Maintaining Minimum Spanning Trees in Dynamic Graphs

Monika R. Henzinger[1] and Valerie King[2]

[1] Systems Research Center, Digital Equipment Corporation, 130 Lytton Ave, Palo Alto, CA, USA 94301; email: monika@pa.dec.com.
[2] University of Victoria, Dept. of Computer Science, P.O. Box 3055, Victoria, BC., Canada, V8W 3P6; email: val@csr.uvic.ca

Abstract. We present the first fully dynamic algorithm for maintaining a minimum spanning tree in time $o(\sqrt{n})$ per operation. To be precise, the algorithm uses $O(n^{1/3} \log n)$ amortized time per update operation. The algorithm is fairly simple and deterministic. An immediate consequence is the first fully dynamic deterministic algorithm for maintaining connectivity and, bipartiteness in amortized time $O(n^{1/3} \log n)$ per update, with $O(1)$ worst case time per query.

1 Introduction

We consider the problem of maintaining a minimum spanning tree during an arbitrary sequence of edge insertions and deletions. Given an n-vertex graph G with edge weights, the *fully dynamic minimum spanning tree problem* is to maintain a minimum spanning forest F under an arbitrary sequence of the following update operations:

insert(u,v): Add the edge $\{u, v\}$ to G. Add $\{u, v\}$ to F if this reduces the cost of F, and return the edge of F that has been replaced.
delete(u,v): Remove the edge $\{u, v\}$ from G. If $\{u, v\} \in F$, then (a) remove $\{u, v\}$ from F and (b) return the minimum-cost edge e of $G \setminus F$ that reconnects F if e exists or return *null* if e does not exist.

In addition, the data structure permits the following type of query:

connected(u,v): Determine if vertices u and v are connected.

In 1985 [7], Fredrickson 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 His data structure permitted connectivity queries to be answered in $O(1)$ time. In 1992, Eppstein et. al. [3, 4] improved the update time to $O(\sqrt{n})$ using the *sparsification technique*. If only edge insertions are allowed, the Sleator-Tarjan dynamic tree data structure [13] maintains the minimum spanning forest in time $O(\log n)$ per insertion or query. If only edge deletions are allowed ("deletions-only"), then no algorithm faster than the $\Omega(\sqrt{n})$ fully dynamic algorithm was known.

Using randomization, it was recently shown that the fully dynamic connectivity problem, i.e., the restricted problem where all edge costs are the same, can be solved in amortized time $O(\log^2 n)$ per update and $O(\log n)$ per connectivity query [9, 10]. However, this approach could not be extended to arbitrary edge weights, leaving the question open as to whether the fully dynamic minimum spanning tree problem can be solved in time $o(\sqrt{n})$.

In this paper we give a positive answer to this question: We present a fully dynamic minimum spanning tree data structure that uses $O(n^{1/3}\log n)$ amortized time per update and $O(1)$ worst case time per query when update time is averaged over any sequence of $\Omega(m_{in})$ updates, for m_{in} the initial size of the graph. Our technique is very different from [7].

The result is achieved in two steps: First, we give a deletions-only minimum spanning tree algorithm that uses $O(m'^{1/3}\log n + n^{\epsilon})$ amortized time per update and $O(1)$ worst case time per query when the update time is averaged over any sequence of $\Omega(m_{in})$ updates. Here ϵ is any constant such that $0 < \epsilon < 1/3$, and m' is the number of *nontree* edges at the time of the update.

Then we present a general technique which, given a deletions-only minimum spanning tree data structure with a certain property, generates a fully dynamic data structure with the same running time as the deletions-only data structure. Let $f(m', n)$ be the amortized time per deletion in the deletions-only data structure with m' nontree edges and n vertices. The property required is that, upon inserting into the graph no more than m' edges at the same time (a "batch insertion"), the deletions-only data structure can be modified to reflect these insertions and up to m' subsequent deletions can be performed in a total of $O(m'f(m', n))$ time.

