SRC Technical Note
1997 - 004
June 12, 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.

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 insertions or deletions of edges is $O(p \log^5 n)$ and 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 a 2-edge connectivity during an arbitrary sequence of edge insertions and deletion. Given an n-vertex graph G with edge weights, 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 [10] maintains the minimum spanning forest 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 polynomial-time algorithm for the problem: we present a fully dynamic 2-edge connectivity problem in amortized time $O(\log^5 n)$ per update and $O(\log n)$ per 2-edge connectivity query. If the problem is restricted to deletions-only, our algorithm runs in amortized time $O(\log^4 n)$. A preliminary version of this result appeared in [6].

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 $E_1, \ldots, E_l$, $l = \lceil 2 \log n \rceil$. We let F_i denote a forest of 2-edge connected components of $G_i = (V, \cup_{j \leq i} E_j \cup F)$, where $F_i \subseteq F_{i+1}$. The level of a tree edge e is defined to be the smallest i such that e is covered in G_i. Let $l = \lceil \log m \rceil + 2$.

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 the number of non-tree edges incident to T.

We maintain the following data structures:

- F is stored in a dynamic tree data structure $D(F)$ whose edges are marked only while processing a deletion [9].
- For each level i, F_i is stored in a dynamic tree data structure $D(i)$. Each tree edge which is also in F_j, $j < i$, is colored black; all other edges are initially white.
- 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. An efficient data structure for maintaining a minimum spanning tree with a small number of weights can be found in [6, 7].

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. Set $F_1 = F_2 = ... = F_l$. Construct the $D(i)$ accordingly.
To answer the query: "Are x and y 2-edge connected?": To test if x and y are 2-edge connected, check if they are connected in $D(l)$. 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 is in level i. Call **Delete**(e, i).

Delete (e, i)

Case A: $e \in F$: If $e \in F_l$ (then e is a bridge), remove e from all data structures representing F. Otherwise, use D(MST) to find a minimum cost replacement edge e' for e, make e a nontree edge of E_i and e' an edge of F and continue as in Case B.

Case B: $e \in E_i$:

1. Mark the edges of $\pi(u, v)$ (using $D(F)$). These are the possible new bridges of F.
2. *while* there exist unmarked edges *do*
   Let a and b be two leaves in the marked subforest of F. (Initially, a and b are u and v).
   Call **Test_Path(**i, a, b**)**.
3. *if* $\pi(u, v)$ was not covered on level i, increment i, and goto Step 1.

The procedure **Test_Path(**i, u, v**)** either

(1) determines that all edges in $\pi(u, v)$ are covered by a sequence of random samples; or

(2) finds one tree edge f in level i which is suspected of being "sparsely covered" by edges in E_i.
(2A) If f is sparsely covered, the algorithm moves to $i + 1$ those edges which cross the cut induced by f's removal and marks their path. Thus the tree in F_i containing f is split into two subtrees, and the data structures representing F_i are modified accordingly.
(2B) With very low probability, f is not sparsely covered. In this case f is not removed from F_i.
In either case the subtree of $T \setminus f$ which is marked in $D(F)$ is searched exhaustively to determine which edges have become bridges in F_i. $D(F)$ is unmarked.

Test_Path (i, u, v)
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 **Sample** returns **false**.

   (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 **Sample**(u, v, e_u, e'_u).

   (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 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 **Sample**(v, u, e_v, e'_v).

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

Sample (z, w, e_z, e'_z) requires that $e_z, e'_z \in \pi(z, w)$. Let c and c' be constants to be determined later.

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

1. If $w(T_z \setminus e_z) < c \log^2 n$ then the set X consists of all edges incident to $T_z \setminus e_z$. Otherwise the set X is determined by sampling $c \log^2 n$ edges of E_i incident to nodes of $T_z \setminus e_z$. An edge with both endpoints in $T_z \setminus e_z$ is picked with probability $2/w(T_z \setminus e_z)$ and an edge with one endpoint in $T_z \setminus e_z$ is picked with probability $1/w(T_z \setminus e_z)$. In this case X consists of one edge.

2. Let $e_z = \{x, y\}$, where x is the endpoint closest to z. Determine (using $D(i)$) if all tree edges in level i on $\pi(z, y)$ between e'_z and e_z (including e'_z and e_z) are covered by edges in X. If so, increment i_z and return **true**.

3. Else let $f = \{x', y'\}$ be the uncovered such edge nearest to y. Wlog let $x' \in \pi(z, y')$.

   (a) Search all edges in E_i incident to $T_z \setminus f$ and determine $S = \{$edges of F'_i connecting $T_z \setminus f$ and $T_w \setminus f\}$.

   (b) If $0 \leq |S| \leq w(T_z \setminus f)/(15c' \log n)$, $\{$*f is sparsely covered*$\}$, remove f from $D(i)$ update f's color in $D(i+1)$, and remove the elements of S from E_i and insert them into E_{i+1}. For all edges (a, b) in S, mark the path $\pi(a, b)$ (using data structure $D(F)$).

   (c) Determine all tree edges in $T_z \setminus f$ which are in level i and are not covered by an edge of the new E_i (using data structure $D(i)$) and remove them from F_i. Unmark all edges in $T_z \setminus f$ in $D(F)$.

   (d) Return **false**.

2.2 Proof of Correctness

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

Lemma 2.1 *With probability at least*[1] $1 - 1/n^2$, *when Step 3 in* **Sample** *is executed, then* $|S| \leq w(T_z \setminus e_z)/(15c' \log n)$.

[1] 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

Proof: For each edge $e' \in \pi(z, y)$ between e'_z and e_z (including e'_z and e_z), $2^{i_u - 1} < w(T_z \setminus e') \le 2^{i_u}$. Thus, each nontree edge incident to $T_z \setminus e'$ is sampled with probability at least $1/(4w(T_z \setminus e'))$. Let us test the hypothesis that $|S| \le w(T_z \setminus e_z)/(15c' \log n)$. The error probability, i.e., the probability that $|S| > w(T_z \setminus e')/(15c' \log n)$, but no edge of S is selected is

$$(1 - 1/(60c' \log n))^{c \log^2 n} = O(1/n^5)$$

for $c \ge 300c'$.

Thus the probability that $|S| > w(T_u \setminus e')/(c' \log n)$ for any of the at most n edges on $\pi(z, w)$ with $2^{i_u - 1} < w(T_z \setminus e') \le 2^{i_u}$ is at most $1/n^{3+}$, implying that that the probability is at most $1/n^2$ that it happens at any deletion. ∎

Thus with high probability a subtree is searched only once and then is split off from its tree. Note that the weight of the subtree is at most half the weight of its tree. In the following we denote the split-off subtree by T_1. Let m_i be the number of edges ever in E_i.

Lemma 2.2 *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_0 and charges one coin to each edge e of E_i incident to T_0. Let w_0 be the weight of T_0 when T_0 was created. We show that if e's regular account balance was not reset since the creation of T_1, then $w(T_0) \le 3w_0/4$. 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_1 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_1. Thus, immediately before the split, $w(T_1) \le 3w_0/2$. Since $w(T_0) \le w(T_1)/2$, the claim follows. ∎

Lemma 2.3 *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 2.2, $\sum w(T_1) \le 15 m_{i-1} \log n$. 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 2.4 *For $l = \lfloor \log m \rfloor + 2$ all nontree edges of G are contained in some E_i for $i < l$, i.e. E_l is empty.*

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

Corollary 2.5 $\sum_i m_i = O(m)$.

Lemma 2.6 *If in* **Test_Path** *i_u and i_v are greater than $\log w(T) - 1$ 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 $w(T_v \setminus e_v) \le w(T)/2$. Let e_u be the furthest edge from u on $\pi(u, v)$ such that $w(T_u \setminus e_u) \le w(T)/2$ and let e'_u be its incident edge closer to v. Note that $w(T_u \setminus e'_u) > w(T)/2$. Thus $w(T_v \setminus e'_u) \le w(T)/2$, 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. ▮

Theorem 2.7 *The $D(i)$ are correctly maintained, i.e., they represent the F_i and a tree edge is marked iff it is in F_{i-1}.*

Proof: The correctness of the data structures depends on the 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' 's coverage 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

Sampling edges: We keep F_i in a ET-tree data structure, and with each active occurrence we store the nontree edges in E_i which are incident to corresponding vertex.
Covering paths: We use $D(i)$. Each edge in F_{i-1} has cost has cost ∞. At the start of each **Delete**, all other tree edges (i.e. those in level i) have cost 0. We implement the following operations:

- $Cover(s, t, u, v)$: Find the node a (b) closest to s (t) on $\pi_{\bar{F}}(u, v)$ and add 1 to all edges on $\pi_{\bar{F}}(a, b)$.
- $Uncover(s, t, u, v)$: Find the node a (b) closest to s (t) on $\pi_{\bar{F}}(u, v)$ and subtract 1 from all edges on $\pi_{\bar{F}}(a, b)$.
- $FirstUncovered(y, u)$: Return the edge on $\pi(y, u)$ that is closest to y and in level i and has cost 0.

- *LastUncovered*(y, u): If no edge on $\pi(y, u)$ has cost 0, return the edge incident to u on $\pi(y, u)$. Otherwise, determine the edge e on $\pi(y, u)$ of cost greater than 0 which is closest to y and return the edge of cost 0 immediately before e on $\pi(y, u)$ (if it exists).

- *Midpoint(u,v):* Return the middle edge of $\pi u, v$.

- *Pathwt*(u, v, wt)*:* Find the furthest edge e_v from u on $\pi(u, v)$ such that $w(T_u \setminus e_v \leq wt)$.

Using dynamic trees each operation except the last can be implemented in time $O(\log n)$. *Pathwt*(u, v, wt) 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 $w(T_u \setminus e_i)$ to wt.

The coverage data structure is used in *Sample* to cover the edges on $\pi(u, v)$ with the sampled edges and to find the first uncovered edge on level i.

Afterwards we uncover the edges on $\pi(u, v)$ again.

2.4 Analysis of Running Time

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

For each tree edge which is deleted, 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]).

In the case where the path is completely covered by a sequence of sampled edges, 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 $O(\log w(T)(\log^2 n + \log n \min(w(T), \log^2 n)) = O(\min(w(T), \log^2 n) \log^2 n)$.

Consider next the case where the path is not completely covered. Let f be the first uncovered edge in step 3 and let $T_1 = T_z \setminus f$. An exhaustive search of subtree T_1 is carried out and **Test_Path** stops. The cost of the exhaustive search is $O(\log n)$ per nontree edge, using the ET-tree data structure, or $O(w(T_1) \log n)$. (In this case sampling may cost as much as the exhaustive search.) 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 subtree in T_z 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_1 is split off from T_z. From Lemma 2.2, we see that on level i, $\sum w(T_1) \leq 15 m_i \log n$. Thus the total cost of **Test_Path**'s which terminate with the discovery of a sparsely covered edge f is $O(m_i \log^3 n)$. Summed over all levels this is $O(m \log^3 n)$. We charge this cost to the edges, charging $O(\log^3 n)$ each.

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 $O(m \log^2 n/n^2) = O(\log^2 n)$ to the expected cost of the algorithm.

Lemma 2.8 *During a* **Delete** *the number of times* **Test_Path** *runs to completion on level i, having covered a path by successful sampling is no greater than the number of paths marked by the algorithm, i.e., the number of deletions on level i plus the number of edges moved to level* $i + 1$.

Proof: It is not hard to see that the total number Z of leaves plus number of nodes of degree 3 in the marked subtree is no greater than twice the number of paths which has been marked. Each marked path contributes at most 2 to this total. