique improves the bound to $O(n^c \log(m/n))$. In this paper we provide a further improvement, to $O(n^c)$.

4 Improved Sparsification

In the original sparsification technique, the edge partition was arbitrary, and the graphs induced at each node of the sparsification tree could have many vertices. We use a different partition technique, similar to the *2-dimensional topology tree* of Frederickson [7, 8] 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 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 $n/2^i$ in each of V_α and V_β. □

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, and also in stable sparsification to make sure that $f(n)$ is large enough for our improvement to work. All polynomials are well-behaved.

Theorem 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 induced by 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 [4], 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 $O(f(n,O(n))) + g(n,cn)$. □

Theorem 2. *Let $\mathcal{P}$ be a property for which stable sparse certificates can be maintained in time $f(n,m)$ per update for some well-behaved f, 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 out as in [4] so the net effect of the update will be at most s changes in the certificate.

The times to change each certificate then add in a geometric series to give the stated bound. □

Theorem 1 can be used to dynamize static algorithms, while Theorem 2 can be used to speed up existing fully dynamic algorithms. In order to apply effectively Theorem 1 we only need to *compute* efficiently *sparse* certificates, while for Theorem 2 we need to *maintain* efficiently *stable sparse* certificates.

5 Better space and preprocessing

In the original version of sparsification [4], the space and preprocessing time were linear (assuming the same to be true of the certificate computation

algorithm). In the improved sparsification presented above, the time has been improved by a logarithmic factor, but the space and preprocessing time will now be increased by a similar factor. Indeed, the best one can show is a space and preprocessing bound of $O(m \log(n^2/m))$.

In this section we combine the vertex partition of our improved sparsification with the edge partition of our original sparsification, to create a hybrid algorithm that will have the best time and space bounds of both algorithms.

The idea is simple: we use the improved sparsification tree 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 linear space sparsification method of our previous paper [4].

The time for computing or updating certificates in the upper levels of the sparsification tree is $O(f(n))$ by Theorem 1 or Theorem 2 The time in lower levels is $O(f(n^2/m) \log(m^2/n^2))$ by results of [4]; with the assumption of well-bahavedness this can be rewritten $O(f(n)(n/m)^{\log c} \log(m/n)) = O(f(n))$.

We state our results more formally. The assumption of upwards convexity is needed in order to bound the amount of space used in a collection of data structures at different nodes in the sparsification tree, with different numbers of vertices adding to a certain total.

Theorem 3. *The time bound of Theorem 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 1, or either space or preprocessing in Theorem 2, if the bounds on the individual processing or space at a node of the sparsification tree are given by an upwards convex function $h(n)$, the total bound will be $O(\frac{m}{n} h(n))$.*

6 Applications

The improved sparsification technique can be applied to a wide variety of problems. We first discuss several problems related to minimum spanning forests.

Theorem 4. *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 2, with $f(n,m) = g(n,m) = O(m^{1/2})$ by the clustering algorithm of Frederickson [7]. □

The *best swap* in a graph is the pair of edges such that if they are respectively added to and removed from the minimum spanning forest, the result is again a spanning forest with minimum possible weight. Best swaps have been used to enumerate the spanning forests of a graph in order by weight [8]. For this application one needs a *fully persistent* data structure (one in which any update does not modify previous versions of the data structure, but instead creates a new separately existing version, and in which each update or query can be performed in any of the versions that have been so created).

Theorem 5. *The best swap of an undirected graph can be maintained with full persistence in time $O(n^{1/2})$ per update.*

Proof: As in [4], we use the union of the two smallest spanning trees as a certificate, maintained by Frederickson's minimum spanning tree algorithm, and use Frederickson's [8] best swap data structure at the root of the sparsification tree. And as in [4, 8], it is not difficult to modify both the data structures used at each node of the tree, and our sparsification technique itself, to support full persistence. □

Theorem 6. *The k best spanning trees of an undirected graph can be found in a total of $O(m \log \beta(m,n) + k \min(n,k)^{1/2})$ time.*

Proof: As Frederickson [8] shows, this problem can be solved using $O(k)$ operations in the persistent best swap data structure. A graph reduction technique of the first author [3] replaces n by $\min(n,k)$ and m by $\min(m,k)$ in the time bounds at the expense of an additive $O(m \log \beta(m,n))$ term.