Using this technique, we develop a fully dynamic minimum spanning tree algorithm with *amortized* time per update of $O(m^{1/3}\log n)$, for a sequence of updates of length $\Omega(m_{in})$, where m is the size of G at the time of the update. In other words, letting $m_{(i)}$ denote the size of G (vertices and edges) after update i, the total amount of work for processing a sequence of updates of length l is $O(\sum_{i=0}^{l} m_{(i)}{}^{1/3}\log n)$. We then apply sparsification [3, 4] to reduce the running time for the sequence to $O(ln^{1/3}\log n)$.

Our result immediately gives faster *deterministic* fully dynamic algorithms for the following problems: connectivity, bipartiteness, k-edge witness, maximal spanning forest decomposition, and Euclidean minimum spanning tree. See [9] for all but the last reduction; see Eppstein [2] for the last reduction. For these problems, the new algorithm achieves an $O(n^{1/6}/\log n)$ factor improvement over the previously best *deterministic* running time. If randomization is allowed, however, much faster times are achievable [9, 10].

Additionally, improvements can be achieved in the following static problems (see [4, 3]): randomly sampling spanning forests of a given graph [6]; finding a color-constrained minimum spanning tree [8].

The paper is structured as follows: In Section 2 we give a deletions-only minimum spanning tree algorithm. In Section 3, we show how to use a sequence of deletions-only data structures to create a fully dynamic data structure.

2 Maintaining a minimum spanning tree–deletions-only

In this section, we give an algorithm which maintains a minimum spanning tree while edges are being deleted. The amortized update time is $O(m^{1/3} \log n)$ and the query time is $O(1)$ for queries of the form "Are vertices i and j connected?". Let $G = (V, E)$ be an undirected graph with edge weights. Without loss of generality, we assume that edge weights are distinct.

Initially, we compute the minimum spanning forest F of G. Let m'_{in} be the number of nontree edges in G initially and $k = {m'}_{in}^{1/3} \log n$. We sort the nontree edges by weight and partition them into m'_{in}/k levels of size k so that the k lightest are in level 0, the next k lightest are in level 1 and so on. The set of edges in a level i is denoted by E_i. In addition, all tree edges of the initial minimum spanning forest F are placed in level 0.

Throughout the algorithm, the level of an edge remains unchanged, and F denotes the minimum spanning forest. For $i = 0, 1, ..., (m'_{in}/k) - 1$, let F_i denote the minimum spanning forest of the graph with vertex set V and edgeset $\cup_{j \leq i} E_j$. (Initially, all $F_i = F$, but in later stages, an edge from any level may become a tree edge. Thus, $F_0 \subseteq F_1 \subseteq \ldots F_{(m'_0/k)-1} = F$.) Let $T_i(x)$ denote the tree in F_i which contains x and let $T(x)$ without the subscript denote the tree in F containing x.

The main idea is the following. If a nontree edge is deleted, then the minimum spanning forest F is unchanged. Suppose a tree edge $\{u, v\}$ in level i is deleted. Then for each F_j, $j \geq i$, the deletion splits the tree in F_j containing u and v into $T_j(u)$ and $T_j(v)$. We search for the minimum weight nontree edge e (called the "replacement edge") that connects $T(u)$ and $T(v)$ by gathering and then testing a set S of candidate edges on level i. If none is found, we repeat the procedure on level $i+1$, etc. until one is found or all levels are exhausted. We now describe the update operations:

delete(u, v): Delete edge $\{u, v\}$ from any data structures in which it occurs. If a tree edge $\{u, v\}$ from level i is deleted, then remove $\{u, v\}$ from F and search for a replacement by calling **Replace**(i, u, v). We refer to i as the level of the call to **Replace**.

In the algorithm below, the subroutine **Search** when applied to a tree in F_i finds all nontree edges in level i which are incident to the tree. A phase consists of the examination of a single edge. (Its exact definition and the details of **Search** are given in Section 2.2 below.)

Replace(i, u, v)

1. Alternating in lockstep, one phase at a time, **Search**$(T_i(u))$ and **Search**$(T_i(v))$ until $k/\log n$ phases are executed (Case A) or one of the searches has stopped (Case B).
   - Case A: Let S be the set of all nontree edges in level i.
   - Case B: Let S be the set of (nontree) edges produced by the **Search** that stopped.

