SRC Technical Note
1997 - 019

September 5, 1997

Maintaining Minimum Spanning Trees in Dynamic Graphs

Monika Rauch Henzinger and Valerie King

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Valerie King is at the University of Victoria, Victoria, BC. Her electronic mail addresses is: 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 it connects two previously unconnected trees of F or if it reduces the cost of F. If the latter, 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 $k = m'^{1/3}_{in} \log n$. 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. (We omit floors and ceilings to simplify notation; either may be used without affecting the asympotic analysis.)

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 undesirable 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 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 $k' = \max\{m'^{1/3}_{in} \log n, n^\epsilon\}$, 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 $c = m'^{1/3}_{in} / \log n$. 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($T_i(u)$) returns all nontree edges in level i incident to $T_i(u)$. It begins by searching $T_{f(i)}(u)$ which is a subtree of $T_i(u)$. 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 $T_i(u)$; and all subtrees of $T_i(u)$ will be found by this procedure. 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 x in the ET-tree and for each edge $\{x, y\}$ in each $L_{f(i)}(x)$:
     - If $\{x, y\}$ is a nontree edge on level i, add it to the set of edges to return.
     - Else if $\{x, y\}$ is a tree edge on level l such that $l \leq i$, then add $T_{f(i)}(y)$ 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: If a replacement edge is found on level i then we charge the cost of **Replace**(u, v, i) to the deletion. In addition, we charge the cost of modifying F to the deletion so the total cost charged to the deletion is $O(\min\{k, w(T_1)\log n\} + ((m'_{in}/k)/c)\log n + k') = O(((m'_{in}/k)/c)\log n + k')$.

If no replacement edge is found on level i then a tree of F_i which was split by the deletion remains split. We use the following:

Claim 2.1 *$O(\sum w(T_1))$ 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 follows [5]. The first time a smaller component T_1 of a tree T is searched, it can have weight no greater than $w(T)/2$. Between two successive times that $|L_{f(i)}(v)|$ contributes to the weight of a smaller component T_1 and that component splits off, the weight of a smaller component T_1 containing v is no more than half its weight the previous time. Hence $|L_{f(i)}(v)|$ contributes to the weight of any T_1 no more than $log_2 w(T) = O(\log n)$ times. I.e., $O(\sum w(T_1)) = O(\sum_{v\in T}|L_{f(i)}(v)|)\log n) = O(w(T)\log n)$.

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 From deletions-only to fully dynamic

In this section, we show a general technique to develop a fully dynamic data structure using several deletions-only data structures with an added operation. (We call these "extended" deletions-only data structures.) As before, we assume the edge weights are distinct.

First, we define the following operation on a deletions-only data structure A.

batch_add(G, E', F'): Given a graph $G = (V, E)$ with minimum spanning forest F, insert all edges of E' into G, if they are not already there. The resulting spanning forest F' is given.

We refer to the period of time which occurs between two consecutive calls to *batch_add* on a graph G, or between the start of the algorithm and the first *batch_add* on G as a *period of* G. Alternatively, a period may be terminated prematurely (see below).

We prove the following theorem:

Theorem 3.1 *Suppose for any value of* n *and* m'_{in}*, there is an extended deletions-only data structure for any dynamic graph* $G = (V, E)$ *with* $|V| = n$ *and the number of nontree edges in the edgeset* E *is initially* m'_{in}*, such that* $(n+m'_{in})f^0(m'_{in}, n)$ *is the worst case time needed to initialize* A*, and* $(y + m'_{in})f(m'_{in}, n)$ *is an upper bound on the time to process* y *deletions.*

Suppose we can process a $batch\_add(H, E', F')$*, following any period in which* y *edges were deleted from* G*, in time* $O((y + m'_{in} + |F' \setminus F|)f^B(m'_{in}, n))$*, where* m'_{in} *is an upper bound on the total number of nontree edges in* G *after the batch_add.*

We also assume that $f^0,\ f,\ f^B$ *are monotone nondecreasing functions.*

There is a fully dynamic minimum spanning tree data structure that runs in amortized cost per edge deletion or insertion of $O(\log^2 n + \sum_{i=0}^{s}(s-i+1)[f^0(m_i, n) + f(m_i, n) + f^B(m_i, n)]$ *where* $s \leq 3 + \lg m$ *and* m *is the size (vertices plus edges) of the dynamic graph at the time of the update. Here, costs are amortized over a sequence of* m_{in} *update operations, where* m_{in} *is the size of the initial graph.*

In Section 3.4, we show that

Corollary 3.2 *A minimum spanning forest can be maintained in a fully dynamic graph with amortized cost per update of* $O(m^{1/3}\log n)$*, where* m *is the size of the graph at the time of the update, for a sequence of* $\Omega(m_{in})$ *operations.*

We prove the theorem, by constructing a fully dynamic data structure from extended deletions only data structures.

Definitions: We refer to the current minimum spanning forest of G as the (global) MST. Let m' be the number of nontree edges in the current graph, m'_{in} denote the number of nontree edges in the initial graph, and m denote the current size (vertices and edges) of G.

During the course of the algorithm, we simultaneously maintain up to $s \leq \max\{\lg n, \lg(4m')\}$ extended deletions-only data structures $A_0, A_1, ..., A_s$, where each A_i is an extended deletions-only minimum spanning tree data structure for a subgraph $G^i + (V, E^i)$ of the global graph $G = (V, E)$. 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 2.

For $i = 0, \ldots, \lceil 2 \lg n \rceil$, let $m_i = 2^i$. The minimum spanning forest of G^i as maintained by A_i is referred to as *local spanning forest* and denoted F^i. A *local nontree edge* of A_i is an edge of G^i which is not in A_i's local spanning forest or the MST. Let x_i be the number of local nontree edges in $\cup_{j \leq i} A_j$.

When m' falls below $2^s/4$ and $s > \lg n$, s is *reset* and the composite data structure is reinitialized. Between two consecutive resets, we define the the period of time which occurs between two consecutive calls to *batch_add* on a graph G, or between the initialization or reinitialization of the composite data structure and the first *batch_add* on G as a *period of* G. A reset terminates all periods.

The *size* of a graph refers to the number of vertices plus edges. Note that the size of a graph is always $\Theta(s)$.

We maintain the following invariants:

Invariants: (1) Every edge in the local forest of some A_i is (a) in the MST, or (b) is a local nontree edge in some A_j, $j \neq i$.
(2) $E = (\cup E^i) \cup MST$.

We now describe the algorithm.

To initialize: Let the initial value of $s = \lceil \lg m'_{in} \rceil$. We initialize A_s as an extended deletions-only data structure for $G^s = G$ with $F^s = MST$ and the set of local nontree edges being all nontree edges of G.

To perform an insertion operation, **insert**(u, v) is called, where (u, v) is an edge to be inserted into G.

insert(e):

1. Use the dynamic tree to determine if e should be added to the MST:
   Find the maximum weight edge f on the path between e's endpoints in the MST.

2. If there is no path between e's endpoints or if there is a path and e is lighter than f, remove f from the MST, call **insert_nontree**(f), and add e to the MST.

3. Else call **insert_nontree**(e).

The following subroutine inserts a nontree edge e into the composite data structure:

insert_nontree(e). Let i be the smallest index such that $m_i \geq x_i$, the number of local nontree edges in $\cup_{j \leq i} A_j$.

Let E' be the set of local nontree edges in $\cup_{j<i} E^j \cup \{e\}$.

1. Delete the edges of E' from A_j, $j < i$.
   Set $x_j = 0$.

2. If A_i is not initialized, initialize A_i on the empty graph G^i consisting of n nodes and no edges.

3. Call $batch\_add(G^i, E' \cup MST, MST)$.
   Adjust x_i accordingly.

After the procedure, the local nontree edges of G^i are the nontree edges previously contained in $\cup_{j \leq i} A_j$. Its local forest $F^i = MST$. Note that at the beginning of a period of G^i, $x_j = 0$ for $j < i$.

To delete an edge e from G:

delete(e):

1. Delete e from all data structures in which it appears, including all G^i, and update corresponding A_i accordingly. Thus for each local spanning forest F^i which contained e, the local replacement edge e' is determined, if there is one.

2. If e was in the MST, use the ET-tree representation of the MST to determine which of those local replacement edges reconnect the two subtrees of the MST which result from the deletion of e. Insert the lightest connecting edge into the MST.

3. All other local replacement edges are reinserted into the composite data structure using the procedure $insert\_nontree$.

4. If $x_s \leq m_{s-2}$ and $x_s > n$, reinitialize the composite data structure. That is, set $s = \lceil \lg x_s \rceil$; initialize A_s as an extended deletions-only data structure for $G^s = G$ with $F^s = MST$ and the set of local nontree edges being all nontree edges of G.

3.1 Proof of correctness

It is easy to see that the invariants are maintained, by induction on the number of operations. Initially, the invariants hold since $G^{s'} = G$. Invariant (2) is preserved after each insertion, since each edge when added to G is either added to the MST or some G^i. Each edge when deleted from G is deleted from all data structures in which it appears. Invariant (1) holds for A_i when A_i is initialized or a *batch_add* is executed since the local forest $F_i = MST$. The local forest of A_i changes only when an edge is deleted and is replaced by some edge e. Edge e is then either put into the MST or reinserted into the composite data structure. In that case, it is added to some A_j by a *batch_add* operation. If e is not in the MST, then e becomes a local nontree edge of A_j. In either case, invariant (1) is preserved.

The correctness of the algorithm follows easily from the invariants. 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.

Suppose e' is a local nontree edge 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^i. Hence, after e's deletion, e' becomes a local forest edge, i.e., e' is a local replacement edge for e in G^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 the set of local replacement edges, and is chosen in Step 2 of the **delete** algorithm.

3.2 Analysis

We first prove the following claims:

Claim 3.3 *During any full period of G^i, there were at least $m_{i-1}/(s - i + 1)$ updates to G.*

Proof: For $i > 0$, immediately before *batch_add* is executed on G^i, $x_{i-1} > m_{i-1}$. Immediately afterwards, $x_{i-1} = 0$.

We examine the types of insertions into the composite data structure to see how they affect x_i: (a) when a nontree edge is inserted into G (b) when an edge is replaced in the MST after an insertion ; (c) when an edge is deleted in G and it is replaced in up to s local spanning forests. The first two cases cause x_i to increase by no more than one. 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 the composite data structure 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 the composite data structure, x_{s-1} is changed by at most 1, and in general, x_i is changed by at most $s - i$.

Hence, at least $m_{i-1}/(s - i + 1)$ insertions or deletions occurred during any full period of G^i. This concludes the proof of the claim.

We are now ready to analyze the costs of the algorithm.

Initialization: Since each A_i is initialized once, the cost for initialization during the algorithm is $(n + m_i) f^0(m_i, n)$. Note that A_i is initialized only if the number of nontree edges exceed m_{i-1}.

We amortize the initialization costs of the first data structure A_s and all A_i for $i < \lg n$ by requiring there to be $\Omega(m_{in})$ operations, where m_{in} is the size (vertices and edges) of the initial graph. We note that for at least half these operations, the current size of the graph $m \geq m_{in}/2$.

We amortize the cost of initializing A_i, $i \geq \lg n$ over the operations of the preceding period when at least $m_{i-1}/(s - i + 1)$ operations occurred.

The cost of *reinitialization* of the composite data structure may be charged to the $m_s/2$ deletions which must have occurred since the previous reset. Note that a reset only occurs when $m_s > n$, so that the initialization cost of $(n + m_{s-2}) f^0(m_{s-2}, n)$ results in a charge of $f^0(m_{s-2}, n)$ per operation.

Execution of batch_add:

By assumption, $(y + m_i + |MST \setminus F^i|) f^B(m_i, n)$ is an upper bound on the time to perform $batch\_add(G^i, E' \cup MST \setminus F^i, MST)$, where y is the number of deletions performed on G^i in the preceding period.

We can charge the cost of $y f^B(m_i, n)$ to the y deletions for a cost of $f^B(m_i, n)$ each.

To charge the $m_i f^B(m_i, n)$: By the claim, *batch_add* is called on G^i after at least $m_{i-1}/(s - i + 1)$ insertions and deletions occurred in the preceding period. Charging the $m_i f^B(m_i, n)$ to those updates gives a cost per update of $(s - i + 1) f^B(m_i, n)$.

To charge the $|MST \setminus F^i| f^B(m_i, n)$, we note that at the start of the period, $F^i = MST$. and for each i, each insertion or deletion in G can cause at most one edge to be added to and/or one edge to be deleted from F^i. Thus we can charge the $|MST \setminus F^i| f^B(m_i, n)$ to the operations in the preceding period, for a cost of $O(f^B(m_i, n))$ each.

Performing deletions during a period of G^i:.

The cost of maintaining A_i during a period containing y deletions of edges in G^i is by assumption, bounded above by $(y + m_i) f(m_i, n)$. These costs may be charged in the same way as the costs for *batch_add* were charged, to the operations of the preceding period.

