Technical Report DIKU-TR-97/17
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 KBH Ø
DENMARK

September 1997

Poly-Logarithmic Deterministic Fully-Dynamic Graph Algorithms I: Connectivity and Minimum Spanning Tree

Jacob Holm Kristian de Lichtenberg Mikkel Thorup

Poly-logarithmic deterministic fully-dynamic graph algorithms I: connectivity and minimum spanning tree

Jacob Holm* Kristian de Lichtenberg* Mikkel Thorup*

Abstract

Deterministic fully dynamic graph algorithms are presented for connectivity and minimum spanning forest. For connectivity, starting with no edges, the amortized cost for maintaining a spanning forest is $O(\log^2 n)$ per update, i.e. per edge insertion or deletion. Deciding connectivity between any two given vertices is done in $O(\log n/\log\log n)$ time. This matches the previous best randomized bounds. The previous best deterministic bound was $O(\sqrt[3]{n}\log n)$ amortized time per update but constant time for connectivity queries.

For minimum spanning trees, first a deletions-only algorithm is presented supporting deletes in amortized time $O(\log^2 n)$. Applying a general reduction from Henzinger and King, we then get a fully dynamic algorithm such that starting with no edges, the amortized cost for maintaining a minimum spanning forest is $O(\log^4 n)$ per update. The previous best deterministic bound was $O(\sqrt[3]{n}\log n)$ amortized time per update, and no better randomized bounds were known.

Corresponding $O(\log^2 n)$ algorithms for 2-edge connectivity and biconnectivity will be presented in a subsequent report.

1 Introduction

We consider the fully dynamic graph problems of connectivity and of minimum spanning forest. In a *fully dynamic graph problem*, we are considering a graph G over a fixed vertex set V, $|V| = n$. The graph G may be *updated* by insertions and deletions of edges. Unless otherwise stated, we assume that we start with an empty edge set.

For the *fully dynamic connectivity problem*, the updates may be interspersed with *connectivity queries*, asking whether two given vertices are connected in G.

*E-mail:`(samson,morat,mthorup)@diku.dk`. Department of Computer Science, University of Copenhagen.

Both updates and queries are presented *on-line*, meaning that we have to respond to an update or query without knowing anything about the future.

In the *fully dynamic minimum spanning tree problem*, we have weights on the edges, and we wish to maintain a minimum spanning forest F of G. Thus, in connection with any update to G, we need to respond with the corresponding updates for F, if any.

The connectivity problem reduces to the spanning tree problem in that if we can maintain *any* spanning forest F for G at cost $O(t(n) \log n)$ per update, then, using dynamic trees [10], we can answer connectivity queries in time $O(\log n / \log t(n))$.

In this paper, we first show a very simple deterministic algorithm for maintaining a spanning forest in a graph in amortized time $O(\log^2 n)$ per update. Connectivity queries are then answered in time $O(\log n / \log \log n)$.

Second, we derive a quite simple deterministic algorithm for maintaining a minimum spanning forest in a graph in amortized time $O(\log^4 n)$.

History For deterministic algorithms, all the previous best solutions to the fully dynamic connectivity problem were also solutions to the minimum spanning tree problem. In 1985 [5], Frederickson introduced a data structure known as *topology trees* for the fully dynamic minimum spanning tree problem with a worst case cost of $O(\sqrt{m})$ per update, permitting connectivity queries in time $O(\log n / \log(\sqrt{m}/\log n)) = O(1)$. In 1992, Epstein et. al. [3, 2] improved the update time to $O(\sqrt{n})$ using the *sparsification technique*. Finally in 1997 Henzinger and King [8] gave an algorithm with $O(\sqrt[3]{n} \log n)$ update time and $O(\log n / \log \log n)$ time per connectivity query.

Randomization has been used to improve the bounds for the connectivity problem. In 1995 [6], Henzinger and King showed that a spanning forest could be maintained in $O(\log^3 n)$ expected amortized time per update. Then connectivity queries are supported in $O(\log n / \log \log n)$ time. The update time was further improved to $O(\log^2 n)$ in 1996 [9] by Henzinger and Thorup. No randomized technique was known for improving the deterministic $O(\sqrt[3]{n} \log n)$ update cost for the minimum spanning tree problem.