2. Test every edge in S to see if it connects $T(u)$ and $T(v)$.
   - If a connecting edge is found, insert the minimum weight connecting edge into F and the data structures representing the F_j, $j \geq i$.
   - Else if i is not the last level, call **Replace**$(i+1, u, v)$.

2.1 Data Structures

The idea here is to use the ET-tree data structure developed in [9]: (1) to represent and update each tree in F, so that in constant time, we can quickly test if a given edge joins two trees; and (2) to represent each tree in an F_i in such a way that we can quickly retrieve nontree edges in E_i which are incident to the tree. To avoid excessive cost, we explicitly maintain only those F_i where i is a multiple of ${m'}_{in}^{1/3}/\log n$. An unpleasant consequence of this is that when retrieving nontree edges in E_i, other nontree edges are also retrieved.

Below, we refer to input graph vertices as "vertices" and use "node" to mean a nodes of the B-tree in which we store the "ET-sequences."

ET-trees: An *ET-sequence* is a sequence generated from a tree by listing each vertex each time it is encountered ("an occurrence of the vertex") as a tree is searched depth-first. Each ET-sequence is stored in a B-tree of degree d. This allows us to implement the deletion or insertion of an edge in the forest as follows: we split a tree by deleting an edge or join two trees by inserting an edge in time $O(d \log_d n)$, using a constant number of splits and joins on the corresponding B-trees. Also we can test two vertices of the forest to determine whether they are in the same tree in time $O(\log_d n)$. See for example [1, 11] for operations on B-trees. If $d = n^\alpha$, for α a positive constant, then the join and split operations take time $O(d)$ and the test operation takes time $O(1)$. We refer to the B-trees used to store ET-sequences as ET-trees.

This data structure allows us to keep information about a vertex so that the cumulative information about all vertices in a tree may be maintained. For example, we may keep the number of nontree edges incident to a vertex at one designated occurrence of the vertex. Then each internal node of the ET-tree stores the sum of the numbers of nontree edges kept with designated occurrences in its subtree. In a degree d ET-tree, each split or join operation or each change to the number associated with an occurrence requires the adjustment of $O(\log_d n)$ internal nodes with each adjustment taking $O(d)$ timesteps.

We maintain the following data structures.

- Each edge is labelled by its level and a bit which indicates if it is a tree edge.
- Let $k' = \max\{{m'}_{in}^{1/3} \log n, n^\epsilon\}$, for any constant $0 < \epsilon \leq 1/3$. Each tree in F is represented as an ET-sequence which is stored in a degree k' B-tree.
- Let $c = {m'}_{in}^{1/3}/\log n$. We map each level i to the j which is the largest multiple of c no greater than i by the function $f(i) = c\lfloor i/c \rfloor$.
  For each level j such that $c|j$ ("c divides j"):

- we represent each tree in F_j as an ET-sequence which is stored in a binary B-tree;
- for each vertex v, we create a list $L_j(v)$ which contains:
  (i) all nontree edges incident to v which are in any level $i \in f^{-1}(j)$ and;
  (ii) all tree edges incident to v which are in any level $i > j, i \in f^{-1}(j)$.
- We mark each designated occurrence of a vertex v whose list $L_j(v)$ is nonempty. Each internal node of the ET-tree is marked if its subtree contains a marked occurrence.

2.2 The Search routine

Search$(T_i(x))$ returns all nontree edges in level i incident to $T_i(x)$. It begins by searching $T_{f(i)}(x)$ which is a subtree of $T_i(x)$. It proceeds by examining all edges in $L_{f(i)}(v)$ for all vertices v in the tree being searched. Nontree edges in level i are picked out and tree edges in levels $i', f(i) < i' \leq i$ are followed to other trees of $F_{f(i)}$ which are then searched in turn. Note that all such tree edges lead to other trees of $F_{f(i)}$ which are subtrees of $T_i(x)$. A *phase* of the algorithm consists of the examination of one edge e in a list L.

Search$(T_i(u))$

1. $S' \leftarrow \emptyset$;
2. $treelist \leftarrow T_{f(i)}(u)$;
3. Repeat until $treelist$ is empty:
   - Remove an ET-tree from the $treelist$.
   - For each marked vertex u in the ET-tree and for each edge e in each $L_{f(i)}(u)$:
     - If $\{u, v\}$ is a nontree edge on level i, add it to the set of edges to return.
     - Else if $\{u, v\}$ is a tree edge on level l such that $l \leq i$, then add $T_{f(i)}(v)$ to $treelist$.

2.3 Analysis

Initialization: We compute the minimum spanning forest F, create the ET-trees for F_j, for each j such that $c|j$, and partition the nontree edges by weight. Recall that m'_{in} is the number of nontree edges in the initial graph. Let t be the number of edges in the initial minimum spanning forest. The creation of all the lists L takes time proportional to the number of nontree edges m'_{in}. The building of ET-trees for F and all F_j such that $c|j$ and the marking of internal nodes takes time proportional to the size of each forest or $O(((m'_{in}/k)/c)t + m'_{in}) = O(m'^{1/3}_{in} t + m'_{in})$.

Deletions of nontree edges: Deleting a nontree edge on any level may require resetting the bit of an occurrence of a vertex in some ET-tree, which may require resetting bits on all internal nodes on the path to the root in $O(\log n)$ time.

Deletions and insertions of tree edges: Deleting a tree edge takes $O(k')$ time to delete it from the ET-tree of F and $O(\log n)$ time to delete it from the ET-tree of each F_j such that $c|j$, for a total of $O(k' + ((m'_{in}/k)/c)\log n)$ time per edge. Inserting a replacement edge takes the same time.

Finding a replacement edge: We first analyze the cost of **Search**. Let the *weight* $w(T)$ *of a tree* T *of some* F_i be $\sum |L_{f(i)}(v)|$ summed over all vertices v in T. It costs $O(\log n)$ to move down the path from the root to a leaf in an ET-tree to find a marked occurrence of a vertex, or to move up a tree from an occurrence to the root. Thus, the cost of **Search**$(T_i(x))$ is $O(\log n)$ times the number of edges examined, or $O(w(T_i(x))\log n)$, if **Search** is carried out until it ends, and $O(k)$ if it is run for $k/\log n$ phases.

In **Replace**(u, v, i), if $w(T_i(u)) \leq w(T_i(v))$, then we refer to $T_i(u)$ as the *smaller component* T_1; otherwise T_1 is $T_i(v)$. The cost of a call to **Replace**(u, v, i) is the cost of the **Search** plus the cost of testing each edge in S. The number of edges in S is $O(\min\{k, w(T_1)\})$. We may use the k'-degree ET-tree representation for F to test each edge at cost $O(1)$. Thus the cost of a call to **Replace** is $O(\min\{k, w(T_1)\log n\})$.

To pay for these costs: We charge the cost of a call to **Replace**(u, v, i) to level i if no replacement edge is found on that level. In that case, a tree of F_i which was split by the deletion remains split. Otherwise, we charge the cost to the deletion. In addition, we charge the cost of modifying F to the deletion so the total cost charged to the deletion is $O(\min\{k, w(T_1)\log n\} + ((m'_{in}/k)/c)\log n + k') = O(((m'_{in}/k)/c)\log n + k')$.

Claim 1 *$O(\sum w(T_1))$ summed over all smaller components T_1 which split from a tree T on any given level during all* **Replace** *operations is $O(w(T)\log n)$.*

The proof of the claim is not hard and follows [5]. The details are omitted here.

There are at most k edges per level (except for level 0, which has at most k nontree edges). Each $L_j(v)$ consists of edges from c levels. Since level 0 tree edges do not belong to any list $L_j(v)$, the maximum weight of a tree $w(T)$ is ck. Thus the total cost charged to a level is $O(ck\log^2 n)$. Summing over all levels we have $O((m'_{in}/k)(ck\log^2 n) = O(m'_{in}c\log^2 n)$, or an amortized cost per deletion of $O(c\log^2 n) = O(m'^{1/3}_{in}\log n)$, if $\Omega(m'_{in})$ edges are deleted.

The cost charged to each deletion is $O((m'_{in}/ck)(\log n) + k')$. For $k' = max\{m'^{1/3}_{in}\log n, n^{\epsilon}\}$ and $c = m'^{1/3}_{in}/\log n$, this is $O(m'^{1/3}_{in}\log n + n^{\epsilon})$.

To summarize the cost of initialization when amortized over $\Omega(m_{in})$ operations is $O(m'^{1/3}_{in})$ and the cost per deletion of an edge and finding replacement edges, when amortized over $\Omega(m'_{in})$) operations is $O(m'^{1/3}_{in}\log n + n^{\epsilon})$. Thus for a sequence of $\Omega(m_{in})$ operations, the amortized time per update is $O(m'^{1/3}_{in}\log n + n^{\epsilon})$.

Finally, we note that the query of the form "Are nodes i and j connected?" may be answered using the ET-tree data structure for F in $O(1)$ time.

3 Allowing insertions

As in the previous section, we assume all edge weights are unique. We refer to the current minimum spanning forest of G as the MST. Let m' be the number of nontree edges in the current graph.

Let $s = \lceil \lg m' \rceil$. Initially, we build and maintain s simultaneous deletions-only data structures $A_s, A_{s-1}, .., A_1$ and a set of edges B. We call this the *composite data structure*. We maintain the MST in a Sleator-Tarjan dynamic tree [13] and also in an ET-tree of degree $\max\{m'^{1/3} \log n, n^{\epsilon}\}$.

Below, we distinguish between the number of edges inserted into G and the number of edge insertions into B, as an edge of G may be inserted more than once into B even though it has not been deleted and reinserted into G. The minimum spanning forests of the deletions-only data structures are referred to as *local spanning forests*. A *local nontree edge* of an A_i is an edge which is not in A_i's local spanning forest or the MST. We will see that every nontree edge of G is a local nontree edge of some A_i or B, but may also be a tree edge in a local spanning forest of an $A_j, j \neq i$.

Initially A_s is the deletions-only data structure described in the previous section, with $F = MST$ and the set of local nontree edges being all nontree edges of G, and the parameter m'_{in} set to 2^s. The set B is empty and the remaining A_j, $1 \leq j < s$, are initialized ("built") as though the edges of the MST were the only edges in A_j, i.e., they contain no nontree edges. The set B is empty.

For $i = 1, \ldots, s$, let $m_i = 2^i$, $k_i = m_i^{1/3} \log m_i$, and $k'_i = \max\{m_i^{1/3} \log n, n^{\epsilon}\}$. When an edge is inserted into G, it is placed into B or into the MST.

Let x_i be the number of local nontree edges in $\cup_{j \leq i} A_j \cup B$. Each A_i is built (or rebuilt) when i is the smallest index such that $m_i \geq x_i$ and the number of edges in B has increased to $m'^{1/3}$. At that time, B is emptied and all local nontree edges $\cup_{j \leq i} A_j$ and edges in B are removed from $A_j, j < i$, and B and placed into A_i. Then A_i becomes the deletions-only data structure described in the previous section, which is initialized (or reinitialized) to contain the tree edges of the MST and the local nontree edges previously contained in $\cup_{j \leq i} A_j$ and the edges B. Thus, throughout the algorithm, B contains fewer than $m'^{1/3}$ edges, i.e., the most recent insertions into B, which have not yet been added to some A_j and for $j < \lfloor \lg m'^{1/3} \rfloor$ A_j never contains any nontree edges. These A_j are maintained in the event that they will be used later, if m' is reduced.

To insert edge e into G: Use the dynamic tree to find the maximum weight edge f on the path between e's endpoints in the MST. If e is lighter than f, remove f from the MST, and insert f into B. Else insert e into B.

To delete an edge e from G: (1) Delete e from all data structures in which it appears. (2) For each A_i which contained e in its local spanning forest, update the A_i by determining e's local replacement edge e' (if there is one). Insert e' into A_i's local forest and into B, if it is not already there. (3) If e was in the MST, then for each local replacement edge e' and each edge in B, use the ET-tree representation of the MST to determine which of those edges connect the two subtrees resulting from the deletion of e. Insert the lightest connecting edge into the MST.

