SRC Technical Note
1997 - 004a
June 27, 1997

Fully Dynamic 2-Edge Connectivity Algorithm in Polylogarithmic Time per Operation

Monika Rauch Henzinger and Valerie King

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301
http://www.research.digital.com/SRC/

Abstract

This paper presents the first dynamic algorithm that maintains 2-edge connectivity in polylogarithmic time per operation. The algorithm is a Las-Vegas type randomized algorithm.

The expected time for $p = \Omega(m_0 + n)$ insertions or deletions of edges is $O(p \log^5 n)$, where m_0 is the number of edges in the initial graph with n nodes. The worst-case time for a query is $O(\log n)$. If only deletions are allowed then the cost for p updates is $O(p \log^4 n)$ expected time.

1 Introduction

We consider the problem of maintaining 2-edge connectivity during an arbitrary sequence of edge insertions and deletion. Two nodes are *2-edge connected* iff there are two edge-disjoint paths between them. Given an n-vertex graph G, the *fully dynamic 2-edge connectivity problem* is to maintain a data structure under an arbitrary sequence of the following update operations:

insert(u,v): Add the edge $\{u, v\}$ to G.

delete(u,v): Remove the edge $\{u, v\}$ from G.

query(u,v): Return true iff u and v are 2-edge connected in G.

In 1991 [5], Fredrickson introduced a data structure known as *topology trees* for the fully dynamic 2-edge connectivity problem with a worst case cost of $O(\sqrt{m})$ per update, where m is the number of edges in the graph at the time of the update. His data structure permitted 2-edge connectivity queries to be answered in $O(\log n)$ time. In 1992, Eppstein et. al. [1, 2] improved the update time to $O(\sqrt{n})$ using the *sparsification technique*. If only edge insertions are allowed, the Westbrook-Tarjan data structure [11] maintains the 2-edge connectivity in time $O(\alpha(m, n))$ per insertion or query. If only edge deletions are allowed ("deletions-only"), then no algorithm faster than the $\Omega(\sqrt{n})$ fully dynamic algorithm was known.

Using randomization, we give the first polylogarithmic-time algorithm for the problem: we present a fully dynamic 2-edge connectivity problem in amortized expected time $O(\log^5 n)$ per update and $O(\log n)$ worst case time per 2-edge connectivity query. If the problem is restricted to edge deletions only, our algorithm runs in amortized time $O(\log^4 n)$. [1]

Let T be a spanning forest of the graph. An edge e of T is *covered* iff there exists a nontree edge $\{x, y\}$ such that e lies on the tree path between x and y.

Two nodes u and v are 2-edge connected iff all edges on the tree path between them are covered. This leads to a first naive approach for maintaining 2-edge connectivity dynamically: Keep for each edge e a "coverage count", i.e., the number on nontree edges covering e. If T is stored in a dynamic tree data structure with the coverage count as cost [9], then after the insertion and deletion of a nontree edge all coverage counts can be updated in time $O(\log n)$. However when a tree edge is deleted and a nontree edge becomes a tree edge, $\Omega(n)$ coverage counts can change and it is not known how to update the dynamic tree efficiently.

We avoid this problem as follows. (1) When a tree edge e is deleted, we first *swap* it with a nontree edge e' such that the cycle induced by e' in T contains e. This means e' becomes a tree edge and e becomes a nontree edge. Then we delete the nontree edge e. (2) We do not keep a coverage count, but remember for each tree edge simply whether it is covered or not, i.e. a *coverage bit*. This requires a more complicated

[1] A preliminary version of this result appeared in [6].

routine for deleting nontree edges than when keeping coverage counts. However it has the advantage that the coverage bit of tree edges are not modified during a swap. Thus we reduced the problem of arbitrary edge deletions to the problem of maintaining the coverage bit. We show below how to solve the latter problem.

2 A Deletions-only 2-Edge Connectivity Algorithm

Let F be a spanning forest of G. We give an algorithm with amortized expected time $O(\log^4 n)$.

2.1 A Deletions-only Algorithm

Definitions and notation: Edges of F are called *tree* edges and the tree path between u and v is denoted by $\pi(u, v)$. 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 nontree edge of G. Two nodes u and v are 2-edge connected iff all edges on $\pi(u, v)$ are covered [5].

Throughout the algorithm, the nontree edges of G are partitioned into levels $E_1, \ldots, E_l$, $l = \lceil 2 \log n \rceil$. Let F_i denote a forest of subtrees of F which are 2-edge connected in $G_i = (V, (\cup_{j \le i} E_j) \cup F)$. Then $F_i \subseteq F_{i+1}$ and an edge is a bridge in G iff it is in $F \setminus F_l$. The *level of a tree edge e* is defined to be the smallest i such that e is covered in G_i.

If T is the tree of F_j containing an edge e and a node u, let $T_u \setminus e$ denote the subtree of $T \setminus e$ containing u. Let the *weight* $w(T)$ of a spanning tree be two times the number of nontree edges with both endpoints in T plus the number of nontree edges with exactly one endpoint in T. The *size* $s(T)$ of a spanning tree is the number of nodes in it.

We maintain the following data structures:

- F is stored in a dynamic tree data structure $D(F)$ whose edges are labeled only while processing a deletion [9].
- For each level i, F is stored in a dynamic tree data structure $B(i)$ in which each edge of F_i has cost 1 and all others have cost 0.
- For each level i, the trees of F_i are stored in an ET-tree in which all nontree edges of E_i are stored, referred to as the ET-trees of level i. The weight of an ET-tree is the number of nontree edge stored there. (See appendix for a full description of ET-trees.)
- For G we keep a dynamic minimum spanning tree data structure $D(MST)$ in which edges are weighted by their level number. The edges in the initial spanning forest F are weighted 0. When an edge is added to F, its weight is decreased to 0. An efficient data structure for maintaining a minimum spanning tree with a small number of weights can be found in [6, 7].

In addition, we keep for each nontree edge a pointer to where it is stored and for each tree edge, its level number and a pointer to its location in each data structure in which it appears. We keep the following invariant.

Invariant:

1. In $B(i)$ an edge of F has cost 1 iff it is in F_i and otherwise it has cost 0.

To initialize the data structures: Initially, put all nontree edges into E_1. The remaining E_j, $j \neq i$ are empty. Compute the 2-edge connected components of G. Let F_1 be the 2-edge connected subforest of F. Then $F_1 = F_2 = ... = F_l$. Construct the data structures accordingly.

To answer the query: "Are u and v 2-edge connected?": To test if u and v are 2-edge connected, output "yes" iff every edge on the path between u and v has weight greater than 0 in $B(l)$. This test takes time $O(\log n)$.

