Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation

Monika Rauch Henzinger *
Valerie King †

Abstract

This paper solves a longstanding open problem in dynamic algorithms: We present the first dynamic algorithms that maintain connectivity, 2-edge connectivity, bipartiteness, cycle-equivalence, and approximate minimum spanning trees in polylogarithmic time per operation. 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.

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 $O(p \log^3 n)$ for connectivity and bipartiteness and $O(p \log^4 n)$ for 2-edge connectivity. 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 $O(qk \log^3 n)$. Note that cycle-equivalence is equivalent to the 2-edge witness problem. 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. We sketch a modification to our connectivity algorithm which reduces the update time for this and other algorithms by a factor of $\log\log n$.

*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.

STOC '95, Las Vegas, Nevada, USA

1 Introduction

In many areas of computer science, graph algorithms play an important role: Problems modeled as a 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 dynamic 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.

Previous Work. In recent years a lot of work has been done in 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 dynamic algorithms in undirected 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]; $O(\sqrt{n})$ per update and $O(\log n)$[1] per query for 2-edge connectivity ("Are there 2 edge-disjoint paths between two vertices?") [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, 2-edge 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.

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

In planar graphs dynamic algorithms for minimum spanning forest, connectivity and 2-edge 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 and not suited for implementation [5]. Thus, the following questions were posed as challenging open questions in [4, 5, 17]:

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

(2) Is the constant factor in the 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 dynamic algorithms with polylogarithmic time per operation and applies this technique to the dynamic connectivity, 2-edge connectivity, bipartiteness, 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. 2-edge connectivity in update time $O(\log^4 n)$ and query time $O(\log n)$;
3. bipartiteness in update time $O(\log^3 n)$ and query time $O(1)$;
4. minimum spanning tree of a graph with k different weights in update time $O(k\log^3 n)$;
5. 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.

We also sketch a modification to our connectivity algorithm which saves a factor $\log\log n$ in the update time for this algorithm and its applications.

As an immediate consequence of these results we achieve faster 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. A simple minimum spanning tree algorithm that combined with sparsification achieves an amortized expected update time of $O(\sqrt{n}\log^2 n)$.
3. A dynamic algorithm for maintaining a maximal spanning forest decomposition of order k of a graph in time $O(k\log n)$ per update by keeping k 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].

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 dynamic algorithms is are (Monte-Carlo type) approximation algorithms for minimum cuts [14, 12].

This paper is structured as follows: Section 2 gives the dynamic connectivity algorithm, Section 3 presents the 2-edge connectivity result. Sections 4 sketches the results for k-weight minimum spanning trees, bipartiteness, and minimum spanning trees with arbitrary weights.

2 A Connectivity Algorithm

2.1 A Deletions-Only Algorithm

Definitions and notation: Let $G=(V,E)$ with $|V|=n$ and $|E|=m$. The edges of G are partitioned into l subgraphs $G_1,\ldots,G_l$ such that $G_i=(V,E_i)$, $\cup_i E_i = E$, and for all $i\neq j$, $E_i\cap E_j=\emptyset$. For each G_i, we keep a forest F_i of tree edges such that $F=\cup_{j\leq i}F_j$ is a spanning forest of G_i. A *spanning tree T on level i* is a tree of $\cup_{j\leq i}F_j$. If T contains an edge e and a vertex u, let $T_u\setminus e$ denote the subtree of $T\setminus e$ which contains u.

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. A tree is *smaller* than another tree if its weight is no greater than the other's. We say level i is *below* level $i+1$.

Initially all edges are in G_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 index of a forest such that $e\in F_i$. 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. Assume $w(T_1)\leq w(T_2)$.

- If $w(T_1) \leq \log^2 n$ goto *Case 2.*
- *Sample:* We sample $c \log^2 m$ nontree edges of G_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 add it to F_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 $S = \{$edges connecting T_1 and $T_2\}$.
  - If $|S| > w(T_1)/(c' \log m)$ make one element of S a tree edge in F_i.
  - If $0 < |S| \leq w(T_1)/(c' \log m)$, remove the elements of S from E_i, and insert them into E_{i+1}. Then make one of the newly inserted edges a tree edge in F_{i+1}.
  - If $S = \emptyset$ then if $\cup_{j>i} E_j \neq \emptyset$ then do *Replace*$(e, i+1)$. Else stop.

Theorem 2.1 *$\cup_{j \leq i} F_j$ is a spanning forest of $\cup_{j \leq i} G_j$.*

The proof is straightforward and is omitted here.

2.2 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 this 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 the (unordered) list of nontree edges in level i which are incident to v, stored as a balanced binary tree. Each node in the ET-tree contains the number of nontree edges stored in its subtree.

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 $e \in \cup_{1 \leq j \leq k} F_j$, pointers to each of the four (or three, if an endpoint is a leaf) occurrences associated with its traversal in $\cup_{1 \leq j \leq k} F_j$;
- For each nontree edge, pointers to the two leaves of the *ET-tree* in which it is stored;
- 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 F_i and for each edge a pointer back to its position in the list.

2.3 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 $ET(T)$.
- *sample&test*(T): Randomly select a nontree edge of G, 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*(T,e): Insert the nontree edge e incident to T.
- *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 sketch the implementation details of some of these operations.

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.

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 a tree edge on level i, then for $i \geq j$ do *delete_tree*(e,j) and call *Replace*(u,v,i).

If e is not a tree edge, execute *delete_nontree*(e).

Replace(u,v,i)
if $w(tree(u,i)) \leq w(tree(v,i))$ **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: $0 < |S| < w(T_1)/(2c'\log n)$
Select one $e' \in S$;
for $j > i$ **do** *insert_tree*(e',j);
$T_3 = tree(e', i+1)$;
for every nontree edge $e'' \in S$ **do**
delete_non_tree(e''); *insert_nontree*(T_3, e'').
Case 2.3: $S = \emptyset$
if $\cup_{j>i} E_j \neq \emptyset$ **then** *Replace*$(u,v,i+1)$.

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 $w(T_1) < \log^2 n$. The cost of gathering the nontree edges, i.e., executing *nontree_edges*(T_1) is $O(\log n)$ per nontree edge and the cost of testing each edge is $O(\log n/\log\log n)$ for a total cost of $O(w(T_1)\log n)$. Now there are three possible cases.

Case 2.1: $|S| \geq w(T_1)/(2c'\log m)$. If $w(T_1) < \log^2 n$, we charge the cost of executing *nontree_edges*$(T_1) = O(\log^3 n)$ to the *delete* operation. Otherwise, the probability of this subcase occurring is $(1-1/(c'\log m))^{c\log^2 n} = O(1/n^2)$ 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: $|S| < w(T_1)/(2c'\log m)$ 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 searched and tested. Let m_i be the number of edges ever in level i. Every time an edge is incident to a smaller tree, the weight of the tree to which the edge is incident is halved. Thus, over the course of the algorithm, each edge is incident to a smaller tree at most $\log(2m_i)$ times in a given level. Thus, for all such trees T_1 on level i, $\sum w(T_1) \leq m_i \log(2m_i)$ and the total cost of this case over the course of the algorithm for level i is $O(m_i \log^2 n)$. We show that $\sum_i m_i = O(m)$, giving a total cost of $O(m \log^2 n)$.

Lemma 2.2 *The total number of edges ever in G_i is m/c'^{i-1}.*

The proof is straightforward, by induction on the number of levels, and is omitted in this abstract.

Choosing $c' = 2$ immediately gives a bound on the number of levels.

Corollary 2.3 *There are $\log m = O(\log n)$ levels.*

Corollary 2.4 *The sum over all levels of the total number of edges in each level is $O(m)$.*

2.4 A Fully Dynamic Connectivity Algorithm

Next we also consider insertions. When an edge $\{u,v\}$ is inserted into G, use binary search to find the smallest i such that $tree(u,i) = tree(v,i)$ in time $O(\log n \log\log n)$. If u and v are not connected, add $\{u,v\}$ to G_l and F_l. We say that $\{u,v\}$ is *newly inserted* into G at level i or l, respectively.