As mentioned, we present a deterministic fully dynamic connectivity algorithm with an update cost of $O(\log^2 n)$, thus matching the previous best randomized bound and improving substantially over the previous best deterministic bound of $O(\sqrt[3]{n} \log n)$. For the minimum spanning tree problem our deterministic update cost of $O(\log^4 n)$ improves substantially over the previous bound of $O(\sqrt[3]{n} \log n)$, and here no better randomized bound was known.

The result is achieved in three steps: First, we give a deterministic fully dynamic connectivity algorithm that uses $O(\log^2 n)$ amortized time per update and $O(\log n / \log \log n)$ time per query. Then, we extend this algorithm to give a deletions-only minimum spanning tree data structure supporting deletes in

$O(\log^2 n)$ amortized time. Finally we use a technique from [8] to convert our deletions-only structure to a fully dynamic data structure for the minimum spanning tree problem using $O(\log^4 n)$ amortized time per update. Our technique relies on some of the same intuition as was used in Henzinger and King [6] in their randomized algorithm. Our deterministic algorithm is, however, much simpler, and in contrast to their algorithm, it generalizes to the minimum spanning tree problem.

The reader is referred to [1, 3, 4, 6] for discussions of problems that get improved by our improvements for the fully dynamic connectivity and minimum spanning tree problems.

2 Connectivity

In this section, we present an $O(\log^2 n)$ time deterministic fully dynamic algorithm for graph connectivity. First we give a high level description, ignoring all problems concerning data structures. Second, we implement the algorithm with concrete data structures and analyze the running times.

2.1 High level description

Our dynamic algorithm maintains a spanning forest F of a graph G. The edges in F will be referred to as *tree-edges*. Internally, the algorithm associates with each edge e a level $\ell(e) \leq L = \lfloor \log_2 n \rfloor$. For each i, F_i denotes the sub-forest of F induced by edges of level at least i. Thus, $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$. The following invariants are maintained.

(i) F is a maximum (w.r.t. ℓ) spanning forest of G, that is, if (v, w) is a non-tree edge, v and w are connected in $F_{\ell(v,w)}$.

(ii) The maximal number of nodes in a tree in F_i is $n/2^i$. Thus, the maximal relevant level is L.

Initially, all edges have level 0, and hence both invariants are satisfied. We are going to present an amortization argument based on increasing the levels of edges. The levels of edges are never decreased, so we can have at most L increases per edge. Intuitively speaking, when the level of a non-tree edge is increased, it is because we have discovered that its end points are close enough in F to fit in a smaller tree on a higher level. Concerning tree edges, note that increasing their level cannot violate (i), but it may violate (ii).

We are now ready for a high-level description of insert and delete.

Insert(e) The new edge is given level 0. If the end-points were not connected in $F = F_0$, e is added to F_0.

Delete(e) If e is not a tree-edge, it is simply deleted. If e is a tree-edge, it is deleted and a *replacement edge*, reconnecting F at the highest possible level, is searched for. Since F was a maximum spanning forest, we know that the replacement edge has to be of level at most $\ell(e)$. We now call Replace$(e, \ell(e))$. Note that when a tree-edge e is deleted, F may no longer be spanning, in which case (i) is violated until we have found a replacement edge. In the time in between, if (v, w) is not a replacement edge, we still have that v and w are connected in $F_{\ell(v,w)}$.

Replace$((v, w), i)$ Assuming that there is no replacement edge on level $> i$, finds a replacement edge of the highest level $\leq i$, if any.

Let T_v and T_w be the trees in F_i containing v and w, respectively. Assume, without loss of generality, that $|T_v| \leq |T_w|$. Before deleting (v, w), $T = T_v \cup \{(v, w)\} \cup T_w$ was a tree on level i with at least twice as many nodes as T_v. By (ii), T had at most $n/2^i$ nodes, so now T_v has at most $n/2^{i+1}$ nodes. Hence, preserving our invariants, we can take all edges of T_v of level i and increase their level to $i + 1$, so as to make T_v a tree in F_{i+1}.

Now level i edges incident to T_v are visited one by one until either a replacement edge is found, or all edges have been considered. Let f be an edge visited during the search.

If f does not connect T_v and T_w, we increase its level to $i+1$. This increase pays for our considering f.

If f does connect T_v and T_w, it is inserted as a replacement edge and the search stops.

If there are no level i edges left, we call Replace$((v, w), i-1)$; except if $i = 0$, in which case we conclude that there is no replacement edge for (v, w).

2.2 Implementation

For each i, we wish to maintain the forest F_i together with all non-tree edges on level i. For any vertex v, we wish to be able to find the tree T_v in F_i containing it. We want to be able to compute the size of T_v. We want to be able to find an edge of T_v on level i, if one exists. Finally, we want to be able to find a level i non-tree edge incident to T_v, if any.

The trees in F_i may be cut (when an edge is deleted) and linked (when a replacement edge is found, an edge is inserted or the level of a tree edge is increased). Moreover, non-tree edges may be introduced and any edge may disappear on level i (when the level of an edge is increased or when non-tree edges are inserted or deleted).

All the above operations and queries may be supported in $O(\log n)$ time using the ET-trees from [6], to which the reader is referred for additional details. An

ET-tree is a standard balanced binary tree over the Euler tour of a tree. Each node in the ET-tree represents the segment of the Euler tour below it. The point in considering Euler tours is that if trees in a forest are linked or cut, the new Euler tours can be constructed by at most 2 splits and 2 concateations of the original Euler tours. Rebalancing the ET-trees affects only $O(\log n)$ nodes.

Here we have an ET-tree over each tree in F_i. Each node of the ET-tree contains a number telling the size of the Euler tour segment below it, a bit telling if any tree edges in the segment have level i, and a bit telling whether there is any level i non-tree edges incident to a vertex in the segment.

Given a vertex v we can find the tree T_v containing v by moving $O(\log n)$ steps up till we find a root of an ET-tree. This root represents the Euler tour of T_v. The size s of the Euler tour of a tree is twice the number of edges, so the number of vertices is $s/2+1$. To find a tree edge of level i or an incident non-tree edge, if any, we move $O(\log n)$ steps down the ET-tree, using the bits telling us under which nodes such edges are to be found. If a tree edge (v, w) is moved from level i, we only need to update the bits on the paths from (v, w) and (w, v) to the root, using $O(\log n)$ time. If a non-tree edge (v, w) is introduced/disappear, we only need to update the bits on the paths from v and w to their respective roots. This takes $O(\log n)$ time. When the trees are cut or linked, only $O(\log n)$ nodes are affected, and the information in each node is updated in constant time.

It is now straightforward to analyze the amortized cost of the different operations. When an edge e is inserted on level 0, the direct cost is $O(\log n)$. However, its level may increase $O(\log n)$ times, so the amortized cost is $O(\log^2 n)$.

Deleting a non-tree edge e takes time $O(\log n)$. When a tree edge e is deleted, we have to cut all forests F_j, $j \leq \ell(e)$, giving an immediate cost of $O(\log^2 n)$. We then have $O(\log n)$ recursive calls to Replace, each of cost $O(\log n)$ plus the cost amortized over increases of edge levels. Finally, if a replacement edge is found, we have to link $O(\log n)$ forests, in $O(\log^2 n)$ total time.