3.1 Proof of correctness

Our algorithm maintains the following invariant:

Invariant: Every edge in the local forest of some A_i is (1) in the MST, or (2) is a local nontree edge in some $A_j, j \neq i$, or (3) is in B.

Lemma 2. *The invariant stated above holds throughout the execution of the algorithm.*

The proof of the lemma is straightforward and is omitted here.

The correctness of the algorithm follows easily from the invariant. We use the well-known fact that an edge is in the minimum spanning tree iff it is not the heaviest edge in any cycle ("red rule" [14]). We also note that every edge in the composite data structure is an edge in G.

Let e be an edge of the MST which is deleted. Let e' be the correct replacement edge. Consider the state of the composite data structures right before the deletion of e. By the invariant, since e' was not in the MST, it was a local nontree edge in some A_i or in B. If e' was in B it would be checked in Step 2 above.

If e' was a local nontree edge in A_i, then consider the subgraph G' of G whose edgeset consists of edges in A_i. Since e' is the correct replacement edge for e in the MST then after e's deletion, e' is not the heaviest edge in any cycle of G and therefore is not the heaviest edge of any cycle of G'. Hence, after e's deletion, e' becomes a local forest edge, i.e., e' is a local replacement edge for e in A_i. Recall that e' is the minimum weight edge which connects the two subtrees of the MST resulting from the deletion of e. Thus, e' is the lightest connecting edge from among the edges of B and the set of local replacement edges, and is chosen in Step 2 by the algorithm.

3.2 Implementation and analysis

At the start of the algorithm, A_i for $i \leq s$ are built. After that, A_i for $i \leq s$ may be "rebuilt". Depending of the value of m', A_{s+1}, A_{s+2},... may be built later. We first consider the (one-time) cost of building the A_i's, then the cost of their rebuilding, and finally the cost of maintaining the A_i between rebuilds.

Initialization of the A_i: Let m_{in} be the size of the initial graph (number of vertices plus edges), let m'_{in} be the initial number of nontree edges, and for each operation let m be the size of the current graph. Recall that the total cost of initialization for a deletions-only data structure with m_i nontree edges is $O(m_i^{1/3}n + m_i)$ and that we are given a sequence of $\Omega(m_{in})$ operations.

We will amortize the building of the first $\lceil \lg m_{in} \rceil$ A_i's to the sequence of $\Omega(m_{in})$ operations, even though only s A_i's are built initially. If more than

$\lceil \lg m_{in} \rceil$ A_i's are necessary at some point, we know that at this point $m > m_{in}$ and at least m_{in} insertions happened. Let $A_{s_{max}}$ be the largest deletions-only data structure built during the execution of the algorithm where $2^{s_max} > m_{in}$. Then there was a sequence of $\Omega(2^{s_max})$ operations during which m was $\Omega((2^{s_max})$ and we can amortize the initialization cost over these operations.

The total cost of initializing the s_{max} $A_i's$ is $O(\sum_{i=1}^{s_{max}} 2^{i/3}n+2^i) = O(2^{s_{max}/3}n+m_{in})$. The average cost over a sequence of $\Omega(2^{s_{max}})$ operations is thus $O(2^{2s_{max}/3}n)$, which is $O(m^{1/3})$ per operation.

Rebuilding: We create ET-trees for the new A_i by modifying the ET-trees for the previous A_i. For each i, we keep a list of all changes made to each ET-tree of A_i since the last rebuild, and a list of all changes made to the MST. We use this list to first restore all the ET-trees for A_i to their previous state when A_i was last built or rebuilt, MST_{old}, by undoing each change, edge by edge. We then transform each MST_{old} to MST, edge by edge.

The cost of restoring the ET-trees of A_i to MST_{old} is charged to operations on the deletions-only data structure A_i which caused the initial change. This results in only a doubling of cost per operation, as the cost for inserting a tree edge into an ET-tree is the same as for deleting a tree edge.

The cost of transforming MST_{old} to MST is charged to the update operation that causes the change in MST (each update causes $O(1)$ changes in MST) as follows: For each A_i, there are $m_i^{1/3}$ forests of ET-trees represented by binary B-trees and one forest (the ET-tree for the local spanning forest F) represented by a degree-k'_i B-tree. Thus, a single tree edge insertion or deletion costs $O(m_i^{1/3} \log n + k'_i)$ for A_i. Note that for each i one change to the MST contributes to the cost of only one rebuild of A_i. The total cost per change over all levels is $O(\sum_{i=1}^{s} m_i^{1/3} \log n + k'_i + \log n) = O(m'^{1/3} \log n + n^\epsilon \log n)$.

Also, when A_i is rebuilt, all local nontree edges from $A_j, j \leq i$ and B are moved from A_j and inserted into A_i. That is, the edges are sorted by weight, assigned to levels in A_i, and put in the appropriate list L. The bits on the internal nodes of ET-trees for $A_j, j \leq i$ are set appropriately. Since each local nontree edge is stored in only one ET-tree on a level, the cost of moving a single local nontree edge is $O(\log n)$. Thus, the total cost is $O(m_i \log n)$. Since A_{i-1} is not rebuilt, $x_{i-1} > m_{i-1} = m_i/2$. We amortize this cost by charging $O(\log n)$ to each edge in $\cup_{j<i} A_j \cup B$, i.e. each edge that is newly added to A_i. We show below (type (3) charges) how to amortize these costs over the update operations.

Maintaining the deletions-only data structures: After a rebuild in A_i there are at most m_i nontree edges in A_i. In Section 2, we have two types of charges: (1) the cost charged to each deletion in a deletions-only data structure which is $O(m_i^{1/3} \log n + n^\epsilon)$ and (2) the cost charged to all the levels which is $O(m_i^{1/3} \log n)$ per nontree edge. Additionally, the rebuilding of an A_i above charged $O(\log n)$ to each nontree edge in A_i. We call these costs type (3) charges.

Type (1) charges: When there is a deletion in G in the fully dynamic data structure, an edge (or one of its copies) may be deleted from each of $A_i, i =$

$1, \ldots s$. We may charge that deletion in G with the (1) charges for all levels for a total cost of $O(\sum_i m_i^{1/3} \log n + n^\epsilon) = O(m'^{1/3} \log n + n^\epsilon \log n)$.

Type (2) and (3) charges: As a special case, the charges incurred by the first deletions-only data structure containing nontree edges A_s must be amortized over the initial sequence of $\Omega(m_{in})$ operations which follow its initialization. Since each A_i, $i < s$ is initialized to contain no nontree edges, there are no type (2) and (3) for these data structures until they are rebuilt.

Note that the A_i, $i > s$ contain nontree edges when initialized. The type(2) and type(3) charges for their building and rebuilding and the rebuilding of the other A_i $i \leq s$ are amortized over the insertions which occurred previous to its building or rebuilding, as analyzed below.

Suppose A_i is rebuilt. Since A_{i-1} was not rebuilt, $x_{i-1} > m_{i-1} = m_i/2$ at the time of the rebuilding of A_i. Thus, $\Omega(m_i)$ insertions into B occurred since the previous rebuild of A_i, and $\Omega(m_i)$ of these occurred when the graph had $\Omega(m_i)$ nontree edges. Thus, we may charge each insertion into B with $O(\sum_i^{s'} m_i^{1/3} \log n + n^\epsilon) = O(m_{s'}^{1/3} \log n + n^\epsilon \log n)$ where $s' = 2^{\lceil \lg m' \rceil}$ where m' is the number of nontree edges in G when the insertion occurred.

To amortize costs over insertions into G, rather than B, we use the following simple but crucial observation: When an edge is inserted into B *that edge may contribute to the type (2) and (3) costs for A_i (when it belongs to A_i) iff it increases x_i*. Note that $m_i \geq x_i \geq x_{i-1} > m_{i-1} = m_i/2$. We charge $O(m_i^{1/3} \log n)$ to each local nontree edge inserted into A_{i+1} to pay for the type (2) and (3) charges while the edges are in A_i.

We examine the types of insertions into B to see how they affect x_i: (a) when an edge is first inserted into B, i.e., when the edge is inserted into G; (b) when an edge is replaced in the MST; (c) when an edge is deleted in G and it is replaced in up to s local spanning forests. The first two cases result in a single insertion into B. The third case may cause up to s' insertions. However, the s insertions do not affect all A_i the same. Each insertion in this case results from a local nontree edge e becoming a local forest edge. Hence if this occurs in some $A_j, j \leq i$, the increase of x_i resulting from the insertion of a copy of e into B is offset by the decrease of x_i caused by the change in status of e from a local nontree edge to a local tree edge. Thus $x_{s'}$ is unchanged by a case-(c) insertion into B, $x_{s'-1}$ is changed by at most 1, and in general, x_i is changed by at most $s'-i$. The type (2) and (3) cost per deletion is $O(\sum_i (s'-i) m_i^{1/3} \log n) = O(\sum i (m_{s'}/2^i)^{1/3} \log n) = O(m'^{1/3} \log n)$.

Thus the deletion cost per update operation is $O(m'^{1/3} \log n + n^\epsilon \log n)$.

Insertion cost: Testing a newly inserted edge to see if it should be added to the MST using the Sleator-Tarjan dynamic trees is an $O(\log n)$ cost operaton. Adding an edge to B can be done in constant time, as B is an unsorted list.

Summary: For rebuilding and maintaining the deletions-only data structures, the algorithm achieves an amortized cost of $O(m'^{1/3} \log n + n^\epsilon)$ per update, where m' is the number of nontree edges in the graph, for processing a sequence of $\Omega(m_{in})$

operations, where m_{in} is the initial size of the graph (vertices plus edges). For the initializations of the deletions-only data structures, the amortized cost per update is $O(m^{1/3} \log n)$, where m is the size of the graph at the time of the update, for a sequence of $\Omega(m_{in})$ operations.

Note: For unweighted graphs, a simpler fully dynamic data structure can be constructed which uses only one deletions-only data structure and adds levels as needed. The details are omitted here.

References

1. T. Corman, C. Leiserson, and Rivest. *Introduction to Algorithms.* MIT Press (1989), p. 381-399.
2. D. Eppstein, "Dynamic Euclidean minimum spanning trees and extrema of binary functions", Disc. Comp. Geom. 13 (1995), 111-122.
3. D. Eppstein, Z. Galil, G. F. Italiano, "Improved Sparsification", Tech. Report 93-20, Department of Information and Computer Science, University of California, Irvine, CA 92717.
4. D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig, "Sparsification - A Technique for Speeding up Dynamic Graph Algorithms" *Proc. 33rd Symp. on Foundations of Computer Science*, 1992, 60–69.
5. S. Even and Y. Shiloach, "An On-Line Edge-Deletion Problem", *J. ACM* 28 (1981), 1–4.
6. T. Feder and M. Mihail, "Balanced matroids", *Proc. 24th ACm Symp. on Theory of Computing*, 1992, 26–38.
7. G. N. Frederickson, "Data Structures for On-line Updating of Minimum Spanning Trees", *SIAM J. Comput.*, 14 (1985), 781–798.
8. G. N. Frederickson and M. A. Srinivas, "Algorithms and data structures for an expanded family of matroid intersection problems", *SIAM J. Comput.* 18 (1989), 112-138.
9. M. R. Henzinger and V. King. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *Proc. 27th ACM Symp. on Theory of Computing*, 1995, 519–527.
10. M. R. Henzinger and M. Thorup. Improved Sampling with Applications to Dynamic Graph Algorithms. To appear in *Proc. 23rd International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 1099, Springer-Verlag, 1996.
11. K. Mehlhorn. "Data Structures and Algorithms 1: Sorting and Searching", Springer-Verlag, 1984.
12. H. Nagamochi and T. Ibaraki, "Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph", *Algorithmica* 7, 1992, 583–596.
13. D. D. Sleator, R. E. Tarjan, "A data structure for dynamic trees" *J. Comput. System Sci.* 24, 1983, 362–381.
14. R. E. Tarjan, *Data Structures and Network Flow*, SIAM (1983) p. 71.