To guarantee that the number of levels does not exceed $3\log m$, a *rebuild* of the data structure is executed periodically. A rebuild of level i, for $i > 1$, is done by a *move_edges*(i) operation, which moves all tree and nontree edges in G_j for $j \geq i$ into G_{i-1}. Note that after a rebuild at level i, G_j, for $j \geq i$, contains no edges. Also, the spanning trees on level $i-1$ are the same as the spanning trees on level l before the rebuild, i.e., they span the connected components of G.

It is not hard to see that these new operations preserve the invariant that $\cup_{j\leq i} F_j$ is a spanning forest of $\cup_{j\leq i} G_j$.

An *edge addition* to G_j occurs when an edge is newly inserted into G at level j, or when it is moved from G_{j-1} during a deletion. A level i is rebuilt when the number of nontree edge additions into $\cup_{j\geq i} E_j$ since the last rebuild at level i or lower, reaches $b_i = n^3/2^{i-2}$. (Note that if an edge is added, deleted, and then added again, this edge contributes two to the number of nontree edge additions.)

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.

We first address (2). Assume a rebuild occurs on level i. Then for each level $j \geq i$, let m_j be the number of nontree edge additions to level j since the previous time level j was empty. Following a similar argument to the one in the deletions only case, it is not difficult to see that $\sum_{T_1} w(T_1) = m_j \log 2m_j$, where the sum is taken over all smaller components T_1 created on level j since the previous time level j was empty. Thus, the cost for the gathering and testing of edges on level j since the previous time level j was empty is $O(m_j \log^2 n)$. Now, note that m_j cannot exceed b_j since this would have caused a rebuild on level j, which would have emptied level j. Noting that $\sum_{j\geq i} b_j = O(b_i)$, the cost for all levels $j \geq i$ since the rebuild on level i is $O(b_i \log^2 n)$.

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 G_j, $j \geq i$ into G_{i-1}. Each move costs $\log n$ per edge. A tree edge is added into a given level no more than once because it is never moved up. (If a tree edge is deleted and reinserted on level l we consider this a different edge.) Therefore, the total cost of all inserts for any particular tree edge is $O(\log^2 n)$ which can be charged to the insertion of the edge. Since there are no more than b_i nontree edges in levels i or above, the costs of moving these to level $i-1$ is $O(b_i \log n)$.

Thus the cost incurred in (1) and (2) is $O(b_i \log^2 n)$. We use a potential function argument. Each new insertion into level i contributes $c'' \log^2 n$ tokens toward the bank account of each level $j \leq i$ for a total of $\Theta(\log^3 n)$ tokens.

We show below that a rebuild occurs on level i only if $b_i/2$ edges have been newly inserted into G at levels i or above, since the last time G_i had no nontree edges, i.e., the time of the last rebuild on level i or lower. Thus, when the rebuild on level i occurs, there are $c''(b_i \log^2 n)/2$ tokens available to pay for the costs incurred in (1) and (2).

It remains to show the following:

Lemma 2.5 *A rebuild occurs on level i only if $b_i/2$ edges have been newly inserted into G at levels i or above, since the time of the last rebuild on level i or lower.*

Proof: Immediately after a rebuild on level i or lower, there are no nontree edges on level i. We show that at most $b_i/2$ edges are moved from G_{i-1} to G_i since the last rebuild at level i or lower, if $c' \geq 8$, where c' is the constant in the *Replace* algorithm.

This is by induction on i. For $i = 2$ since a smaller component contains less than $n(n-1)/2$ nontree edges at any given time and since there can be no more than $n-1$ splits of components, there have been at most $n(n-1)^2/2$ edges gathered and tested on level 1 since the last rebuild on level 2. At most a factor of $1/(2c'\log n)$ of the searched edges has been moved to level 2. Thus, at most $n^3/(4c'\log n) < b_2/2$ edges are moved from G_1 to G_2 since the last rebuild of G_2.