Thus, the cost of inserting and deleting edges from G is $O(\log^2 n)$. The balanced binary tree over $F_0 = F$ immediately allows us to answer connectivity queries between arbitrary nodes in time $O(\log n)$. In order to reduce this time to $O(\log n/\log\log n)$, as in [6], we introduce an extra balanced $\Theta(\log n)$-ary B-tree over the Euler tour of each tree in F. The B-tree has depth $O(\log n/\log\log n)$, which is hence the time it takes for a connectivity query. Each delete or insert gives rise to at most one cut and one link in F, and for $\Theta(\log n)$-ary B-trees, such operations can be supported in $O(\log^2 n/\log\log n)$ time. Thus, we conclude:

Theorem 1 *Given a graph G with m edges and n vertices, there exists a deterministic fully dynamic algorithm that answers connectivity queries in $O(\log/\log\log n)$ time worst case, and uses $O(\log^2 n)$ amortized time per insert or delete.*

3 Minimum spanning forest

We will now expand on the ideas from the previous section to the problem of maintaining a minimum spanning forest (MSF). First we present an $O(\log^2 n)$ deletions-only algorithm, and then we apply a general construction from [8] transforming a deletions-only MSF algorithm into a fully dynamic MSF algorithm.

3.1 Decremental minimum spanning forests

It turns out that if we only want to support deletions, we can obtain an MSF-algorithm from our connectivity algorithm by some very simple changes. The first is, of course, the initial spanning forest F has to be a minimal spanning forest. The second is that when in replace (cf. page 4). we consider the level i non-tree edges incident to T_v, instead of doing it in an arbitrary order, we should do it in order of increasing weights. That is, we repeatedly take the lightest incident level i edge e: if e is a replacement edge, we are done; otherwise, we move e to level $i+1$, and repeat with the new lightest incident level i edge, if any. For the above changes to work, it is crucial, that *all weights are distinct*. To ensure this, we associate a unique number with each edge. If two edges have the same weight, it is the one with the smaller number that is the smaller.

To see that the above simple changes suffice to maintain that F is a minimum spanning forest, we will prove that in addition to (i) and (ii), the following invariant is maintained:

(iii) If e is the heaviest edge on a cycle C, then e has the lowest level on C.

The original replace function found a replacement edge on the highest possible level, but now, among the replacement edges on the highest possible level, we choose the one of minimum weight. Using (iii), we will show that this edge has minimum weight among all replacement edges.

Lemma 2 *For any tree edge e, among all replacement edges, the lightest edge is on the maximum level.*

Proof: Let e_1 and e_2 be replacement edges for e. Let C_i be the cycle induced by e_i; then $e \in C_i$. Suppose e_1 is lighter than e_2. We want to show that $\ell(e_1) \geq \ell(e_2)$.

Consider the cycle $C = (C_1 \cup C_2) \setminus (C_1 \cap C_2)$. Since F is a minimum spanning forest, we know that e_i is the heaviest edge on C_i. Hence e_2 is the heaviest edge on C. By (iii) this implies that e_2 has the lowest level on C. In particular, $\ell(e_1) \geq \ell(e_2)$. □

Since our algorithm is just a specialized version of the decremental connectivity algorithm, we already know that (i) and (ii) are maintained.

Lemma 3 *(iii) is maintained.*

Proof: Initially (iii) is satisfied since all edges are on level 0. We will now show that (iii) is maintained under all the different changes we make to our structure during the deletion of an edge. If an edge e is just deleted, any cycle in $G \setminus \{e\}$ also existed in G, so (iii) is trivially preserved. Also note that replacing a deleted tree-edge cannot in itself violate (iii) since it does not change the levels or weights of any edges.

Our real problem is to show that (iii) is preserved during Replace when the level of an edge e is increased. This cannot violate (iii) if e is not the heaviest edge on some cycle, so assume that e is the heaviest edge on a cycle C. To prove that (iii) is not violated, we want to show that before the increase, all other edges in C have level $\geq i+1$.

No tree edge is heaviest on any cycle, so e is a non-tree edge. When $\ell(e)$ is to be increased from i to $i+1$, we know it is the lightest level i edge incident to T_v (cf. the description of replace on page 4). Moreover, by (iii), all other edges on C have level at least i. Thus, all other edges from C incident to T_v have level at least $i+1$.