To update the data structure after a deletion of edge $e = \{u, v\}$*:*
Case A: $e \in F$: If $e \notin F_l$ (then e is a bridge), remove e from all data structures representing F. Otherwise, let i be the level of e. Use D(MST) to find a replacement edge e' in E_i for e, make e a nontree edge of E_i and e' an edge of F by calling $swap(e, e')$ and continue as in Case B.

Case B: $e \in E_i$:

The **Delete_Nontree**(e, i) algorithm keeps a list L of edge disjoint paths of F containing possible bridges. Initially L is empty.

Delete_Nontree (e, i)

1. Delete e from the ET-tree in which it is stored if it is present.
2. Add $\pi(u, v)$ to L.
3. *while* $L \neq \emptyset$, remove any path $\pi(a, b)$ from L and do **Test_Path(**a, b, i**)**.
4. *if* $\pi(u, v)$ was not covered on level i and $i < l$, do **Delete_Nontree**$(e, i + 1)$.

The procedure **Test_Path(**u, v, i**)** either determines that all edges in $\pi(u, v)$ are covered by a sequence of random samples; or finds one tree edge f in level i which is suspected of being "sparsely covered" by edges in E_i. If indeed, f is sparsely covered, the algorithm removes f from F_i and moves to E_{i+1} those nontree edges which cross the cut induced by f's removal. The edges in paths of F covered by these nontree edges are in turn marked as possible bridges. Thus the tree in F_i containing f is split into two subtrees T_1, T_2, and the data structures representing F_i are modified accordingly. With very low probability, f is not sparsely covered. In this case f is not removed from F_i. In either case, the smaller of T_1, T_2 is searched exhaustively to determine which of its edges have become bridges in G_i. All paths in L contained in the searched tree are removed from L; the one path containing f is replaced by its subpath in the lighter component.

Test_Path (u, v, i)

Let T denote the tree of F_i containing u and v.

1. $i_u = 0$ and $i_v = 0$;
2. Repeat until i_u and i_v are both greater than $\lg w(T) - 1$ or the algorithm stops.
   (a) Find the furthest edge e_u from u on $\pi(u, v)$ such that $w(T_u \setminus e_u) \leq 2^{i_u}$. If this cut was previously examined, increment i_u and repeat.
   Find also the closest edge e'_u to u on $\pi(u, v)$ such that $w(T_u \setminus e'_u) > 2^{i_u - 1}$.
   (b) If $i_u \leq \lg w(T) - 1$ then if **Sample(**u, v, e'_u, e_u, i **)** returns **false**, **stop**.
   (c) Find the furthest edge e_v from v on $\pi(u, v)$ such that $w(T_v \setminus e_v) \leq 2^{i_v}$. If this cut was previously examined, increment i_v and repeat.
   Find also the closest edge e'_v to v on $\pi(u, v)$ such that $w(T_v \setminus e'_v) > 2^{i_v - 1}$.

(d) If $i_v \leq \lg w(T) - 1$ then if **Sample**(v, u, e'_v, e_v, i) returns **false**, **stop**.

3. *if* $i_u > \lg w(T) - 1$ and $i_v > \lg w(T) - 1$ *then* $\{\pi(u, v)$ *is covered.*$\}$ Remove $\pi(u, v)$ from L.

Sample (z, w, e'_z, e_z, i) requires that the path connecting e'_z and e_z (including e'_z and e_z) is in F_i and that e'_z and e_z belong to $\pi(z, w)$. Let c and c' be constants to be determined later. Let x and y be the endpoints of e'_z and e_z which are furthest from each other such that x is closer to z than y. Let T denote the tree of F_i containing x and y.

Sample (z, w, e'_z, e_z, i)**:**

1. *Choose a sample set X:* If $w(T_x \setminus e_z) < c \log^2 n$ then let X be the set of all nontree edges incident to $T_x \setminus e_z$. Otherwise the set X is built by sampling $c \log^2 n$ times from the set of edges of E_i incident to nodes of $T_z \setminus e_z$. Each edge with endpoints in $T_x \setminus e_z$ is picked with probability between $1/w(T_x \setminus e_z)$ and $2/w(T_x \setminus e_z)$
2. Test if all tree edges in level i on the path $\pi(x, y)$ are covered by edges in X by performing $test\_cover(\pi(x, y), i-1, X)$. If so, increment i_z and return **true**.
3. Else let $f = \{x', y'\}$ be the uncovered such edge nearest to x.
   (a) Construct $S = \{$edges of E_i connecting $T_x \setminus f$ and $T_y \setminus f\}$: For each edge e in E_i incident to $T_x \setminus f$, do $test(e, f, i)$ to determine if e connects $T_x \setminus f$ and $T_y \setminus f$. If so, add e to S.
   (b) If $0 \leq |S| \leq w(T_x \setminus f)/(15c' \log n)$ then $\{$*f is sparsely covered* $\}$,
      i. Do $remove(f, i)$ to remove f from F_i;
      ii. For each edge $e = \{a, b\}$ in S:
         A. Do $move\_up(e, i)$ to move e from E_i to E_{i+1};
         B. Add suitable subpaths of the path $\pi(a, b)$ to L by calling $addL(a, b)$.
   (c) If $s(T_x \setminus f) > s(T_y \setminus f)$ then let T' be $T_y \setminus f$ and do $addL(z, x')$;
   else set T' to $T_x \setminus f$ and do $addL(y', w)$.
   Do $check\_bridges(i, T')$ to find all edges of T' which are no longer 2-edge connected in G_i and remove all paths in L which are contained in T' from L.
   (d) Return **false**.

2.2 Proof of Correctness

We show first that if $i_u > \lg w(T) - 1$ and $i_v > \lg w(T) - 1$ then indeed all edges of $\pi(u, v)$ are covered. This implies the correctness of **Test_Path**.

Lemma 2.1 *If in* **Test_Path** i_u *and* i_v *are greater than* $\lg w(T) - 1$ *then all edges on* $\pi(u, v)$ *are covered.*

Proof: We have to show that all edges on $\pi(u, v)$ have been covered. Let e_v be the furthest edge from v on $\pi(v, u)$ such that $w(T_v \setminus e_v) \leq w(T)/2$. Let e_u be the furthest edge from u on $\pi(u, v)$ such that $w(T_u \setminus e_u) \leq w(T)/2$ and let e'_u be its incident edge closer to v. Note that $w(T_u \setminus e'_u) > w(T)/2$. Thus $w(T_v \setminus e'_u) \leq w(T)/2$, i.e., either $e'_u = e_v$ or e_v is further from v than e'_u. Thus, the longest path tested "for u" and the longest path tested "for v" either touch or overlap. ∎

We next show that all nontree edges are contained in $\cup_{i \le l} E_i$, i.e., when $Test\_Path(u, v, l)$ is called, and Step 3 of Sample is executed, no nontree edge is inserted into E_{l+1}. This fact implies that the trees of F_l span the 2-edge connected components of G.