For $i > 2$ note that the last rebuild at level i or lower occurred at or after the last rebuild at level $i-1$. Thus, since the last rebuild at level i occurred, there were less than b_{i-1} edge additions to G_{i-1} and less than $b_{i-1}\log 2b_{i-1} < 4b_{i-1}\log n$ edges were gathered and tested on level $i-1$. At most a factor of $1/(2c'\log n)$ of these edges has been moved to level i. Thus, less than $2b_{i-1}/c' = b_i/2$ edges are moved from G_{i-1} to G_i since the last rebuild at level i or lower. ∎

2.5 Reducing the Cost of Sampling

In this section, we sketch a modification to the data structure which reduces the cost of sampling a single edge to $O(\log n/\log\log n)$ and the cost of *nontree_edges* to $O(1)$ per edge returned. This results in a faster overall update time, of $O(\log^3 n/\log n\log n)$.

The idea is to keep nontree edges incident to each *ET-tree* in a separate $(\log n)$-ary balanced tree, with one nontree edge per leaf. By randomly accessing each branch, we can select a nontree edge in time proportional to the height of the tree, $O(\log n/\log\log n)$.

For each *ET-tree*, we keep a *nontree edge list* $N(T)$ of all nontree edges in level i incident to nodes in T. Those with two endpoints in T appear twice in the list. The list is organized as follows: Let $x_1,...,x_n$ be the list of nodes in T ordered by the appearance of their active occurrence in $ET(T)$. Let $l(x_j)$ consist of the set of nontree edges (on level i) incident to x_j. Then $N(T)$ consists of the edges in $l(x_1)$, if any, followed by the edges $l(x_2)$, etc. The list is stored in a balanced search tree of degree $\log n$.

The active occurrence for a vertex x is said to be *nonempty* if $l(x) \neq \emptyset$. There is a pointer from each nonempty active occurrence of a node x_j to the start of $l(x_j)$. Also, each internal node of the ET(T)-tree contains a bit indicating if a nonempty active occurrence is contained in its subtree. Given any occurrence, these bits can be used to find the nearest nonempty active occurrence preceding it and succeeding it, in $ET(T)$.

The procedures for *delete_tree* and *insert_tree* are modified so that the nontree edges in the *NT-tree* are reordered when $ET(T)$ is changed. For example, suppose an interval in $ET(T)$ which is delimited by occurrences o_a and o_b is removed. Let $o_{a'}$ and $o_{b'}$ be the nearest nonempty active occurrences succeeding a, and preceding b, respectively. Then the interval in NT from $l(a')$ to the end of $l(b')$ is removed from $NT(T)$. *Delete_nontree*, *insert_nontree*, *nontree_edges* are also slightly modified to accomodate the new data structure.

Using this data structure the asympototic running time of *delete_nontree*, *insert_nontree*, *delete_tree*, and *insert_tree* is $O(\log^2 n/\log\log n)$, while *move_edges* takes $O(\log^2 n/\log\log n)$ per edge moved. The running time of *nontree_edges* is reduced to $O(1)$ per edge returned and the cost of *sample&test* is reduced to $O(\log n/\log\log n)$. (Testing remains $O(\log n/\log\log n)$.) Since these operations are the "bottlenecks" in our previous implementation, this reduces the amortized expected time per operation to $O(\log^3 n/\log\log n)$.

3 A 2-Edge Connectivity Algorithm

Let F be a spanning forest of G and let F' be a spanning forest of $G\setminus F$. Two nodes are 2-edge connected in G iff they are 2-edge connected in $F\cup F'$ [15]. Thus, it suffices to test 2-edge connectedness in $F\cup F'$. Since F and F' can be maintained with two dynamic connectivity data structures, we restrict our description to updates that modify $F\cup F'$. To shorten our description we give an algorithm with amortized expected time $O(\log^6 n)$. However, this algorithm can be improved to take time $O(\log^4 n)$.