In the unique case of the initial A_s, where $s = \lceil \lg m'_{in} \rceil$ when there was no preceding period, costs are amortized over the initial sequence of $\Omega(m_{in})$ deletions and insertions, as in the analysis of the initialization costs.

After a reset of s, the cost of performing deletions in A_s after A_s is reinitialized is charged to the deletions which resulted in the reset, as in the analysis of the reinitialization costs.

The cost of $O(\log^2 n)$ for Step (2) and to update the MST is charged to the delete operation.

Summary: For each i, the cost per operation is therefore $O((s-i+1)[f^0(m_i, n) + f(m_i, n) + f^B(m_i, n)]$.

Except for the initialization and reinitialization of A_s, we have charged operations of the preceding period for all costs incurred in the following period. Since the preceding periods occur in between resets of the value of s, we know that for the indices of the A_i, $i \leq s \leq max\{\lg 4m', \lg n\}$. Hence $s \leq 2 + \lg m$, m being the size of the graph at the time of the operation.

For the initialization and reinitialization of A_s, we charge operations which occurred when s may have been smaller by 1. Hence $s \leq 3 + \lg m$, m the size of the graph at the time of the operation.

Each operation requires a constant number of updates in the dynamic tree data structure and the binary ET-tree data structure storing the MST. This takes time $O(\log n)$.

Summing over i, we have of $O(\log^2 n + \sum_{i=0}^{s}(s-i+1)[f^0(m_i, n) + f(m_i, n) + f^B(m_i, n)]$, where m is the current size of G at the time of the operation, when amortized over a sequence of m_{in} update operations and m_{in} is the size of the initial graph.

3.3 Implementing *batch_add*

In this section, we show how a deletions-only data structure A in section 2 for a graph G which initially had m'_{in} nontree edges can be extended so that the operation *batch_add* which occurs after a sequence σ of y edge deletions can be implemented in time $O((y + m'_{in} + |F' \setminus F|)(m'_{in}{}^{1/3} \log n)$.

We begin by restoring the ET-trees of A to MST_{old}, the minimum spanning forest of G before the start of the sequence σ of deletions. The cost of joining two ET-trees is asympotically the same as splitting them; thus the calculations of section 2 apply. For each deletion, the cost of restoration is $O({m'}_{in}^{1/3}) \log n + n^{\epsilon})$.

We next transform MST_{old} to MST, by again modifying the ET-trees. We remove every edge in $MST_{old} \setminus MST$ and insert every edge in $MST \setminus MST_{old}$, for a cost of $O({m'}_{in}^{1/3} \log n + n^{\epsilon})$.

To determine the transformations required, we keep a list of sorted changes which occurred since the last $batch\_add$.

We remove all nontree edges which are stored in A and sort the nontree edges of $E \cup E'$, assign them to levels, and store them with the appropriate list L. The cost per edge of removing, sorting and then storing is $O(\log n)$ per edge for the unique (binary) ET-tree in which the edge is stored.

Let $f'(m_i, n) = m_i^{1/3} \log n + n^{\epsilon}$. We have shown an extended deletions-only data structure such that $O((n + m'_{in}) f'(m_{in}, n))$ is an upper bound on the worst case time needed to initialize A, and $O((y + m'_{in}) f'(m_{in}, n))$ is an upper bound on the time to process y deletions.

And, we can process a $batch\_add(G, E', F')$, following any period in which y edges were deleted from G, in time $O((y + m'_{in} + |F' \setminus F|) f'(m'_{in}, n))$, where m'_{in} is an upper bound on the total number of nontree edges in G after the $batch\_add$.

3.4 Proof of corollary

Substituting f' for f^B, f, and f^0, we conclude that there is a fully dynamic minimum spanning tree data structure that runs in amortized cost per edge deletion or insertion of $O(\log^2 n + \sum_{i=0}^{s} (s - i + 1) f'(m_i, n)$, where $s \leq 3 + \lg m$.

Substituting for f' and m_i, we have $O(\log^2 n + \sum_{i=0}^{s} (i + 1)(2^{s-i})^{1/3} \log n + n^{\epsilon}) = O((2^s)^{1/3} \log n + n^{\epsilon} \log n) = O(m^{1/3} \log n + n^{\epsilon'})$ for ϵ' any constant.

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.