Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation

Monika Rauch Henzinger *
Valerie King †

Abstract

This paper solves a longstanding open problem in fully dynamic algorithms: We present the first fully dynamic algorithms that maintain connectivity, bipartiteness, and approximate minimum spanning trees in polylogarithmic time per edge insertion or deletion. The algorithms are designed using a new dynamic technique which combines a novel graph decomposition with randomization. They are Las-Vegas type randomized algorithms which use simple data structures and have a small constant factor.

Let n denote the number of nodes in the graph. For a sequence of $\Omega(m_0)$ operations, where m_0 is the number of edges in the initial graph, the expected time for p updates is $O(p \log^3 n)$[1] for connectivity and bipartiteness. The worst-case time for one query is $O(\log n / \log\log n)$. For the k-edge witness problem ("Does the removal of k given edges disconnect the graph?") the expected time for p updates is $O(p \log^3 n)$ and expected time for q queries is $O(qk \log^3 n)$. Given a graph with k different weights, the minimum spanning tree can be maintained during a sequence of p updates in expected time $O(pk \log^3 n)$. This implies an algorithm to maintain a $1+\epsilon$-approximation of the minimum spanning tree in expected time $O((p \log^3 n \log U)/\epsilon)$ for p updates, where the weights of the edges are between 1 and U.

Finally, we use the data structures to create simple, deterministic algorithms that maintain minimum spanning trees and connectivity fully dynamically. The algorithms take time $O(\sqrt{n} \log n)$ per update.

1 Introduction

In many areas of computer science, graph algorithms play an important role: Problems modeled by graphs are solved by computing a property of the graph. If the underlying problem instance changes incrementally, algorithms are needed that quickly compute the property in the modified graph. Algorithms that make use of previous solutions and, thus, solve the problem faster than recomputation from scratch are called *fully dynamic* graph algorithms. To be precise, a fully dynamic graph algorithm is a data structure that supports the following three operations: (1) insert an edge e, (2) delete an edge e, and (3) test if the graph fulfills a certain property, e. g. are two given vertices connected.

*Department of Computer Science, Cornell University, Ithaca, NY. Email: mhr@cs.cornell.edu. Author's Maiden Name: Monika H. Rauch. This research was supported by an NSF CAREER Award.

†Department of Computer Science, University of Victoria, Victoria, BC. Email: val@csr.uvic.ca. This research was supported by an NSERC Grant.

[1]Throughout the paper the logarithms are base 2.

Previous Work. In recent years a lot of work has been done in fully dynamic algorithms (see [1, 3, 4, 6, 7, 8, 10, 11, 13, 16, 17, 19] for connectivity-related work in undirected graphs). There is also a large body of work for restricted classes of graphs and for insertions-only algorithms. Currently the best time bounds for fully dynamic algorithms in undirected n-node graphs are: $O(\sqrt{n})$ per update for a minimum spanning forest [3]; $O(\sqrt{n})$ per update and $O(1)$ per query for connectivity [3];

$O(\sqrt{n}\log n)$ per update and $O(\log^2 n)$ per query for cycle-equivalence ("Does the removal of the given 2 edges disconnect the graph?") [11]; $O(\sqrt{n})$ per update and $O(1)$ per query for bipartiteness ("Is the graph bipartite?") [3].

There is a lower bound in the cell probe model of $\Omega(\log n/\log\log n)$ on the amortized time per operation for all these problems which applies to randomized algorithms [9, 11]. In [1] it is shown that the average update time of (a variant of) the above connectivity and bipartiteness algorithms is $O(n/\sqrt{m}+\log n)$ if the edges used in updates are chosen uniformly from a given edge set. Thus, for dense graphs their average performance nearly matches the lower bound.

In planar graphs fully dynamic algorithms for minimum spanning forest and connectivity are given in [5] that are close to the lower bound: they take time $O(\log^2 n)$ per deletion and $O(\log n)$ per insertions and query. However, the constant factor of these algorithms is quite large [5]. Thus, the following questions were posed as open questions in [4, 5]:

(1) Can the above properties be maintained dynamically in polylogarithmic time in (general) graphs?

(2) Is the constant factor in the fully dynamic algorithms small such that an efficient implementation is possible?

New Results. This paper gives a positive answer to both questions. It presents a new technique for designing fully dynamic algorithms with polylogarithmic time per operation and applies this technique to the fully dynamic connectivity, bipartiteness, $1+\epsilon$-approximate minimum spanning trees, and cycle-equivalence problem. The resulting algorithms are Las-Vegas type randomized algorithms which use simple data structures and have a small constant factor.

For a sequence of $\Omega(m_0)$ update operations, where m_0 is the number of edges in the initial graph the following amortized expected update times and worst-case query times are achieved:

1. connectivity in update time $O(\log^3 n)$ and query time $O(\log n/\log\log n)$;

2. bipartiteness in update time $O(\log^3 n)$ and query time $O(1)$;

3. minimum spanning tree of a graph with k different weights in update time $O(k\log^3 n)$;

As an immediate consequence of these results we achieve faster fully dynamic algorithms for the following problems:

1. An algorithm to maintain a $1+\epsilon$-approximation of the minimum spanning tree in expected time $O((p\log^3 n\log U)/\epsilon)$ for p updates, where the weights of the edges are between 1 and U.

2. An algorithm for the k-edge witness problem ("does the removal of the given k edges disconnect the graph?") in update time $O(\log^3 n)$ and amortized expected query time $O(k\log^3 n)$. Note that cycle-equivalence is equivalent to the 2-edge witness problem.

3. A fully dynamic algorithm for maintaining a maximal spanning forest decomposition of order k of a graph in time $O(k\log^3 n)$ per update by keeping k fully dynamic connectivity data structures.

A *maximal spanning forest decomposition of order* k is a decomposition of a graph into k edge-disjoint spanning forests $F_1, \ldots, F_k$ such that F_i is a maximal spanning forest of $G_i = G \setminus \cup_{j<i} F_j$. The maximal spanning forest decomposition is interesting since $\cup_i F_i$ is a graph with $O(kn)$ edges that has the same k-edge connected components as G [15].

Additionally we use the data structures to present simple deterministic algorithms that maintain minimum spanning trees and connectivity fully dynamically. The amortized time per update for the minimum spanning tree algorithm is $O(\sqrt{m}\log n)$, which can be improved to $O(\sqrt{n}\log n)$ using the sparsification technique [4]. The amortized time per update for the connectivity algorithm is $O(\sqrt{n}\log n)$. Even though these algorithms do not improve on the running time of the best known algorithms, they are interesting since they present a completely different approach than previous algorithms and use only simple data structures. Additionally, the connectivity algorithm is the first fully dynamic algorithm that does not use the sparsification technique and achieves a running time of less than $\Omega(\sqrt{m})$.

Main Idea. The new technique is a combination of a novel decomposition of the graph and randomization. The edges of the graph are partitioned into $O(\log n)$ levels such that edges in highly-connected parts of the graph (where cuts are dense) are on lower levels than those in loosely-connected parts (where cuts are sparse). For each level i, a spanning forest is maintained for the graph whose edges are in levels i and below. If a tree edge is deleted at level i, we sample edges on level i such that with high probability either (1) we find an edge reconnecting the two subtrees or (2) the cut defined by the deleted edge is too sparse for level i. In Case (1) we found a replacement edge fast, in Case (2) we copy all edges on the cut to level $i+1$ and recurse on level $i+1$.