3.1 A Deletions-only Algorithm

Definitions and notation: The edges of $F\cup F'$ are partitioned into l subgraphs $G_1,\ldots,G_l$. For each G_i, we keep a forest F_i and a forest F_i' of tree edges such that $F = \cup_j F_j$ and $F' = \cup_j F_j'$. Edges of F are called *tree* edges and the tree path between u and v is denoted by $\pi(u,v)$. If T is the tree of $\cup_{j\le i}F_j$ containing an edge e and a node u, let $T_u \setminus e$ denote the subtree of $T\setminus e$ containing u.

A nontree edge $\{u,v\}$ *covers* a tree edge e iff e lies on the tree path between u and v. A *bridge* is an edge of F that is not covered by an edge of F'. Two nodes u and v are 2-edge connected iff all edges on $\pi(u,v)$ are covered [8].

We maintain the following *invariants*: (1) For $i < l$, every edge of F_i is covered by an edge of F_i', and it is not covered by an edge of F_k' for $k < i$. (2) Every edge of F_l is a bridge.

To initialize the data structures: Initially put all bridges of $F\cup F'$ into G_l and all remaining edges into G_1. This guarantees that the invariants hold.

To answer the query: "Are x and y 2-edge connected?": We store F in a dynamic tree data structure [18] and give all edges of F_l cost 0 and all remaining edges cost 1. To test if x and y are 2-edge connected, check if the minimum cost on $\pi(x,y)$ in the dynamic tree is 0. This test takes time $O(\log n)$.

To update the data structure after a deletion of edge e={u,v}: Let i be the index of the graph such that $e \in G_i$. Remove e from G_i and call *Delete*(e,i).

Delete (e,i)

Case A: $e \in F$: Search for a replacement edge e' in F' for e. If e' does not exist, then e is a bridge. Remove e from G_i. Otherwise, make e an edge of F' and e' an edge of F and continue as in Case B.

Case B: $e \in F'$: Search for a replacement edge e' in $G\setminus F$ for e. If e' exists, make e' an edge of F' and stop. If no replacement edge for e exists, call *Test_Path(u,v,i)*. It determines all edges in F_i which have become bridges in $\cup_{j\le i}G_j$ by the removal of $e=\{u,v\}$, moves them to level $i+1$, and recurses. Note that all newly created bridges in F_i must lie on $\pi(u,v)$.

Test_Path (u,v,i)

Let T denote the tree of $\cup_{j\le i}F_j$ containing u and v. Initially all edges on $\pi(u,v)$ are uncovered. Let e_1 =

$\{x,y\}$ be the middle edge of $\pi(u,v)$ and wlog let $w(T_u \setminus e_1) \leq w(T_v \setminus e_1)$.

- *Sample:* Sample $2c\log^2 m$ edges of F_i' incident to nodes of $T_u \setminus e_1$ for some appropriate constant c. An edge with both endpoints in $T_u \setminus e_1$ is picked with probability $2/w(T_u \setminus e_1)$ and an edge with one endpoint in $T_u \setminus e_1$ is picked with probability $1/w(T_u \setminus e_1)$.
  Next determine the first edge $e' = \{x', y'\}$ of F_i on $\pi(u,y)$ that is not covered by a sampled edge. Wlog let $x' \in \pi(u,y')$.
- *Case: e' exists and $w(T_u \setminus e') \leq w(T_u \setminus e_1)/2$:* Goto *Sample* with e' as e.
- *Case: e' exists and $w(T_u \setminus e') > w(T_u \setminus e_1)/2$:*
  - Search all edges incident to $T_u \setminus e'$ and determine $S = \{$edges of F' connecting $T_u \setminus e'$ and $T_v \setminus e'\}$.
  - If $0 < |S| \leq w(T_u \setminus e')/(c' \log m)$, remove e' and the elements of S from G_i, and insert them into G_{i+1}.
  - Determine all edges of $T_u \setminus e'$ that are not covered by an edge of the new forest F_i' and move them to G_{i+1}.
- If $y \neq v$ then $Tree\_Path(y,v,i)$.