To complete the proof, we show that all edges in C are incident to T_v. Suppose, for a contradiction, that C contained an edge f leaving T_v. Since e is to be increased, $e \neq f$. Also, the call to Replace requires that there is no replacement edge of level $> i$, so $\ell(f) \leq i$. This contradicts that all edges $\neq e$ from C incident to T_v have level $\geq i+1$. □

It has now been established that the above change in replace suffices to maintain a minimum spanning forest. A last point is that we need to modify our ET-trees to give us the lightest non-tree edge incident to a tree. So far, for each node in the ET-trees, we had a bit telling us whether the Euler tour segment below it had an incident non-tree edge. Now, with the node, we store the minimum weight of a non-tree edge incident to the Euler tour segment below it. Clearly, we can still support the different operations in $O(\log n)$ time. We conclude

Theorem 4 *There exists a deletions-only MSF algorithm that can be initialized on a graph with n nodes and m edges and support any sequence of $\Omega(m)$ deletions in total time $O(m \log^2 n)$.* □

3.2 Fully dynamic MSF

To obtain a fully dynamic minimum spanning forest algorithm we apply a general reduction, which is a slight generalization of the one provided by Henzinger and King [8, pp. 600-603]. The reduction is described as follows.

Lemma 5 *Suppose we have a deletions-only MSF algorithm that for any k, l, can be initialized on a graph with k nodes and l edges and support any sequence of $\Omega(l)$ deletions in total time $O(l \cdot t(k,l))$ where t is non-decreasing. Then there* *exists a fully-dynamic MSF algorithm for a graph on n nodes starting with no edges, that for m edges, supports an update in amortized time*

$$O\left(\log^3 n + \sum_{i=1}^{3+\log_2 m} \sum_{j=1}^{i} t(\min\{n, 2^j\}, 2^j)\right).$$

"Proof": Essentially, we combine the reduction from [8] with a contraction idea from [7]. We will only sketch the changes needed in [8]. As in [8], we operate on a series of graphs A_i, where A_i has 2^i non-tree edges. In [8], A_i may have $n-1$ MSF-edges, and this forces them to introduce a special efficient operation for adding a batch of edges. Here, instead, when we first create A_i, we contract all MSF-paths that are not incident to any non-tree edge. The "super" edge e replacing a MSF-path P gets the minimum weight on P. Moreover, if any edge from P is deleted, we have to delete e in A_i. As a result, we can base our fully-dynamic algorithm directly on deletions-only algorithms. □

From Theorem 4, we get $t(k,l) = O(\log^2 k)$, and hence we get a fully dynamic algorithm with update cost

$$O\left(\log^3 n + \sum_{i=1}^{3+\log_2 m} \sum_{j=1}^{i} \log^2(\min\{n, 2^j\})\right) = O(\log^4 n).$$

Note for comparison, that in [8], Henzinger and King had $t(k,l) = O(\sqrt[3]{l}\log k)$, giving them an update cost of $O(\sqrt[3]{m}\log n)$. Then sparsification [2, 3] reduces the cost to $O(\sqrt[3]{n}\log n)$. From the combination of Theorem 4 and Lemma 5, we conclude

Theorem 6 *There is a fully-dynamic MSF algorithm that for a graph with n nodes and starting with no edges maintains a minimum spanning forest in $O(\log^4 n)$ amortized time per edge insertion or deletion.* □

References

[1] D. Eppstein. Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:237–250, 1995.

[2] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Improved sparsification. Technical Report 93-20, Univ. of California, Irvine, Dept. Information and Computer Science, 1993.

[3] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. In *Proc. 33rd Symp. Foundations of Computer Science*, pages 60–69. IEEE, 1992.

[4] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, January 1981.

[5] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985.

[6] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing*, pages 519–527, 1995.

[7] M. R. Henzinger and V. King. Fully dynamic 2-edge connectivity algorithm in polygarithmic time per operation. Technical report, Digital, 1997. A preliminary version appeared as [6].

[8] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 594–604, 1997.

[9] Monika Rauch Henzinger and Mikkel Thorup. Improved sampling with applications to dynamic graph algorithms. In *Proceedings of the 23rd International Colloquium on Automata Languages, and Programming (ICALP), LNCS 1099*, pages 290–299, 1996.

[10] D D Sleator and Robert E Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–390, 1983.