To our knowledge the only previous use of randomization in fully dynamic algorithms is are (Monte-Carlo type) approximation algorithms for minimum cuts [12, 14].

This paper is structured as follows: Section 2 gives the fully dynamic connectivity algorithm, Section 3 presents the results for k-weight minimum spanning trees, $1+\epsilon$-approximate minimum spanning trees, and bipartiteness. Section 4 and 5 contain the deterministic algorithms.

2 A Randomized Connectivity Algorithm

2.1 A Deletions-Only Connectivity Algorithm

Definitions and notation: Let $G = (V, E)$ with $|V| = n$ and $|E| = m$. We use the convention that elements in V are referred to as *vertices*. Let $l = \lfloor \log m - \log\log n \rfloor + 1$. The edges of G are partitioned into l levels $E_1, \ldots, E_l$ such that $\cup_i E_i = E$, and for all $i \neq j$, $E_i \cap E_j = \emptyset$. For each i, we keep a forest F_i of *tree edges* such that F_i is a spanning forest of $(V, \cup_{j\leq i} E_j)$. So, for $i > 1$, $F_{i-1} \subseteq F_i$ and $F_i \setminus F_{i-1} \subset E_i$. Then F_l is a spanning tree of G and edges in $E \setminus F$ are referred to as *nontree edges*. A *spanning tree* T *on level* i is a tree of F_i.

All nontree edges incident to vertices in T are stored in a data structure that is described in more detail below. The *weight* of T, denoted *w(T)*, is the number of nontree edges incident to the spanning tree, where edges whose both endpoints lie in the spanning tree are counted twice. The *size* of T, denoted *s(T)*, is the number of vertices in T. A tree is *smaller* than another tree if its size is no greater than the other's. We say level i is *below* level $i+1$.

2.1.1 The Algorithm

Initially all edges are in E_1, and we compute F_1 which is a spanning tree of G.

When an edge e is deleted, remove e from the graph containing it. If e is a tree edge, let i be the level such that $e \in E_i$. Call $Replace(e, i)$.

Replace(e, i)

Let T be the level i tree containing edge e and let T_1 and T_2 be the two subtrees of T that resulted from the deletion of e, such that $s(T_1) \leq s(T_2)$.

- If $w(T_1) \leq \log^2 n$ goto *Case 2.*
- *Sample:* We sample $c \log^2 m$ nontree edges of E_i incident to vertices of T_1 for some appropriate constant c. An edge with both endpoints in T_1 is picked with probability $2/w(T_1)$ and an edge with one endpoint in T_1 is picked with probability $1/w(T_1)$.
- *Case 1: Replacement edge found:* If one of the sampled edges connects T_1 to T_2 then add it to all F_j, $j \geq i$.
- *Case 2: Sampling unsuccessful:* If none of the sampled edges connects T_1 and T_2, search all edges incident to T_1 and determine $S = \{$nontree edges connecting T_1 and $T_2\}$.
  - If $|S| > w(T_1)/(2c' \log n)$ choose one element of S and add it to F_j, $j \geq i$.
  - If $0 < |S| \leq w(T_1)/(2c' \log n)$, remove the elements of S from E_i, and insert them into E_{i+1}. Then add one of the newly inserted edges to F_j, $j > i$.
  - If $S = \emptyset$ then if $i < l$ then do
    $Replace(e, i + 1)$. Else stop.

2.1.2 Proof of Correctness

We first show that all edges are contained in $\cup_{i \leq l} E_i$, i.e., when $Replace(e, l)$ is called, and Case 2 occurs, no edges will be inserted into E_{l+1}. We use this fact to argue that if a replacement edge exists, it will be found.

Let m_i be the number of edges ever in E_i.

Lemma 2.1 *For all smaller trees T_1 on level i, $\sum w(T_1) \leq 2m_i \log n$.*

Proof: The proof follows [6]. When a tree is split into two trees and an endpoint of an edge is contained in the smaller tree, the size of the tree which contains that endpoint has been halved. Thus, over the course of the algorithm, each endpoint of an edge is inci dent to a smaller tree at most $\log n$ times in a given level, and, for all such trees T_1 on level i, $\sum w(T_1) \leq 2m_i \log n$. ∎

Lemma 2.2 *For any i, $m_i \leq m/c'^{i-1}$.*

Proof: We show the lemma by induction. It clearly holds for $i = 1$. Assume it holds for E_{i-1}. When summed over all smaller trees T_1, $\sum w(T_1)/(2c' \log n)$ edges are added to E_i. By Corollary 2.1, $\sum w(T_1) \leq 2m' \log n$ where m' is the total number of edges ever in level $i - 1$, this implies that the total number of edges in E_i is no greater than m/c'^{i-1}. ∎

Choosing $c' = 2$ and observing that edges are never moved to a higher level from a level with less than $2 \log n$ edges gives the following corollary.

Corollary 2.3 *For $l = \lfloor \log m - \log \log n \rfloor + 1$ all edges of E are contained in some E_i, $i \leq l$.*

The following relationship, which will be useful in the running time analysis, is also evident.

Corollary 2.4 $\sum_i m_i = O(m)$.

Theorem 2.5 *F_i is a spanning forest of $(V, \cup_{j \leq i} E_j)$.*

Proof: Initially, this is true, since we compute F_1 to be the spanning tree of $E = E_1$.

Consider the first time it fails, when a tree edge e is deleted, and a replacement edge exists but is not found. By Corollary 2.3, the replacement edge lies in some E_i, $i \leq l$.

Let i be the minimum level at which a replacement edge exists. Let e be in E_k. Then we claim $i \geq k$. Assume not. Then let $\{r, s\}$ be a replacement edge at level i. Since r and s are connected, there is a path from r to s in F_i. Since $e \in F_k$, e is not in this path. Hence, $\{r, s\}$ is not a replacement edge for e. Thus $k \leq i$ and $Delete(e, i)$ will be called.

Either a replacement edge will be found by sampling or every edge incident to T_1 will be examined. We claim that every replacement edge $\{r, s\}$ is incident to T_1. Suppose it is not. By assumption, there is a path from r to s in $\cup_{j \leq i} F_j$. If this path includes e then either r or s is in T_1. If it doesn't include e, then $\{r, s\}$ forms a cycle with $F - e$ contradicting the assumption that $\{r, s\}$ is a replacement edge. ∎

2.1.3 The Euler Tour Data Structure

In this subsection we present the data structure that we use to implement the algorithm of the previous section efficiently. We encode an arbitrary tree T with n vertices using a sequence of $2n-1$ symbols, which is generated as follows: Root the tree at an arbitrary vertex. Then call $ET(root)$, where ET is defined as follows:

$ET(x)$

visit x;
for each child c of x do
    ET(c);
    visit x.

Each edge of T is visited twice and every degree-d vertex d times, except for the root which is visited $d+1$ times. Each time any vertex u is encountered, we call this an *occurrence* of the vertex and denote it by o_u.

New encodings for trees resulting from splits and joins of previously encoded trees can easily be generated. Let $ET(T)$ be the sequence representing an arbitrary tree T.

Procedures for modifying encodings

1. **To delete edge $\{a,b\}$ from T:** Let T_1 and T_2 be the two trees which result, where $a \in T_1$ and $b \in T_2$. Let $o_{a_1}, o_{b_1}, o_{a_2}, o_{b_2}$ represent the occurrences encountered in the two traversals of $\{a,b\}$. If $o_{a_1} < o_{b_1}$ and $o_{b_1} < o_{b_2}$ then $o_{a_1} < o_{b_1} < o_{b_2} < o_{a_2}$. Thus $ET(T_2)$ is given by the interval of $ET(T)$ $o_{b_1}, \ldots, o_{b_2}$ and $ET(T_1)$ is given by splicing out of $ET(T)$ the sequence $o_{b_1}, \ldots, o_{a_2}$.

2. **To change the root of T from r to s:** Let o_s denote any occurrence of s. Splice out the first part of the sequence ending with the occurrence before o_s, remove its first occurrence (o_r), and tack it on to the end of the sequence which now begins with o_s. Add a new occurrence o_s to the end.

3. **To join two rooted trees T and T' by edge e:** Let $e = \{a,b\}$ with $a \in T$ and $b \in T'$. Given any occurrences o_a and o_b, reroot T' at b, create a new occurrence o_{a_n} and splice the sequence $ET(T')o_{a_n}$ into $ET(T)$ immediately after o_a.

If the sequence $ET(T)$ is stored in a balanced search tree of degree b, and height $O(\log n/\log b)$ then one may insert an interval or splice out an interval in time $O(b \log n/\log b)$, while maintaining the balance of the tree, and determine if two elements are in the same tree, or if one element precedes the other in the ordering in time $O(\log n/b)$.

Aside from lists and arrays, the only data structures used in the connectivity algorithm are trees represented as sequences which are stored in balanced b-ary search trees. We next describe these data structures.

Data structures: We have two options for storing the nontree edges: the first is simpler and is explained here. The second shaves off a factor of $\log \log n$ from the update time by reducing the cost of sampling. It is described in the last subsection of this section.

For each spanning tree T on each level $i < l$, each occurrence of $ET(T)$ is stored in a node of a balanced binary search tree we call the *ET(T)-tree*. For each tree T on the last level l, $ET(T)$ is stored in a balanced $(\log n)$-ary search trees. Note that there are no nontree edges on this level. For each vertex $u \in T$, we arbitrarily choose one occurrence to be the *active* occurrence of u.

With the active occurrence of each vertex v, we keep an (unordered) list of nontree edges in level i which are incident to v, stored as an array. Each node in the ET-tree contains the number of nontree edges and the number of active occurrences stored in its subtree. Thus, the root of $ET(T)$ contains the weight and size of T.

In addition to storing G and F using adjacency lists, we keep some arrays and lists:

- For each vertex and each level, a pointer to the vertex's active occurrence on that level.

- For each tree edge, for each level k such that $e \in F_k$, pointers to each of the four (or three, if an endpoint is a leaf) occurrences associated with its traversal in F_k;

- For each nontree edge, pointers to the locations in the two lists of nontree edge containing the edge and the reverse pointers.

- For each level i, a list containing a pointer to each root of a *ET(T)-tree*, for all spanning trees T at level i, and for each root a pointer back to the list;

- For each level i, a list of tree edges in E_i and for each edge a pointer back to its position in the list.

2.1.4 Implementation

Using the data structures described above, the following operations can be executed on each spanning tree on each level. Let T be a spanning tree on level i.

- $tree(x,i)$: Return a pointer to $ET(T)$ where T is the spanning tree of level i that contains vertex x.
- $nontree\_edges(T)$: Return a list of nontree edges stored in $ET(T)$; each edge is returned once or twice.
- $sample\&test(T)$: Randomly select a nontree edge of E_i that has at least one endpoint in T, where an edge with both endpoints in T is picked with probability $2/w(T)$ and an edge with exactly one endpoint in T is picked with probability $1/w(T)$. Test if exactly one endpoint is in T, and if so, return the edge.
- $insert\_tree(e,i)$: Join by e the two trees on level i, each of which contains an endpoint of e.
- $delete\_tree(e,i)$: Remove e from the tree on level i which contains it.
- $insert\_nontree(e,i)$: Insert the nontree edge e into E_i.
- $delete\_nontree(e)$: Delete the nontree edge e.

The following running times are achieved using a binary search tree: *tree*, *sample&test*, *insert_non_tree*, *delete_non_tree*, *delete_tree*, and *insert_tree* in $O(\log n)$ and $nontree\_edges(T)$ in $O(m' \log n)$, where m' is the number of moved edges. On the last level l, when a $(\log n)$-ary tree is used, the running time of *delete_tree* and *insert_tree* is increased to $O(\log^2 n/\log\log n)$ and the running time of *tree* is reduced to $O(\log n/\log\log n)$. We describe the implementation details of these operations.

tree(x,i): Follow the pointer to the active occurrence of x at level i. Traverse the path in the $ET(T)$-tree from the active occurrence to the root and return a pointer to this root.

nontree_edges(T): Traverse all nodes of $ET(T)$ and output every non-empty list of nontree edges encountered at a node.

sample&test(T): Let T be a level i tree. Pick a random number j between 1 and $w(T)$ and find the j^{th} nontree edge $\{u,v\}$ stored in the $ET(T)$. If $tree(u,l) \neq tree(v,l)$ then return the edge.

insert_tree(e,i) Determine the active occurrences of the endpoints of e on level i and follow Procedure 3 for joining two rooted trees, above. Update pointers to the root of the new tree and the list of tree edges on level i.

delete_tree(e,i): Let $e = \{u,v\}$. Determine the four occurrences associated with the traversal of e in the tree on level i which contains e and delete e from it, following Procedure 1, above. Update pointers to the roots of the new trees, the list of tree edges, and (if necessary) the active occurrences of u and v.

insert_nontree(e,i): Determine the active occurrences of the endpoints of e on level i and add e to the list of edges stored at them.

delete_nontree(e): Follow the pointers to the two locations of e in lists of non-tree edges and remove e.

Using these functions, the deletions only algorithm can be implemented as follows.

To initialize the data structures: Given a graph G compute a spanning forest of G. Compute the $ET(T)$ for each T in the forest, select active occurrences, and set up pointers as described above. Initially, the set of trees is the same for all levels. Then insert the nontree edges with the appropriate active occurrences into level 1 and compute the number of nontree edges in the subtree of each node.

To answer the query: "Are x and y connected?": Test if $tree(x,l) = tree(y,l)$.

To update the data structure after a deletion of edge e=$\{u,v\}$: If e is in E_i, then do $delete\_tree(e,j)$ for $j \geq i$, and call $Replace(u,v,i)$.

If e is not a tree edge, execute $delete\_nontree(e)$.

Replace(u,v,i)

if $s(tree(u,i)) \leq s(tree(v,i))$ **then** $T_1 = tree(u,i)$ **else** $T_1 = tree(v,i)$
if $w(T_1) < \log^2 n$ **goto** Case 2.
Repeat $sample\&test(T_1)$ $c\log^2 n$ times.
Case 1: Replacement edge e' is found
  $delete\_nontree(e')$;
  for $j \geq i$ **do** $insert\_tree(e',j)$.
Case 2: Sampling unsuccessful
for each edge $\{u,v\} \in nontree\_edges(T_1)$ **do**
  if $tree(u,l) \neq tree(v,l)$ **then** add $\{u,v\}$ to S.
    $\{S = \{$edges with exactly one endpoint in T_1 $\}$ $\}$.
  Case 2.1: $|S| \geq w(T_1)/(2c' \log n)$
    Select one $e' \in S$;
    for $j \geq i$ **do** $insert\_tree(e',j)$.
  Case 2.2: $0 < |S| < w(T_1)/(2c' \log n)$
    Choose one edge $e' \in S$ and remove it from S;
    for $j > i$ **do** $insert\_tree(e',j)$;
    for every edge $e'' \in S$ **do** $delete\_non\_tree(e'')$; $insert\_nontree(e'',i+1)$.
  Case 2.3: $S = \emptyset$
    if $i < l$ **then** $Replace(u,v,i+1)$.

2.1.5 Analysis of Running Time

We show that the amortized cost per deletion is $O(\log^3 n)$ if there are m deletions.

In all cases where a replacement edge is found, $O(\log n)$ *insert_tree* operations are executed, costing $O(\log^2 n)$. In addition:

Case 1: Sampling is successful. The cost of *sample&test* is $O(\log n)$ and this is repeated $O(\log^2 n)$ times, for a total of $O(\log^3 n)$.

Case 2: Sampling is not successful or $w(T_1) < \log^2 n$. We refer to the executing $nontree\_edges(T_1)$ and the testing of each edge as the cost of *gathering and testing* the nontree edges. The first operation costs $O(\log n)$ per nontree edge and the second is $O(\log n/\log\log n)$ per nontree edge, for a total cost of $O(w(T_1)\log n)$. Now there are three possible cases.

Case 2.1: $|S| \geq w(T_1)/(2c' \log n)$. If $w(T_1) < \log^2 n$, we charge the cost of gathering and testing to the *delete* operation. Otherwise, the probability of this subcase occurring is $(1 -$

$1/(2c' \log n))^{c \log^2 n} = O(1/n^2)$ for $c = 4c'$, and the total cost of this case is $O(w(T_1) \log n)$. Thus this contributes an expected cost of $O(\log n)$ per operation.

Case 2.2 and 2.3: $|S| < w(T_1)/(2c' \log n)$ Each *delete_nontree*, *insert_nontree*, and *tree* costs $O(\log n)$, for a total cost of $O(w(T_1) \log n)$. In this case $tree(u, i)$ and $tree(v, i)$ are not reconnected. Note that only edges incident to the smaller tree T_1 are gathered and tested. Thus, over the course of the algorithm, the cost incurred in these cases on any level i is $O(\sum w(T_1) \log n)$, where the sum is taken over all smaller trees T_1 on level i. From Lemma 2.1, we have the $\sum w(T_1) \leq 2m_i \log n$. From Corollary 2.4, $\sum_i m_i = O(m)$, giving a total cost of $O(m \log^2 n)$.

2.2 A Fully Dynamic Connectivity Algorithm

2.2.1 The Algorithm

Next we also consider insertions. When an edge $\{u, v\}$ is inserted into G, add $\{u, v\}$ to E_l. If u and v were not previously connected in G, i.e., $tree(u, l) \neq tree(v, l)$, add $\{u, v\}$ to F_l.

We let the number of levels $l = \lceil 2 \log n \rceil$. A rebuild of the data structure is executed periodically. A *rebuild of level* i, for $i \geq 1$, is done by a $move\_edges(i)$ operation, which moves all tree and nontree edges in E_j for $j < i$ into E_i. Also, for each $j < i$, all tree edges in E_j are inserted into all F_k, $j < k \leq i$. Note that after a rebuild at level i, E_j, for $j > i$, contains no edges, and $F_j = F_i$ for all $j \geq i$, i.e., the spanning trees on level $j \geq i$ span the connected components of G.

After each insertion, we increment I, the number of insertions modular $2^{\lceil 2 \log n \rceil}$ since the start of the algorithm. Let j be the greatest integer k such that $2^k | I$. After an edge is inserted, a rebuild of level $l - j - 1$ is executed. If we represent I as a binary counter whose bits are $b_0, ..., b_{l-1}$, where b_0 is the most significant bit, then a rebuild of level i occurs each time the i^{th} bit flips to 1.

2.2.2 Proof of Correctness

The proof of correctness is the same as the one for the deletions-only case, except that we must set the value of l to $\lceil 2 \log n \rceil$ and alter the argument which shows that all edges are contained in $\cup_{i \leq l} E_i$.

We define an *i-period* to be any period beginning right after a rebuild of levels $j \leq i$, or the start of the algorithm and ending right after the next rebuild of level $j' \leq i$. I.e., an i-period starts right after the flip to 1 of some bit $j \leq i$, or the start of the algorithm and ends with next such flip. Note that there are two types of i-periods: (A) begins immediately after a rebuild of level $j > i$, i.e., all edges from E_i are moved to some E_j with $j < i$ (b_j, $j > i$, flips to 1); (B) begins immediately after a rebuild of level i, i.e. all edges from $\cup_{j > i} E_j$ are moved into E_i (b_i flips to 1).

It is easy to see that an *i-1*-period consists of two parts, one type A i-period, followed by one type B i-period, since any flip to 1 by some bit b_j, $j < i - 1$ must be followed by a flip of b_i to 1 before a second bit $b_{j'}$, $j' < i$ flips to 1.

Theorem 2.6 *Let a_i be the number of edges in E_i during an* i*-period. Then $a_i < n^2/2^{i-1}$.*

Proof: By a proof analogous to that of Lemma 2.1 we have:

Lemma 2.7 *For all smaller trees T_1 on level i which were searched between two consecutive rebuilds of levels $j, j' \leq i$, $\sum w(T_1) \leq 2a_i \log n$.*

We now bound a_i. Note that we may restrict our attention to the edges which are moved into E_i during any one *i-1*-period since E_i is empty at the start of each *i-1 period.*

Thus, an edge is in E_i either because it was passed up from E_{i-1} during one *i-1*-period or moved there in a single rebuild of level i.

Since E_k for $k \geq i$ was empty at the start of the *i-1*-period, any edge moved to E_i during the rebuild of level i was passed up from E_{i-1} to E_i and then to E_{i-1} during the type A i-period (i.e., the first part of the *i-1*-period) or was inserted into G during the type A i-period. We have

$$a_i \leq h_{i-1} + b_i,$$

where h_{i-1} is the maximum number of edges passed up from E_{i-1} to E_i during a single $i-1$-period (i.e., an A i-period followed by a B i-period) and b_i is the number of edges inserted into G during a single i-period.

The number of edges inserted into G during an i-period is 2^{l-i-1}.

To bound h_i we use Lemma 2.7 to bound $\sum w(T_1)$, summed over all smaller trees T_1 which are searched on level $i-1$ during an $i-1$-period. As in the proof of Lemma 2.2, we can now bound h. Choosing $c' = 4$ gives $h_{i-1} \leq a_{i-1}/4$.

Substituting for h and b yields

$$a_i \leq 2^{l-i-1} + a_{i-1}/4.$$

Choosing $l = \lceil 2 \log n \rceil$ and noting that $a_1 < n^2$, an induction proof shows that $a_i < n^2/2^{i-1}$. ∎

This implies $a_l < 2$ and edges are never passed up to a_{l+1}. We have:

Corollary 2.8 *For $l = \lceil 2 \log n \rceil$ all edges of E are contained in some E_i, $i \leq l$.*

2.2.3 Analysis of the Running Time.

To analyze the running time, note that the analysis of Case 1 and Case 2.1, above, are not affected by the rebuilds. However, (1) we have to bound the cost incurred during an insertion, i.e. the cost of the operation *move_edges* and (2) in Case 2.2 and 2.3, the argument that $O(m_i \log n)$ edges are gathered and tested (using *nontree_edges* and *tree*) on level i during the course of the algorithm must be modified.

The cost of (1), i.e. the cost of executing *move_edges*(i) is the cost of moving each tree edge and each nontree edge in E_j, $j > i$ into E_i and the cost of updating all the F_k, $j < k \leq i$.

To analyze the first part, we note that each move of an edge into E_i costs $O(\log n)$ per edge. The number of edges moved is no greater than $\sum_{j>i} a_j < n^2/2^{i_1}$. Thus the cost incurred is $O(n^2 \log n/2^i)$.

The cost of inserting one tree edge into any given level is $O(\log n)$ per edge. A tree edge is added only once into a level, since a tree edge is never passed up. Thus this cost may be charged to the edge for a total cost of $O(\log^2 n)$ per edge.

We analyze the cost of (2).

For $i < l$, if fewer than 2^{l-i-1} edges have been inserted since the start of the algorithm then no rebuilds have occurred at level i or lower, and the analysis for level i of the deletions-only argument holds. That is, the costs incurred on level i is bounded above by $O(m \log^2 n / c'^i)$, where m is the number of edges in the initial graph.

Applying Lemma 2.7, we conclude that the cost for the gathering and testing of edges from all smaller trees T_1 on level i during an i-period is $O(2a_i \log n * \log n) = O(n^2 \log^2 n / 2^i)$.

For $i = l$, we note that since there are $O(1)$ edges in E_l at any given time, and since the cost is $O(\log n)$ per edge for gathering and testing, the total cost for each instance of gathering and testing on this level is $O(\log n)$.

We use a potential function argument to charge the cost of (1) and (2) to the insertions. Each new insertion contributes $c'' \log^2 n$ tokens toward the bank account of each level, for a total of $\Theta(\log^3 n)$ tokens. Since an i-period occurs every $n^2/2^i$ insertions, the tokens contributed by these insertions can pay for the $O(n^2 \log^2 n / 2^i)$ cost of the gathering and testing on level i during the i-period and the $O(n^2 \log n / 2^i)$ cost of $move\_edges(i)$ incurred at most once during the i-period.

2.3 Improvements

In this section, we present a simple "trick" which reduces the cost of testing all nontree edges incident to a smaller tree T_1 to $O(1)$ per edge so that the total cost of gathering and testing edges incident to T_1 is $O(1)$ per edge.

2.3.1 Constant Time for Gathering and Testing

As noted above, since the nontree edges incident to an ET-tree are available as a list, the time needed to retrieve these edges is $O(1)$ per edge. One can also test each nontree edge in $O(1)$ time, i.e., determine the set S of all nontree edges which contain only one endpoint, by running through the list three times. For each edge in the list, initialize the i, j entry of an $n \times n$ array. Then use these entries to count the number of times each edge appears in the list. Traverse the list again and add to S any edge whose count is one.

2.3.2 Constant Query and Test Time for Deletions-Only Algorithms

We note that determining whether two vertices i and j are in the same component, i.e., is $tree(i) = tree(j)$, can be speeded up to $O(1)$ for the deletions-only algorithm. A component is split when $Replace(e, l)$ is called and no replacement edge is found. In that case, label the nodes of the smaller component T_1 with a new label. The cost of doing so is proportional to the size of T_1. Over the course of the algorithm, the cost is $O(n \log n)$ since each node appears in a smaller component no more than $\log n$ times. Then $tree(i) = tree(j)$ iff i and j have the same label.

This improvement does not affect the asymptotic running time of the randomized connectivity algorithm, as that is dominated by the cost of the random sampling. However it will be used in the deterministic algorithms presented later in the paper.

3 Randomized Algorithms for Other Dynamic Graph Problems

In this section, we show that some dynamic graph problems have polylogarithmic expected update time, by reducing these problems to connectivity and we give an alternative algorithm for maintaining the minimum spanning tree.

3.1 A k-Weight Minimum Spanning Tree Algorithm

The k-weight minimum spanning tree problem is to maintain a minimum spanning forest in a dynamic graph with no more than k different edgeweights at any given time.

Let $G = (V, E)$ be the initial graph. Compute the minimum spanning forest F of G. We define a sequence of subgraphs $G_1, G_2, ..., G_k$ on nodeset V and with edgesets $E_1, E_2, ..., E_k$ as follows: Let $E_i = \{$edges with weight of rank $i\} \cup F$. If initially, there are $l < k$ distinct edgeweights, then for $i > l$, $E_i = F$. These E_i are called "extras". The spanning forests of each G_i are maintained as in the connectivity algorithm. These forests and F are also stored in dynamic trees. The subgraphs are ordered by the weight of its edgeset and stored in a balanced binary tree.

To insert edge $\{u, v\}$ into G: determine if u and v are connected in F. If so, find the maximum cost edge e on the path from u to v in F. If the weight of e is greater than the weight of $\{u, v\}$, replace e in F by $\{u, v\}$. If u and v were not previously connected, add $\{u, v\}$ to F. Otherwise, just add $\{u, v\}$ to E_j where j is the rank of the weight of $\{u, v\}$. If $\{u, v\}$ is the only edge of its weight in G, then create a new subgraph by adding $\{u, v\}$ to an extra and inserting it into the ordering of the other G_i. Update the E_i to reflect the changes to F.

To delete edge $\{u, v\}$ from G: Delete $\{u, v\}$ from all graphs containing it. To update F: If $\{u, v\}$ had been in F, then a tree T in F is divided into two components. Find the minimum i such that u and v are connected in G_i using binary search on the list of subgraphs. Now, search the path from u to v in the spanning forest of G_i to find an edge crossing the cut in T. Use binary search: Let x be a midpoint of the path. Recurse on the portion of the path between u and x if u and x are not connected in F; else recurse on the path between x and v.

Correctness: When an edge $\{u, v\}$ is inserted and its cost is not less than the cost of the maximum cost edge on the tree path between u and v, then the minimum spanning forest F is unchanged. If the cost of $\{u, v\}$ is less than the cost of the maximum cost edge e' on the tree path between u and v, then replacing e' by $\{u, v\}$ decreases the cost of the minimum spanning tree by the maximum possible amount and gives, thus, the minimum spanning tree of $G \cup \{u, v\}$.

Analysis of Running Time: The algorithm (1) determines how F has to be changed, and (2) updates the data structures. (1) After an insertion the maximum cost edge on the tree path between u and v can be determined in time $O(\log n)$ using the dynamic tree data structure of F. After a deletion, it takes time $O(\log^2 n / \log\log n)$ to find the minimum i such that u and v are connected in G_i, since a connectivity query on a level takes time $O(\log n / \log\log n)$. The midpoint of the tree path between u and v in G_i can be determined in time $O(\log n)$ using the dynamic tree data structure of the spanning tree of G_i. The algorithm recurses at most $\log n$ times to determine the replacement edge for $\{u, v\}$, for a total of $O(\log^2 n)$.

(2) The insertion or deletion of $\{u, v\}$ into E_i, where i is the rank of the weight of $\{u, v\}$, takes amortized expected time $O(\log^3 n)$. If F changes, one additional insertion and deletion is executed in every E_j. For each update there are a constant number of operations in the dynamic tree of F and of the spanning tree of every E_j, each costing $O(\log n)$. Thus, the amortized expected update time is $O(k \log^3 n)$.

3.2 A $1+\epsilon$-Approximate Minimum Spanning Tree Algorithm

Given a graph with weights between 1 and U, a *$1+\epsilon$-approximation of the minimum spanning tree* is a spanning tree whose weight is within a factor of $1+\epsilon$ of the weight of the optimal. The problem of maintaining a $1+\epsilon$ approximation is easily seen to be reducible to the k-weight MST problem, where a weight has rank i if it falls in the interval $[(1+\epsilon)^i, (1+\epsilon)^{i+1})$ for $i = 0, 1, .., \lfloor \log U / \log(1+\epsilon) \rfloor$. This yields an algorithm with amortized cost $O((\log^3 n \log U)/(\epsilon \log\log n))$.

3.3 A Bipartiteness Algorithm

The bipartite graph problem is to answer the query "Is G bipartite?" in $O(1)$ time, where G is a dynamic graph.

We reduce this problem to the 2-weight minimum spanning tree problem. We use the fact that a graph G is bipartite iff given any spanning forest F of G, each nontree edge forms an even cycle with F. Call these edges "even edges" and the remaining edges "odd". We also use the fact that if an edge e in F is replaced with an even edge then the set of even edges is preserved. Let C be the cut in F induced by removing e. If e is replaced with an odd edge then for each nontree edge e' which crosses C the parity of e' changes. We replace an edge by an odd replacement edge only if there does not exists an even replacement edge. Thus, the parity of an even edge never changed. F is stored as a dynamic tree.

Our algorithm is: generate a spanning forest F of the initial graph G. All tree and even nontree edges have weight 0. Odd edges have weight 1. If no edges have weight 1, then the graph is bipartite.

When an edge is inserted, determine if it is odd or even by using the dynamic tree data structure of F, and give it weight 1 or 0 accordingly.

When an edge is deleted, if it is a tree edge, and if it is replaced with an odd edge (because there are no weight 0 replacements), remove the odd edge and find its next replacement, remove that, etc. until there are no more replacements. Then relabel the replacement edges as even and add them back to G.

Correctness: When an edge is inserted, the algorithm determines if it is even or odd. If an edge is deleted, we replace it by an even edge if possible. This does not affect the parity of the remaining edges. If no even replacement edge exists, but an odd replacement edge, the parity of every edge on the cut changes. However, since no even edge exists on the cut, it suffices to make all odd edges into even edges.

Analysis of Running Time: An even edge never becomes odd. Thus, the weight of an edge changes at most once, which shows that an insertion of an edge causes the edge to be added to the data structure at most once with weight 1 and at most once with weight 0. The deletion of an edge leads to the removal of the edge from the data structure. Thus, amortized expected update time is $O(\log^3 n)$.

4 A Deterministic Minimum Spanning Tree Algorithm

In this section we present a fully dynamic deterministic minimum spanning tree algorithm with amortized time $O(\sqrt{m} \log n)$ per operation, where m is the current number of edges in G. We note that the technique of sparsification [4], when applied on top of this algorithm, will yield an algorithm with $O(\sqrt{n}\log n)$ amortized time per operation.

4.1 A Deletions-Only Minimum Spanning Tree Algorithm

The data structures which we use are very similar to those of the randomized connnectivity algorithm. Instead of random sampling, we always exhaustively search edges incident to the smaller components. Edges are stored in $\sqrt{m}$ levels, according to their weight and do not move between levels. A deletions-only data structure is kept for "old" edges. Newly inserted edges are kept in a separate level. Periodically, a rebuild is executed in which the deletions-only data structure is revised to include the newly inserted edges.

Let $G = (V, E)$ with $|V| = n$ and $|E| = m$. Rank the edges by weight. The edges of G are partitioned into l levels $E_1, \ldots, E_l$ such that E_i for $i \le l$ contains all edges with rank j, where $(i-1)\sqrt{m} < j \le i\sqrt{m}$. Compute a minimum spanning forest F of G. For each i, we keep a forest F_i of *tree edges* such that F_i is a minimum spanning forest of $(V, \cup_{j\le i} E_j)$, $F_{i-1} \subseteq F_i$ for $i > 1$, and $F_l = F$. Note that $F_i \setminus F_{i-1} \subseteq E_i$.

4.1.1 The Algorithm

To update the data structure after a deletion of edge e=$\{u$,$v\}$**:** If e is not a tree edge, then do *delete_nontree*(e). If e is a tree edge in E_i, then do *delete_tree*(e, j) for $j \ge i$, and call *Replace*(u, v, i).

Replace(u,v,i)

if $s(tree(u,i)) \le s(tree(v,i))$ **then** $T_1 = tree(u,i)$
  else $T_1 = tree(v,i)$
Gather and test nontree edges incident to T_1
Case 1: Replacement edge is found
  Let e' be a minimum weight replacement edge;
  for $j \ge i$ **do** *insert_tree*(e', j).
Case 2: No Replacement edge is found
  if $i < l$ **then** *Replace*$(u, v, i+1)$.

4.1.2 Implementation and Analysis of Running Time

We use the same data structure as in the randomized algorithm, storing all nontree edges in E_i in the appropriate ET-tree on level i. Each ET-tree is stored as a binary tree; all nontree edges are stored in NT-lists, so that gathering and testing of all edges incident to T_1 may be done in $w(T)$ time. (See Improvements Section 2.3.1, above.)

When a tree edge is deleted and a replacement edge is sought, the total cost of gathering and testing of nontree edges in a given level i is charged to the deletion, if a replacement edge is found in E_i, or to the level, if not. We consider the latter case first. Then T_1 becomes a new component of F_i. The cost of gathering and testing edges incident to T_1 is $O(w(T_1))$. On a given level, $\sum w(T_1) \le \sqrt{m}\log n$ (see Lemma 2.1). Over all levels, the total cost is $O(m\log n)$.

The cost of gathering and testing in the level in which the replacement edge is found is no greater than the cost of gathering and testing every edge in that level, or $O(\sqrt{m})$.