The proof of the following is omitted in this extended abstract:

Theorem 3.1 *For $i < l$, every edge of F_i is covered by an edge of F_i' and it is not covered by an edge of F_k' for $k < i$.*

3.2 A Dynamic 2-Edge Connectivity Algorithm

If an edge $\{u,v\}$ is added to G, it is inserted at the maximum level of all tree edges on $\pi(u,v)$, i.e. into the highest-level graph in which all tree edges on $\pi(u,v)$ are covered. This guarantees that the invariant of Theorem 3.1 is maintained. For each level i if the number of nontree edges added into G_j for $j \geq i$ reaches b_i, level i is rebuilt by adding all edges to level $i-1$. This preserves the invariant as well.

The Analysis of the Running Time. We keep $F \cup F'$ in an Euler Tour data structure and for each i $\cup_{j \leq i} F$ in a dynamic tree data structure. The latter allows us to find the midpoint in a path, cover F with a nontree edge, and find the uncovered edge on level i closest to x in time $O(\log n)$. It is straightforward to implement the algorithm using these data structures. As in the case of connectivity there are $O(\log n)$ levels.

To analyze procedure $Test\_Path$ note that the procedure calls itself $O(\log n)$ times since each call halves the length of the path that is tested. During each recursive call the sampling step is executed $O(\log n)$ times, since every call halves the weight of the smaller component. As in the connectivity case, the cost for searching all edges incident to the smaller component if sampling is not successful can be charged to either the low-probability event that sampling fails or to the edges that are copied to the next level. Thus, the amortized time for $Test\_Path$ is $O(\log^5 n)$.

Looking for a replacement edge takes amortized time $O(\log^3 n)$ and if successful we can stop. If no replacement edge exists, we determine in $Test\_Path$ all edges that are no longer covered at level i and recurse. The amortized time for one call to $Test\_Path$ is $O(\log^5 n)$, which gives an amortized time for a *Delete* of $O(\log^6 n)$. Insertions are payed for using the same charging scheme as in the case of connectivity. The algorithm can be speeded up to $O(\log^4 n)$ by replacing recursive calls with special-case procedures.

4 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.

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 minimum 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. Now, search the path from u to v in 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.

We note that if a tree is stored as a dynamic tree, the midpoint of a path in the tree may be found in $O(\log n)$ time. The amortized update time of this algorithm is $O(k \log^3 n)$. The analysis and proof of correctness are omitted here.

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 $O(\log^3 n (\log U)/\epsilon)$.

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.

Note that an edge weight is only changed once per edge. The amortized update time is $O(\log^3 n)$. We omit the analysis and proof of correctness here.

The minimum spanning tree problem. Here we give an alternative algorithm to do MST updates in $O(\sqrt{n}\log^2 n)$ time, using a simple reduction to the k-weight MST problem.

Let m be the number of edges currently in the graph G. A weight is in block i if it ranks between the $(i-1)\sqrt{m}\log n$ and $i\sqrt{m}\log n$ smallest weights. Now we use the $\sqrt{m}/\log n$-weight MST solution to maintain a minimum spanning tree where all edgeweights in the same block are identified. The only change is as follows: when an edge in F is supposed to be replaced by an edge from block i, check every edge in block i to find the one with minimum weight which is also a suitable replacement. This edge is added to F.

As edges are inserted and deleted, the blocks must be adjusted, which can take $O(\sqrt{m}\log^2 n)$ time. The amortized running time per update is $O(\sqrt{m}\log^2 n)$. Combining this with sparsification [4] yields an update time of $O(\sqrt{n}\log^2 n)$.

5 Acknowledgements

We want to thank Phil Klein for suggesting the use of $\log n$-ary tree. We are also 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", to appear in *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", submitted.

[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.