With the same graph reduction technique, the best swap data structure can be constructed in time $O(\min(m,k) \log \min(n,k) \log \log^* \min(n,k))$. With a more complicated sparsification tree in which our original structured tree is compressed to avoid nodes containing few edges, we can eliminate the $O(\log \min(n,k))$ term in this bound, however such an improvement is not necessary as the setup time is dominated by the $O(k \min(n,k)^{1/2})$ term. □

Corollary 1. *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 $O(n^{2-2/(\lceil d/2 \rceil + 1) + \epsilon} + k n^{1/2})$.*

Proof: In an earlier work [2] we reduced these geometric problems to the general graph problem in these time bounds. □

Feder and Mihail [6] use a random walk technique to select uniformly distributed spanning trees of a graph. Their walk either removes a random edge from the spanning tree, or adds a random edge making the current forest into a spanning tree. As shown in [4], this can be done by using a dynamic minimum spanning tree algorithm with weights determined by a separately maintained random permutation on non-tree edges.

Theorem 7. *We can implement the random walk of Feder and Mihail* [6] *on the spanning forests of a given graph, in time $O(n^{1/2})$ per step.*

In the *color-constrained minimum spanning tree problem*, 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. Frederickson and Srinivas [9] show that this can be done in time $O(d^2U(m,n) + d^{11/3}(d!)^2n^{1/3}\log n)$ time per update, where $U(m,n) = O(m^{1/2})$ is the time to update a minimum spanning tree after an edge insertion or deletion.

Theorem 8. *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.*

We next discuss a variety of connectivity problems.

Theorem 9. *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 2 with Frederickson's minimum spanning tree algorithm. As noted by Frederickson [8], connectivity queries in his data structure can be performed in constant time. □

Theorem 10. *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 [7] to maintain a stable strong certificate consisting of two edge-disjoint spanning

trees [4], giving $f(n,m) = O(m^{1/2})$. 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 $q(n,m) = O(\log n)$ [8]. Applying Theorem 2 gives our result. □

Theorem 11. *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: We use Theorem 1 with a certificate consisting of two, three, or four edge-disjoint breadth first search trees, constructed in linear time [4]. 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 $k = 3$ [10, 13]. 4-vertex-connectivity can be tested in time $g(n,m) = O(m + n\alpha(n))$ [11]. □

Finally, we use our technique to maintain dynamic information about the bipartiteness of a graph.

Theorem 12. *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: In the full version of [4] we show that a graph formed by adding to the minimum spanning forest the shortest edge inducing an odd cycle is a stable certificate of bipartiteness, and can be maintained in $O(m^{1/2})$ time using Frederickson's clustering techniques [7, 8]. We use these certificates in Theorem 2. □

References

[1] P. K. Agarwal, D. Eppstein, and J. Matoušek. Dynamic algorithms for half-space reporting, proximity problems, and geometric minimum spanning trees. In *Proc. 33rd IEEE Symp. Foundations of Computer Science*, pages 80–89, 1992.

[2] D. Eppstein. Tree-weighted neighbors and geometric k smallest spanning trees. *Int. J. Comput. Geom. & Appl.* To appear.

[3] D. Eppstein. Finding the k smallest spanning trees. *BIT*, 32:237–248, 1992.

[4] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Symp. Foundations of Computer Science*, pages 60–69, 1992.

[5] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification for dynamic planar graph algorithms. In *Proc. 25th ACM Symp. Theory of Computing*, pages 208–217, 1993.

[6] T. Feder and M. Mihail. Balanced matroids. In *Proc. 24th ACM Symp. Theory of Computing*, pages 26–38, 1992.

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

[8] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *Proc. 32nd Symp. Foundations of Computer Science*, pages 632–641, 1991.

[9] G. N. Frederickson and M. A. Srinivas. Algorithms and data structures for an expanded family of matroid intersection problems. *SIAM J. Comput.*, 18:112–138, 1989.

[10] J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2:135–158, 1973.

[11] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. On-line maintenance of the four-connected components of a graph. In *Proc. 32nd IEEE Symp. Foundations of Computer Science*, pages 793–801, 1991.

[12] M. Rauch. Fully dynamic biconnectivity in graphs. In *Proc. 33rd IEEE Symp. Foundations of Computer Science*, pages 50–59, 1992.

[13] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.