In addition, for each tree edge, no more than $\sqrt{m}$ *insert_tree* operations are executed, once per level, for a total cost of $O(\sqrt{m}\log n)$. The cost of *delete_nontree* is $O(\log n)$.

The cost is thus $O(\sqrt{m}\log n)$ per edge deletion plus a total cost of $O(m \log n)$ during the course of the algorithm.

4.2 A Fully Dynamic Minimum Spanning Tree Algorithm

4.2.1 The Algorithm

A partial rebuild of the data structure is executed after $\sqrt{m_0}$ operations, where m_0 is the number of edges which were in the graph at the last rebuild, or initially, if there has been no rebuild. These edges are referred to as *old edges* and, if they are still in G, are stored in the deletions-only data structure described above. All *newly inserted* edges are stored in a separate set E_{l+1}. Then F_l is a minimum spanning tree consisting only of the old edges, i.e., of $G(V, \cup_{1\le i\le l} E_i)$. Let F be a minimum spanning tree of $G(V,E)$, i.e., of old and new edges. We store F in a dynamic tree data structure. For each E_i with $i \le l$ we keep a list of its edges ordered by rank.

To rebuild: When the number of updates (insertions or deletions) since the last rebuild or the start of the algorithm reaches $\sqrt{m}_0$ then m_0 is reset to the current number of edges in the graph. Insert each edge in E_{l+1} into an appropriate E_i, for $i \le l$ and redistribute the old edges among the E_i so that for $i \le l$, E_i contains all edges with rank j, where $(i-1)\sqrt{m_0} < j \le i\sqrt{m_0}$. All edges are now "old edges". If tree edge of level i moves to level i' then add the edge to all levels $i, \ldots i'-1$ if $i < i'$ or remove the edge from all levels $i', \ldots i-1$ if $i' < i$.

To delete an edge e from G: If e is an old tree edge, then update the deletions-only data structure. If e does not belong to F, stop. Otherwise, delete $e = \{u,v\}$ from F and find the minimum cost edge e' with exactly one endpoint in T_1 in E_{l+1}. Add to F the smaller-cost edge of e' and of the replacement edge in the deletions only data structure (if existent).

To insert an edge e to G: Add the edge to E_{l+1} and test if it replaces an edge of F. That is, use the dynamic tree to find the heaviest edge in the path between e's endpoints, if there is such a path, and determine if e is lighter. If yes, modify F accordingly.

4.2.2 Analysis of Running Time

The cost of an edge insertion is $O(\log n)$, since it consists of constant number of operations on the dynamic tree data structure.

The cost of an edge deletion may include the cost of updating the dynamic tree data structure, which is $O(\log n)$; the cost of testing edges in E_{l+1}, which is $O(\sqrt{m_0}\log n)$; and the cost of updating the deletions-only data structure. We show how to bound the latter.

The cost involved in the deletions-only data structure between rebuilds is given by the analysis in Section 4.1.2 : $O(\sqrt{m_0}\log n)$ per deletion of a tree edge plus an overall cost of $O(m_0 \log n)$ between rebuilds, which is $O(\sqrt{m_0}\log n)$ per update operation.

We bound next the cost of a rebuild. First we discuss adding the edges to the appropriate level, then we discuss "rebalancing" the size of a level. Determining the appropriate level for an edge takes time $O(\log n)$ using a binary search. Moving a nontree edge from E_{l+1} to E_i with $i \le l$ costs $O(\log n)$ per edge, for a total of $O(\sqrt{m}_0 \log n)$. Moving a tree edge from E_{l+1} to E_i with

$i \leq l$ requires the insertion of the edge into all levels $j \geq i$ and costs $O(\sqrt{m}_0 \log n)$ per edge, for a total of $O(m_0 \log n)$. Any edge in E_i, for $i \leq l$, is never moved by more than one level during the "rebalancing", i.e., either into E_{i-1} or E_{i+1}. This costs $O(\log n)$ per tree or non-tree edge. Thus the time of a rebuild is $O(m_0 \log n)$.

The cost of a rebuild, when amortized over the $\sqrt{m}_0$ updates which occur, is $O(\sqrt{m_0} \log n)$.

Let $m > 0$ be the number of edges currently in the graph when an operation occurs. Since $m = O(m_0)$, the cost of this algorithm per edge deletion or insertion is $O(\sqrt{m} \log n)$.

5 A Deterministic Connectivity Algorithm

The above minimum spanning tree algorithm can be easily converted into an algorithm for connectivity with the same running time. In this section, we give an alternative connectivity algorithm with $O(\sqrt{n} \log n)$ amortized update time without the use of sparsification. The data structure used here is similar to those used in the previously presented algorithms. There are two important differences. As in the previous algorithm, edges are partitioned into levels $E_1, E_2, \ldots$. Here, each level contains at most $\sqrt{n}$ edges, and levels are added as needed. That is, initially, $m_0/\sqrt{n}$ levels with $\sqrt{n}$ (arbitrary) edges each are created. New edges are inserted into the highest level if it contains less than $\sqrt{n}$ edges, and into a newly created (empty) highest level, otherwise. The data structure as a whole is never rebuilt; new levels are added one at a time. If all edges in a level have been deleted, the level is discarded. We use the term *level i* to denote the ith non-empty level. As in the previous algorithm, edges do not move between levels.

The F_i are defined as before: For each i, we keep a forest F_i of *tree edges* such that F_i is a spanning forest of $(V, \cup_{j \leq i} E_j)$, and for $i > 1$, $F_{i-1} \subseteq F_i$ and $F_i \setminus F_{i-1} \subseteq E_i$. Here, the F_i are stored in "relaxed" versions of ET-trees, called *RET-trees*. Unlike the ET-trees used in the previous algorithms, RET-trees have the following property: If an edge on level i is deleted and a replacement edge is found on level j $(i < j)$, then only the RET-trees on levels i, $i+1$, ..., j have to be updated. This allows us to keep a larger number of levels.

5.1 The RET-Tree Data Structure

Let T be a spanning tree on level i and let $C(T)$ be the tree that is created from T by contracting every spanning tree on level $i-1$ to one node. Each node of $C(T)$ is called a *big* node. Thus, all edges of $C(T)$ belong to F_i.

An *RET(T)-tree* consists of an ET-tree for the sequence $ET(C(T))$ where the list of edges at the active occurrence of a vertex a is replaced with:

- A list $Nodes(a)$ of all the vertices of G that have been contracted to form a, in the following order: first all vertices that are incident to edges in E_i, then all other vertices. At each vertex v of G in $Nodes(a)$ we keep (1) a list of all the edges of F_i incident to v, (2) a list of all the nontree edges incident to v, (3) the number of nontree edges incident to v, and (4) a pointer to a.

- The number of edges incident to the vertices in $Nodes(a)$.

In addition, we keep an ordered list of nonempty levels stored in a doubly linked list L and, for each nonempty level, an array whose jth entry is a pointer to the location of the vertex j in the $Nodes$-list on level i which contains j.

The operations for RET-trees are the same as those defined for ET-trees, with the omission of *sample&test*, and the addition of two more operations defined below:

- *big_node*(u, i): Return the big node on level i that contains vertex u.
- *split_big_node*$(e, i+1, T_1)$: The operation assumes that e is a deleted edge of F_i, whose endpoints are (still) contained in the same big node a on level $i+1$. Let T_1 and T_2 be the spanning trees of level i containing the endpoints of e, where T_1 is the smaller of these two trees. Split a into two big nodes that represent T_1 and T_2.

To implement *big_node*(u, i), follow the pointer from level i, vertex u to the $Nodes$-list on level i which contains u, and the pointer from there to the big node containing u. Each call to *big_node* is dominated by the cost of finding level i in L, which takes time $O(\log n)$.

To implement *split_big_node*$(e, i+1, T_1)$, let T_1 denote the smaller of the two subtrees and let $e = \{u, v\}$. By traversing $ET(C(T_1))$ on level i determine the set U of all vertices of G in T_1. Using *big_node*$(x, i+1)$ for one node of U, determine the big node a that represents all nodes of U. Remove each node in U from $Nodes(a)$, create a new node b, and add all nodes of U into a new (ordered) $Nodes(b)$ list. Then update the number of edges incident to the nodes in $Nodes(a)$ and in $Nodes(b)$ accordingly. A call to *split_big_node* takes time proportional to the number of vertices in T_1 plus $O(\log n)$ for the call to *big_node*.

The implementation of the ET-tree operations need only be modified slightly, and each has a running time of $O(\log n)$. That is, in *insert_nontree*(T, e) and *delete_nontree*(e) an endpoint of e may be moved to the beginning or the end of the $Nodes$-list containing it. The implementation of *tree*(x, i), *nontree_edges*(T), *insert_tree*(e, i), and *delete_tree*(e, i) are unchanged, except for a constant number of calls to *big_node*.

5.2 The Algorithm

To update the data structure after a deletion of edge e=$\{u,v\}$**:** If e is not a tree edge, execute *delete_nontree*(e). If e is a tree edge on level i, then *delete_tree*(e, i) and call *Replace*(u, v, i).

Replace(u,v,i)

if $size(tree(u, i)) \leq size(tree(v, i))$ **then** $T_1 = tree(u, i)$ **else** $T_1 = tree(v, i)$
Gather and test nontree edges incident to T_1
Case 1: Replacement edge e' **is found**
  delete_nontree(e');
  insert_tree(e', i).
Case 2: No Replacement edge is found
  if $i \neq l$ **then**
    Let j be the next lowest nonempty level above i.
    split_big_node(e, j).
    Replace(u, v, j).

To update the data structure after an insertion of edge e=$\{u,v\}$**:** If the number of edges in level(l) is $\sqrt{n}$ then build a data structure for a new level and reset l to this new level. If u and v are connected then call *insert_nontree*$(e, tree(u, l))$ else *insert_tree*(e, l).

5.3 Implementation and Analysis of Running Time

To gather and test non-tree edges we store the nontree edges in NT-lists (see Section 2.3.1) so that gathering and testing of all edges incident to T_1 may be done in time $O(w(T_1))$ where $w(T_1)$ is the number of nontree edges incident to T_1.

We show that the amortized time per update operation is $O(\sqrt{n}\log n)$. Since building a data structure for a new level takes time $O(n)$ and occurs every $\sqrt{n}$ insertions, the total time for all insertions charge $O(n/\sqrt{n}) = O(\sqrt{n})$ to each insertion. Next we show that the total cost of all deletions is $O((m_0+k)\sqrt{n}\log n)$, where m_0 is the number of edges in the initial graph and k is the total number of update operations.

The search for a replacement edge takes time $O(w(T_1))$. We separately consider the cost of the search on the last level in which the search occurred (and either terminated successfully or discontinued because there were no more levels) and the cost on the other levels where the search occurred. The total cost of searches for the deletions on the last level is no more than $\sqrt{n}$ per search or $O((m_0+k)\sqrt{n}\log n)$. The total cost of searching on all levels where no replacement edge is found is $O(\sum w(T_1))$ summed over all T_1 which are created, on all levels. Applying Lemma 2.1, we have a total cost of $O((m_0+k)\sqrt{n}\log n)$.

The cost of *split_big_node* can be charged to the the nodes in T_1, for a total cost of $n\log n$ per level. Since at most $O((m_0+k)/\sqrt{n})$ levels are created during the whole course of the algorithms, the total cost of *split_big_node* is $O((m_0+k)\sqrt{n}\log n)$.

Thus the cost of update operations, when amortized over all updates is $O(\sqrt{n}\log n)$, for $k = \Omega(m_0)$.

6 Acknowledgements

We are thankful for David Alberts for comments on the presentation.

References

[1] D. Alberts and M. Rauch Henzinger, "Average Case Analysis of Dynamic Graph Algorithms", to appear in *Proc. 5th Symp. on Discrete Algorithms*, 1995.

[2] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph". *Proc. 1st Symp. on Discrete Algorithms*, 1990, 1–11.

[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] D. Eppstein, Z. Galil, G. F. Italiano, and T. Spencer. "Separator Based Sparsification for Dynamic Planar Graph Algorithms". *Proc. 25th Symp. on Theory of Computing*, 1993, 208–217.

[6] S. Even and Y. Shiloach, "An On-Line Edge-Deletion Problem", *J. ACM* 28 (1981), 1–4.

[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, "Ambivalent Data Structures for Dynamic 2-Edge-connectivity and k Smallest Spanning Trees", *Proc. 32nd Symp. on Foundations of Computer Science*, 1991, 632–641.

[9] M. L. Fredman and M. Rauch Henzinger, "Lower Bounds for Fully Dynamic Connectivity Problems in Graphs", submitted to *Algorithmica.*

[10] Z. Galil and G. F. Italiano, "Fully Dynamic Algorithms for 2-Edge Connectivity", *SIAM J. Comput.* 21 (1992), 1047–1069.

[11] M. Rauch Henzinger, "Fully Dynamic Cycle-Equivalence in Graphs", *Proc. 35th Symp. on Foundations of Computer Science*, 1994, 744–755.

[12] M. Rauch Henzinger, "Approximating Minimum Cuts under Insertions", to appear in *Proc. 22nd International Colloquium on Automata, Languages, and Programming*, 1995.

[13] M. Rauch Henzinger and H. La Poutré, "Sparse Certificates for Dynamic Biconnectivity in Graphs", to appear in *Proc. 3rd European Symposium on Algorithms* 1995.

[14] D. R. Karger, "Using Randomized Sparsification to Approximate Minimum Cuts", *Proc. 4th Symp. on Discrete Algorithms* 1994, 424–432.

[15] 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.

[16] M. H. Rauch, "Fully Dynamic Biconnectivity in Graphs". *Proc. 33rd Symp. on Foundations of Computer Science*, 1992, 50–59.

[17] M. H. Rauch, "Improved Data Structures for Fully Dynamic Biconnectivity in Graphs". *Proc. 26th Symp. on Theory of Computing*, 1994, 686–695.

[18] D. D. Sleator, R. E. Tarjan, "A data structure for dynamic trees" *J. Comput. System Sci.* 24, 1983, 362–381.

[19] P. M. Spira and A. Pan, "On Finding and Updating Spanning Trees and Shortest Paths", *SIAM J. Comput.,* 4 (1975), 375–380.