Lemma 2.2 *With probability at least*[2] $1 - 1/n^2$*, when Step 3 in* **Sample** *is executed, then* $|S| \le w(T_z \setminus f)/(15c' \log n)$.

Proof: For each edge $e' \in \pi(x, y)$ note that $w(T_x \setminus e_z)/2 \le w(T_y \setminus e')$. Thus, when we sample from $T_x \setminus e_z$, we are sampling from $T_x \setminus e'$ with probability at least 1/2. Then each nontree edge is sampled with probability at least $1/(2w(T_x \setminus e'))$. Let us test the hypothesis that $|S| \le w(T_x \setminus e')/(15c' \log n)$. The error probability, i.e., the probability that $|S| > w(T_x \setminus e')/(15c' \log n)$, but no edge of S is selected is

$$(1 - 1/(30c' \log n))^{c \log^2 n} = O(1/n^5)$$

for $c \ge 150c'$.

Thus the probability that $|S| > w(T_x \setminus e')/(15c' \log n)$ for any of the at most n edges on $\pi(x, y)$ is at most $1/n^4$, implying that that the probability is at most $1/n^2$ that it happens at any deletion. ∎

Thus with high probability step (3a) of sample is carried out only once for a subtree before it is split off from its tree. In the following we denote $T_x \setminus f$ by T_1. Note that $w(T_1) \le w(T)/2$. Let m_i be the number of edges ever in E_i.

Lemma 2.3 *For all smaller trees T_1 on level i,* $\sum w(T_1) \le 15 m_i \log n$.

Proof: We use the "bankers view" of amortization: Every edge of E_i receives a coin whenever it is incident to the smaller tree T_1. We show that the maximum number of coins accumulated by the edges of E_i is $15 m_i \log n$.

Each edge of E_i has two accounts, a *start-up account* and a *regular account*. Whenever an edge e of E_i is moved to level $i > 1$, the regular account balance of the two edges on level i with maximum regular account balance is set to 0 and all their coins are paid into e's start-up account. Whenever an edge of E_i is incident to the smaller tree T_1 in a split of T, one coin is added to its regular account.

We show by induction on the steps of the algorithm that a start-up account contains at most $10 \log n$ coins and a regular account contains at most $5 \log n$ coins. The claim obviously holds at the beginning of the algorithm. Consider step $k + 1$. If it moves an edge to level i, then by induction the maximum regular account balance is at most $5 \log n$ and, thus, the start-up account balance of the new edge is at most $10 \log n$.

Consider next the case that step $k + 1$ splits tree T_1 off T and charges one coin to each edge e of E_i incident to T. Let w_0 be the weight of T when T was created. We show that if there exists an edge e such that e's regular account balance was not reset since the creation of T, then $w(T_1) \le 3w_0/4$. This implies that at most $2 \log_{4/3} n < 5 \log n$ splits can have charged to e after e's last reset. The lemma follows.

Edges incident to T at its creation are reset before edges added to level i later on. Since e was not reset, at most $w_0/2$ many inserts into level i can have occurred since the creation of T_0. Thus, immediately before the split, $w(T) \le 3w_0/2$. Since $w(T_1) \le w(T)/2$, the claim follows. ∎

[2] The probability can be increased for $1 - 1/n^d$ for any constant d by increasing the number of sampled edges by a constant factor

Lemma 2.4 *For any i, $m_i \le m/c'^{i-1}$.*

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, $\sum w(T_1)/(15c' \log n)$ edges are added to E_i. By Lemma 2.3, $\sum w(T_1) \le 15m_{i-1} \log n$. This implies that the total number of edges in E_i is no greater than m/c'^{i-1}. ▮

Choosing $c' = 2$ gives the following corollary.

Corollary 2.5 *All nontree edges of G are contained in some E_i for $i \le l$, i.e. E_{l+1} is empty.*

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

Corollary 2.6 $\sum_i m_i = O(m)$.

Finally we show that the invariant is maintained.

Theorem 2.7 *In $B(i)$ a tree edge has cost 1 iff it is in F_i and otherwise has cost 0.*

Proof: The correctness of the invariant depends on the fact that two nodes are 2-edge connected iff they are joined by a path of spanning tree edges which are covered by nontree edges.

When a tree edge e is swapped, if the tree edge is not a bridge, then let i be the minimum index such that e is contained in F_i. Since the endpoints of e are 2-edge connected in F_i, the D(MST) returns a replacement edge e' in E_i.

For F_j, $j < i$, the swap causes no change to the coverage, as e' is not contained in any E_j, $j > i$, and there is no change to the structure of F_j as the two subtrees of F joined by e remain in separate components of F_j.

For levels $j \ge i$, each G_j contains the fundamental cycle formed by e' with edges of F. Hence we observe that when e is swapped with its replacement edge e', e' is covered and all other tree edges' coverage remains unchanged.

When a nontree edge e' is deleted, we observe that the only path in which coverage may change is the tree path between e''s endpoints. Our deletions algorithm either finds edges covering the path or removes from the appropriate F_i those edges in the path which are no longer covered. ▮

2.3 Details of the Implementation

We describe the basic operations available on the two types of data structures we use: dynamic trees, ET-trees, and the k-weight MST dynamic data structure.

For dynamic trees, we can do the following, each in $O(\log n)$ time:

- $link(e)$: links two trees with a new tree edge e.
- $cut(e)$: removes e from a tree, splitting it into two.
- $increment(a, b)$: adds 1 to the cost of each edge on $\pi(a, b)$.
- $decrement(a, b)$: subtracts 1 from the cost of each edge on $\pi(a, b)$.

- $add(p, x)$: adds x to each edge on path p.
- $subtract(p, x)$: subtracts x from each edge on path p
- $min(P)$: returns the first minimal cost edge on the path P.
- $max(P)$: returns the first maximal cost edge on the path P.
- $label(e)$: returns the cost of edge e.
- $midpoint(u, v)$: returns the middle edge of $\pi(u, v)$. This is not a standard dynamic tree operation, but can be easily implemented by storing, in each node of the balanced search tree used to implement dynamic tree, the number of leaves in its subtree.

  In addition, in time $O(\log^2 n)$, we can do:
- $pathwt(u, v, wt)$: returns the furthest edge e from u on $\pi(u, v)$ such that $w(T_u \setminus e) \leq wt$. This can be implemented in $O(\log^2 n)$ time by performing a binary search on $\pi(u, v)$ using no more than $\lg n$ applications of *midpoint* and tests comparing $w(T_u \setminus e)$ to wt.

We use ET-trees to represent dynamically changing trees. An *ET-sequence* is a sequence generated from a tree by listing each vertex each time it is encountered ("an occurrence of the vertex") as a tree is searched depth-first. Each ET-sequence is stored in a balanced binary tree. For each vertex we choose one designated occurrence of the vertex in the sequence and call it the *active occurrence*. Each nontree edge is stored twice, with the active occurrence of each of its two endpoints. At each internal node of the ET-tree, we keep the number of nontree edges incident to active occurrences in its subtree and the number of active occurrences contained in the subtree. The ET-tree imposes an ordering on the edges of the tree being represented, and on the nontree edges as well. Using ET-trees, we can do the following, each in time $O(\log n)$:

- $link(e)$: inserts a tree edge e which links up the two ET-trees containing its endpoints.
- $cut(e)$: removes a tree edge, splitting an ET-tree into two.
- $insert\_nontree(e)$: inserts nontree edge e into an ET-tree which contains an endpoint of the edge.
- $delete\_nontree(e)$: remove a nontree edge e from the ET-tree in which it is stored.
- $find\_nontree(i, T)$: return the i^{th} nontree edge stored in the ET-tree T.
- $test(a, b, i)$: tests if nodes a and b are in the same tree of F_i.

The $D(MST)$ maintains a minimum spanning tree while allowing two operations:

- $insert(e, w)$: inserts an edge e of weight w and returns the replaced edge, if an edge is replaced.
- $delete(e)$: deletes an edge e from the graph and returns the edge which replaces e in the MST, if e is replaced.

For each edge which is deleted or inserted or whose weight is changed by a combination of these operations, the amortized expected update cost of the fully dynamic MST algorithm when there are k weights is $O(k \log^2 n)$, so that the cost per deletion is $O(\log^3 n)$ (see [6, 7]).

To maintain L we use a dynamic tree $D(F)$ representing F. All edges in paths of L have labels naming the path it is in.

- $addL(a,b)$: adds to L the maximal subpaths of $\pi(a,b)$ which are edge-disjoint from the other elements of L. It is implemented by repeatedly applying min and max to find the start and end of each subpath of $\pi(a,b)$ with cost 0 in $D(F)$. These subpaths are added to L. Each edge in a subpath p is labeled in $D(F)$ with the name x of the subpath by doing $add(p,x)$. The cost of $addL$ is $O(\log n)$ times to number of subpaths added to L.

We describe how to implement the operations specified in the algorithm.

- $swap(e,e')$ swaps a tree edge e with a nontree edge e': For each dynamic tree and each ET-tree containing e, do $cut(e)$, $link(e')$,give e' cost 1 in $B(i)$, and give e' weight 0 in $D(MST)$. Each $swap$ operation costs $O(\log n)$ per level or $O(\log^2 n)$ in total.

- $test\_cover(p,j,X)$ covers the path p in $B(j)$ with edges of X and returns the first edge in the path which is not in F_j and is not covered, if there is such an edge. To implement, use $B(j)$. For each edge $\{a,b\}$ in X, do $increment(a,b)$; then do $min(p)$. If the cost of the edge returned by $min(p)$ is 0, return that edge. To restore $B(j)$, reverse the process by decrementing the costs on each path. The cost of $test\_cover$ is $O(|X|\log n)$.

- $remove(f,i)$ removes f from F_i. Run $decrement(f)$ in $B(i)$.
  Do $cut(f)$ on the ET-tree containing f for F_i. Decrement the weight of f in $D(MST)$. The cost is $O(\log^3 n)$.

- $move\_up(e,i)$ moves a nontree edge e from E_i to E_{i+1}. Do $delete\_nontree(e)$ from the ET-tree containing e for F_i and $insert\_nontree(e)$ on the ET-tree for F_{i+1} which contains e's endpoints. Increment the weight of the edge in the connectivity data structure $D(MST)$. The cost is dominated by the cost of the last operation, which is $O(\log^3 n)$.

- $check\_bridges(i,T')$ checks every tree edge in T' to determine if it has become a bridge in G_i.

  1. For each edge $\{a,b\}$ in E_i which is incident to a node in T', do $increment(a,b)$ in $B(i-1)$.
  2. For each (tree) edge e in T', if $label(e)\neq 0$, then remove $label(e)$ from L and do $subtract(label(e),label(e))$.
  3. For each (tree) edge e in T' which has cost 0 in $B(i-1)$ (it is a bridge in G_i) do $remove(e,i)$.
  4. Restore $B(i-1)$ by decrementing the paths which were previously incremented in $check\_bridges$.

  The cost is $O((s(T')+w(T'))\log n)$ plus $O(\log^3 n)$ for each bridge in T'.

2.4 Analysis of Running Time

We show that the amortized cost per edge deletion is $O(\log^4 n)$ if there are $m+n$ deletions.

The cost of initializing data structures is $O(n\log n)$ per level plus $O(m\log n)$, for a total cost of $O(n\log^2 n + m\log n)$.

The cost of Case A is dominated by the cost $O(\log^3 n)$ of finding a replacement edge for e, plus the cost of Case B.

For Case B, the cost of the calls to **Test_Path** dominate the running time.

In the case where the path is completely covered by a sequence of sampled edges, (**Test_Path** runs to completion), there are less than $2\log w(T)$ points in the path during which the nontree edges are sampled. Each point is discovered using *pathwt* in $O(\log^2 n)$ time. At each point, the sampling and testing involve $O(\min(w(T), \log^2 n))$ edges at a cost of $O(\log n)$ per edge, for a total cost, for the whole path, of $O(\log w(T)(\log^2 n + \log n \min(w(T), \log^2 n)) = O(\min(w(T), \log^2 n) \log^2 n)$.

To analyze the number of times **Test_Path** runs to completion, we need the following:

Lemma 2.8 *Let s be the number of times $addL$ is called in one call to* **Delete_Nontree**. *Let t be the number of subpaths added to L during that call to* **Delete_Nontree**. *Then $t \leq 2s - 1$.*

Proof: To see this, let v be the number of connected components of F induced by the labeled edges in $D(F)$. Consider $t + v$. Initially, $v = t = 0$. When two components are disconnected they are not reconnected and may be regarded as the same component for the purpose of this analysis.

If $addL(a, b)$ results in r subpaths added to L then $\pi(a, b)$ connects up $r - 1$ labeled subtrees, reducing v by $r - 2$. Hence each $addL$ adds at most 2 to $t + v$. Since $v \geq 1$, $t \leq 2s - 1$. ∎

Consider a fixed level i. Procedure **Test_Path** is run once for each path removed from L. Hence, the number of calls to **Test_Path** on level i, is no greater than the number q of paths added to L during a call to **Delete_Nontree** (e, i). From the lemma we know that q is at most twice the number of calls to $addL$ on level i minus 1. We show below that the number of calls to $addL$ on level i is bounded by the number of edges moved to level $i + 1$ plus one. Thus, q, and hence the calls to **Test_Path**, is at most twice the number of edges moved to level $i + 1$ plus one. If subsequently **Delete_Nontree**$(e, i + 1)$ is called, then at least one of the calls to **Test_Path** on level i did not run to completion and thus the number of calls that did run to completion is at most twice the number of edges moved to level $i + 1$. Thus the cost of **Test_Path** on all levels running to completion is $O(\log^4 n)$ per deletion and $O(m_i \log^4 n)$ for each level during all deletions, for a total of $O(\log^4 n \sum_i m_i) = O(m \log^4 n)$ during all deletions.

Consider next the case where the path is not completely covered, i.e., **Test_Path** does not run to completion. Let f be the first uncovered edge in step 3. Let $T_1 = T_x \setminus f$. We claim that the cost of all the successful sampling up to the point of the exhaustive search is $O(w(T_1) \log^2 n)$. Let $T^{(0)}, T^{(1)}, \ldots, T^{(p)}$ be the sequence of subtrees in T previously sampled. Now $w(T^{(j)}) \leq w(T^{(j+2)})/2$, and $w(T^{(p)}) \leq w(T_1)$. The successful sampling in $T^{(j)}$ has cost $O(w(T^{(j)}) \log^2 n)$. Thus the total cost of successful sampling is $O(\sum_j w(T^{(j)}) \log^2 n) = O(w(T_1) \log^2 n)$.

When f is indeed sparsely covered, T is split into two trees and an exhaustive search of T' using $check\_bridges$ is carried out, where T' is the smaller in size of $T_x \setminus f$ and $T_y \setminus f$. Then **Test_Path** stops. The cost of $check\_bridges$ is $O((s(T') + w(T')) \log n)$ plus $O(\log^3 n)$ for each new bridge. Note that $w(T_1) \leq w(T)/2$ and therefore, $w(T') \geq w(T_1)$. Thus, the cost of the successful sampling up to the time the split is made plus the cost of $check\_bridges$ is $O(w(T') \log^2 n + s(T') \log n)$ plus $O(\log^3 n)$ for each new bridge.

Now each time a node is in a component of F_i which is exhaustively searched and split, the size of its component is at least halved. Hence, it can be in no more than $\lg n$ T''s. Similarly, the endpoints of an edge can be in no more than $\lg n$ T''s. Hence, $\sum_{T'} s(T') + w(T') \leq (n + m_i) \log n$ so that the total cost of all $check\_path$'s on level i is bounded above by $O((n \log^2 n + m_i \log^3 n)$ plus $O(\log^3)$ for each new bridge. Noting that an edge of F_i can become a bridge only once, this implies a bound of $O(n \log^3 n + m_i \log^3 n)$ per level or $O(n \log^4 n + m \log^3 n)$ over all.

As shown in Lemma 2.1, the probability that during the course of the algorithm, an edge f that is suspected of being sparsely covered is not sparsely covered is no greater than $1/n^2$. Thus this case adds no more than $O((n + m) \log^2 n/n^2) = O(\log^2 n)$ to the expected cost of the algorithm.

We conclude that the total cost over the course of the algorithm for deleting edges in E_i is $O((n + m) \log^4 n)$.

3 A Dynamic 2-Edge Connectivity Algorithm

The basic idea is to insert edges into the last level and *rebuild* levels as necessary. This follows the technique used in the fully dynamic connectivity algorithm of [6].

Let F be a spanning forest of G. We define l, E_i, and F_i as in the deletions-only section. Initially, all nontree edges of G are put into E_1 and the other E_i are empty.

We represent F_i for each level i as labeled subtrees in a dynamic tree data structure representing F. Unlike the deletions-only algorithm, we also keep a compressed version F_i^c of F_i with size proportional to the size of E_i, so that the ET-trees for a level represent F_i^c, rather than F_i. The node set of F_i^c is called V_i^c, its edges are called *superedges*.

Initially and during each rebuild of level i, the compressed forest $F_i^c = (V_i^c, E^c)$ is constructed using E_i. For each edge $\{x, y\}$ in E_i, we mark the path $\{x, y\}$ in F and call the resulting marked subforest F^m. Let V_i^c contain all nodes which are leaves and all nodes which have degree at least three in F^m. A *superedge* $\{x, y\}$ is in F_i^c iff $x, y \in V_i^c$, the path between x and y is in F^m and there are no other nodes in V_i^c which are on the path $\pi(x, y)$.

We note that a component of F_i may contain several components of F_i^c. As the algorithm proceeds, we allow there to be additional superedges in level i which cover paths covered in F_j, $j < i$, and which are not covered by edges in E_i. We describe below which exact invariants hold for F_i^c.

We say that *path $\pi(x, y)$ is represented by* superedge $\{x, y\}$. We say a *node is represented in F_i^c* if it is contained in a path which is represented in F_i^c. The node need not be in V_i^c. The *weight $w(T)$* of a spanning tree T in F_i^c is two times the number of nontree edges with both endpoints in T plus the number of nontree edges with exactly one endpoint in T. The *size $s(T)$* of a spanning tree is the number of nodes of V_i^c in it.

We keep for each level i:

- a dynamic tree data structure $N(i)$ storing F, where each tree edge e which is represented by a superedge of F_i^c is labeled with its name.
- a dynamic tree data structure $N'(i)$ storing F, where each node $v \in V$ which is not in V_i^c, but is represented by a superedge is labeled with the name of the superedge.

  Note that this means that in the dynamic tree data structure $N'(i)$ vertices rather than edges have costs. Operations analogous to those defined for dynamic trees with edge costs can be implemented with the same time bounds [10].
- a dynamic tree data structure $C(i)$ storing F where an edge e has cost $c_i(e)$ if it is represented by $c_i(e)$ superedges in F_j, $j \le i$.
- for each 2-edge connected component of F_i^c, an ET-tree data structure in which all nontree edges of E_i are stored, referred to as the ET-trees of level i.

In additon, we keep a dynamic tree data structure for F, $D(F)$ which is only marked during the course of a deletion, and a fully dynamic minimum spanning tree data structure $D(MST)$. Both are used as in the deletions-only algorithm. We also keep for each nontree edge a pointer to where it is stored, for each tree edge, its level number and a pointer to its location in each data structure in which it appears, for each superedge a pointer to its location in each ET-tree in which it appears, and a list of all nodes in in each V_i^c.

We keep the following invariants.

Invariants:

1. For each edge e in E_i, all edges of F which are covered by e are represented in F_i^c.
2. V_i^c contains every node which is represented in F_i^c by more than one superedge and the endpoints of every edge in E_i. The intermediate nodes in a path that is represented by a superedge cannot be in V_i^c. (Consequently, each edge in F is represented by at most one superedge.)
3. All edges of F which are represented by a superedge in F_i^c must be in F_i.
4. If two nodes of V_i^c are connected in F_i^c, they remain connected in F_i^c until either they are no longer connected in F_i or level j, $j \leq i$ is rebuilt.
5. In $C(i)$, an edge of F has cost $c(e)$ if it is represented by exactly $c(e)$ superedges in F_j^c, $j \leq i$.

3.1 Subroutines

In addition to the basic operations on dynamic trees, ET-trees, and the dynamic minimum spanning tree data structure, in the fully dynamic data structure,

We introduce the following subroutines for operations on compressed forests:

- $removeS(e, i)$: This assumes $e = \{u, v\}$ is a superedge in F_i^c and removes it from F_i^c.
  1. Remove e from the ET-tree representing F_i^c.
  2. For $j \geq i$, decrement $\pi(u, v)$ in $C(j)$.
  3. Remove the name of e from $\pi(u, v)$ in $N(i)$ and $N'(i)$.
- $insertS(e, i)$: This inserts the superedge $e = \{u, v\}$ into F_i^c. It assumes no intermediate node of $\pi(u, v)$ is represented in F_i^c and u and v are in V_i^c.
  1. Insert the tree edge e into the ET-tree representing F_i^c.
  2. For $j \geq i$, increment $\pi(u, v)$ in $C(j)$.
  3. Label $\pi(u, v)$ with its name in $N(i)$ and $N'(i)$.
- $insertV(u, i)$:
  1. If u is represented in F_i^c and $u \notin V_i^c$:
     (a) Let $\{x, y\} = label(u)$;
     (b) Do $removeS(\{x, y\}, i)$; add u to V_i^c; $insertS(\{x, u\}, i)$; $insertS(\{u, y\}, i)$.
  2. Else if $u \notin V_i^c$:
     (a) Create an ET-tree containing only u.
- $connect(e, i)$: Let $e = \{u, v\}$. This routine either inserts a superedge between u and v into F_i^c or, if $\pi(u, v)$ is partially represented by superedges in F_i^c, the whole path is now represented, by connecting up the represented segments by new superedges. It adds u and v to V_i^c if they are not already in V_i^c.
  1. Do $insertV(u, i)$.
  2. If $u = v$ stop.

3. If the first edge of the path $\{u, v\}$ is represented in F_i^c, then find the first edge $\{s, t\}$ in $\pi(u, v)$ which is not represented. Do $connect(\{s, v\}, i)$.
4. Else $\{$*the first edge of the path* $\{u, v\}$ *is not represented*$\}$ do:
   (a) If there is no next node in $\pi(u, v)$ which is represented then do $insertV(v, i)$ and $insertS(\{u, v\}, i)$.
   (b) Else let a be the next node $\pi(u, v)$ which is represented.
      i. Do $insertV(a, i)$.
      ii. Do $insertS(\{u, a\}, i)$.
      iii. Do $connect(\{a, v\}, i)$.

- $insertF(e)$: connects two previously unconnected components of F by inserting a new tree edge e. This operation applies $link(e)$ to each data structure containing F and $insert(e, 0)$ to $D(MST)$. The cost of $insertF(e)$ is dominated by the last operation which has cost $O(\log^3 n)$.
- $deleteF(e, i)$: splits a tree of F into two trees by deleting edge e. This operation involves $cut(e)$ applied to each data structure containing F on levels $j \geq i$ and does $delete(e)$ on $D(MST)$. The cost of $deleteF(e, i)$ is dominated by the last operation which has cost $O(\log^3 n)$.

3.2 Insertions

When edge e is inserted into G then if e connects two unconnected components of F, e is inserted into the data structures representing F using $insertF(e)$. If e is a nontree edge then do $insert\_nontree(e, l)$ to insert e into an ET-tree on level l; $connect(e, l)$; insert e into the $D(MST)$ by calling $insert(e, l)$.

After each operation, we increment I, the number of operations 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 $b_1, ..., b_{l-1}$, where b_1 is the most significant bit, then a rebuild of level i occurs each time b_i bit flips to 1. Note that level l is emptied of nontree edges every time an edge is inserted, and for $i < l$, no more than 2^{l-i-1} operations occur before a rebuild on level i or lower occurs.

3.3 Rebuilding level i:

During a level i rebuild, we remove all nontree edges from E_j $j > i$ and put them into E_i. For each such nontree edge, we increase its weight in $D(MST)$ to i. Then $F_i = F_l$, i.e., it is the 2-edge connected forest of G.

For each level $j \geq i$, for all superedges e in F_j^c do $removeS(e, j)$; also discard the ET-trees. Construct the new compressed graph F_i^c by applying $connect(e, i)$ for each edge of E_i. We construct the compressed graph F_i^c in the same way at the start of the algorithm.

3.4 Deletions

- **Test_Path**(u, v, i) runs on F_i^c rather than F_i which implies that only subtrees of the tree T_u in F_i may be represented in F_i^c. In **Delete_Nontree**(e, i) we may need to add superedges to F_i^c in order to connect portions of the path $\pi(u, v)$ which were (before the deletion) connected in F_i.

  Thus, we add the following to **Delete_Nontree**(e, i):

0. $connect(e, i)$.

- In **Sample**, we reset c' to 4. Consequently, each level receives no more than 1/4 of the edges in the previous level, and we prove a lemma analogous to Lemma 2.4 below.
- $addL(a, b)$ is unchanged. We note that the endpoints of subpaths added to L on a level i are in V_i^c, so that all edges represented by the same superedge are labeled the same.
- $swap(e, e')$: Let i be the level of the tree edge. Let $e = \{s, t\}$.
  1. $delete\_nontree(e', i)$
  2. If there is a superedge $e_j = \{u, v\}$ in F_j^c which represents a path containing e, then do $removeS(e_j, j)$ for all $j \geq i$; do $connect(u, s)$ and $connect(t, v)$.
  3. Do $deleteF(e, i)$ to delete e from F.
  4. Do $insertF(e')$ to insert e' into F.
  5. Do $insert\_nontree(e, i)$.
  6. Do $connect(e, j)$ for all $j \geq i$.
- Sampling is unchanged.
- In $test\_cover$ we use $C(j)$ instead of $B(j)$. In $C(j)$ a tree edge has cost greater than 0 iff it is covered on a level j, $j > i$. The only difference is that costs greater than 0 may also be greater than 1.
- $remove(f, i)$ removes a superedge f from F_i^c. Do $removeS(f, i)$.
- $move\_up(e, i)$ moves a nontree edge e from E_i to E_{i+1}. Do the same operation as in the deletions-only version, followed by $connect(e, i)$.
- $check\_bridges(i, T^c)$ checks every superedge in T^c to determine if it contains a bridge in G_i.
  1. For each superedge $\{a, b\}$ in F_i^c which is incident to a node in T^c, do $increment(a, b)$ in $C(i - 1)$.
  2. For each superedge e in T^c, let e' be any edge represented by e. If $label(e') > 0$, then remove $label(e')$ from L and do $subtract(label(e'), label(e'))$. (Recall that all the edges represented by a superedge have the same label.)
  3. For each superedge e in T^c which represents a path that contains an edge of cost 0 in $C(i - 1)$ (the edge is a bridge in G_i) do $removeS(e, i)$.
  4. Restore $C(i-1)$ by decrementing the paths which were previously incremented in $check\_bridges$.

  The cost is $O((s(T^c) + w(T^c)) \log n)$ plus $O(\log^3 n)$ for each bridge in T^c.

3.5 Queries

To test whether u and v are 2-edge connected: Find the path from u to v in F as stored in $C(l)$ and output "yes" iff $c(e) > 0$ for all e in the path.

3.6 Proof of correctness

We first show:

Lemma 3.1 *For any i, $m_i \leq (5/4)n^2/2^i$.*

Proof: When level i is rebuilt, since no more than $n^2/2^i$ operations have occurred, no more than $n^2/2^i$ new edges are added to E_i and no more than $m/4^i$ edges are already contained in E_i, so that $m_i \leq (5/4)n^2/2^i$.

▮

Corollary 3.2 *All nontree edges of G are contained in some E_i for $i \leq l$, i.e. E_{l+1} is empty.*

Therefore $G_l = G$ and F_l spans the 2-edge connected components of G.
We show next that the invariants hold.

Lemma 3.3 *For each edge e in E_i, all edges of F which are covered by e are represented in F_i^c.*

Proof: Note that if the claim holds before a call to $connect(e, i)$, it will also hold afterwards. Note further that after the call to *connect* all edges on the path in F_i between the endpoints of e are represented by a superedge and the endpoints of e are added to V_i^c if they are not already there.

During an insertion of a nontree edge e, e is inserted into E_l and $connect(e, l)$ is called, ensuring the invariant holds for level l. The other levels are unchanged.

Since *connect* is called for each edge in E_i during a rebuild of level i, it follows that the claim holds right after rebuilding level i.

Consider next a deletion. There are three cases to consider: moving down an edge to E_i, updating whenever a tree edge e is swapped with e', and removing a superedge from F_i^c. In $move\_down(e, i)$, $connect(e, i)$ is called and the claim holds.

When a tree edge $e =$ is swapped with a nontree edge, if e is represented by a superedge $\{a, b\}$, $\{a, b\}$ is removed from F_i^c, e becomes a nontree edge and $connect(e, i)$ is called. The calls to *connect* ensures that the remaining tree edges previously represented by $\{a, b\}$ are again covered and that the path now covered by e is represented by superedges.

We next consider the case when a superedge $e = \{u, v\}$ is removed from F_i^c. This occurs during a **Delete_Nontree**. By the invariants, we know that no nontree edges in E_i which cover an edge in $\pi(u, v)$ have endpoints which are intermediate nodes in $\pi(u, v)$ or which are in subtrees of intermediate nodes in $\pi(u, v)$. Otherwise there would be a node in $\pi(u, v)$ which is either an endpoint of a nontree edge or represented by two superedges and therefore in V_i^c. Thus, each nontree edge covering edges of $\pi(c, d)$ covers all edges of $\pi(c, d)$. We show next that **Test_Path** removes all such edges. The lemma follows.

By connecting u and v in **Delete_Nontree**, we ensure that **Test_Path** runs on tree T_u^c which contains u and v and represents a subtree of T. Since the path $\pi(u, v)$ is represented in T_u^c, every nontree edge which covers a portion of the path $\pi(u, v)$ is incident to the tree T^c, by Invariant (1). Thus, $T_x \setminus f$ and $T_y \setminus f$, where f is the set of edges in $\pi(c, d)$, contain all the edges which might cover $\pi(c, d)$. It follows that the set S connecting $T_x^c \setminus f$ and $T_y^c \setminus f$ is exactly the set of edges crossing the cut between $T_x \setminus f$ and $T_y \setminus f$. When S is removed, the edges in $\pi(c, d)$ are not covered by any nontree edges in E_i.

▮

Lemma 3.4 *V_i^c contains every node which is represented in F_i^c by more than one superedge and the endpoints of every edge in E_i. The intermediate nodes in a path that is represented by a superedge cannot be in V^c. (Consequently, each edge in F is represented by at most one superedge.)*

Proof: Each time a nontree edge is added to F_i^c *connect* puts its endpoints in V_i^c if they are not already there. Each time a superedge is added to F_i^c, if the path covered by the superedge contains a node which is already represented, it is added to V_i^c and the superedges containing that node are split. ▮

Lemma 3.5 *In $C(i)$, an edge of F has cost $c(e)$ if it is represented by exactly $c(e)$ superedges in F_j^c, $j \leq i$.*

Proof: Note that every time an edge is added to or removed from F_j^i, $C(i)$ is updated accordingly. The lemma follows by induction over the steps of the algorithm. ▮

Lemma 3.6 *All edges of F that are represented by a superedge in F_i^c must be in F_i.*

Proof: Superedges are added by a $connect(e, i)$ operation which adds superedges which cover only paths between the endpoints of e. New superedges may also be created by splitting old ones, but these do not change the edges represented in F_i^c. The operation $connect(e, i)$ is called when e is added to E_i during the execution of $swap$, $move\_up(e, i-1)$, or in E_i during a rebuild. In these cases, clearly, the invariant is not violated since e covers the edges represented by the newly created superedges.

The operation $connect(\{u, v\}, i)$ is also called before a call to **Test_Path**(u, v, i). Immediately, **Test_Path** either covers the path $\{u, v\}$ or removes any superedges representing an edge with cost 0 in $C(i-1)$ and not covered by edges in E_i. An edge has cost greater than 1 in $C(i-1)$ only if it is covered by a superedge in F_j^c, $j > i$. If we assume the invariant held true for F_{i-1}^c then we can conclude that all superedges which are not removed cover paths which are in F_j $j \leq i$ which implies they are in F_i.

▮

Lemma 3.7 *If two nodes in V_i^c are connected in F_i^c, they remain connected until either they are no longer connected in F_i or level j, $j \leq i$, is rebuilt.*

Proof: Note that neither an $insertS(e, i)$ nor a $connect(e, i)$ disconnects previously connected nodes in F_i^c, only a $removeS(e, i)$ does. A $removeS(e, i)$ is executed either (a) during a rebuild on level $j \leq i$, (b) when removing a superedge from level i, (c) during a swap, or (d) in $insertV$. In case (b), the path between the endpoints of the superedge is no longer in F_i. In case (c), as shown in the proof of Lemma 3.3, any edges which were represented before are still represented. In case (d), the endpoints of e are immediately reconnected by two $insertS$ operations. Therefore, until level $j \leq i$ is rebuilt, the endpoints must remain disconnected to preserve the previous invariant. ▮

3.7 Analysis of the Running Time.

The cost of initializing data structures is the same as for the deletions-only algorithm, $O(n \log^2 n + m \log n)$.

The routines *insertS(e,i)* and *removeS(e,i)* take $O(\log n)$ time per level or $O(\log^2 n)$ time. Routine *insertV(v,i)* takes time $O(\log^2 n)$. The routine *connect(e,i)* takes $O(\log^2 n)$ time per superedge which is inserted. The number of superedges inserted is proportional to a constant plus the number of components of F_i^c connected up.

Insertions take time $O(\log^3 n)$ plus the cost of $connect$ which is $O(\log^2 n)$ times the number of components of F_i^c connected up. If a tree edge is inserted then $insertF$ is called for a cost of $O(\log^3 n)$.

A rebuild on level i requires a $removeS$ for each superedge on level $j > i$, a $connect$ for each edge in the new E_i and a weight change to the edge in $D(MST)$. We will show the number of superedges in F_i^c is proportional to m_i. Thus the total cost is $O((\log^3 n)m_i)$.

The analysis of **Test_Path** is almost the same as in the deletions-only algorithm, except for the costs of $connect$ which is analyzed below.

As in the deletions-only analysis, the cost of **Test_Path**'s which run to completion is $O(m_i \log^4 n)$ per level and $O(\log^4 n)$ per deletion.

We next analyze the cost when **Test_Path** does not run to completion. It is possible that when we search a component of F_i, we are searching only a portion of the component since that is all that is connected in F_i^c, and we may in fact be searching a portion of the larger component of F_i, rather than the smaller. Yet the size of this portion must be no greater than the size of the smaller component of F_i; therefore the number of times the coverage of a nontree edge is looked at is bounded as in the deletions-only algorithm.

Thus the analysis of the amortized costs of splits charged to a level i between two consecutive rebuilds of levels i or lower is the same as in the deletions-only algorithm except that the maximum number of nodes and superedges on a level i is $O(m_i)$ rather than n. Therefore the cost per level is $O(m_i \log^3 n)$.

The total cost of **Test_Path**'s is thus $O(m_i \log^4 n)$ for level i plus $O(\log^4 n)$ per deletion, plus the cost of $connect$'s.

The cost of $connect$'s on a level is $O(\log^2 n)$ times the number of superedges inserted on this level.

Lemma 3.8 *The total number t of superedges inserted into a level i between two consecutive rebuilds of any levels $j, j' \leq i$ is linear in m_i plus the number of operations since the rebuild.*

Proof: Let s be the number of times $connect$ on level i is called between two consecutive rebuilds of any levels $j, j' \leq i$. Then we show $t \leq 4s - 3$. To see this, let v be the number of connected components of F_i^c. Consider $t + 3v$.

Note that initially, $v = 0$ and that by Invariant 4, when two components are disconnected they are not reconnected and may be regarded as the same component for the purpose of this analysis. If $connect(\pi(a,b), i)$ results in r additional new superedges then $\pi(a,b)$ connects up at least $(r-1)/3$ components of F_i^c, reducing v by $(r-1)/3 - 1$. Hence each $connect$ adds at most 4 to $t + 3v$. Since $v \geq 1$, $t \leq 4s - 3$.

The number of times $connect$ is called on level i is proportional to the number of operations between rebuilds (once per **Delete_Nontree**, once per edge insertions, and three times per $swap$) plus the number of nontree edges moved to level i or $O(m_i)$.

∎

After an edge is inserted into G, it participates at most once per level in a rebuild of a level. Thus, each edge may be charged for the cost of the number of levels times $O(\log^4 n)$ to pay for the amortized costs of **Test_Path**, giving a total amortized cost per insertion of $O(\log^5 n)$, over a sequence of $\Omega(n + m_0)$ operations, where m_0 is the initial number of edges in the graph.

4 Acknowledgements

We would like to thank Madhukar Reddy Korupolu for bringing to our attention errors in the fully dynamic portion of the preliminary version of this algorithm.

References

[1] 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.

[2] 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.

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

[4] G. N. Frederickson, "Data Structures for On-line Updating of Minimum Spanning Trees", *SIAM J. Comput.*, 14 (1985), 781–798.

[5] G. N. Frederickson, "Ambivalent Data Structures for Dynamic 2-edge-connectivity and k smallest spanning trees" *Proc. 32nd Annual IEEE Symposium on Foundation of Comput. Sci.*, 1991, 632–641.

[6] M. R. Henzinger and V. King. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *Proc. 27th ACM Symp. on Theory of Computing*, 1995, 519–527.

[7] M. R. Henzinger and M. Thorup. Improved Sampling with Applications to Dynamic Graph Algorithms. To appear in *Proc. 23rd International Colloquium on Automata, Languages, and Programming (ICALP)*, Springer-Verlag 1996.

[8] 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.

[9] D. D. Sleator, R. E. Tarjan, "A Data Structure for Dynamic Trees" *J. Comput. System Sci.* 24 (1983), 362–381.

[10] R. E. Tarjan, *Data Structures and Network Algorithms* SIAM (1983), 57–70.

[11] J. Westbrook, R. E. Tarjan, "Maintaining Bridge-Connected and Biconnected Components On-Line" *Algorithmica* 7(5) (1992), 433–464.

5 Appendix

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 arbitrar y 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 prece des the other in the ordering in time $O(\log n/b)$.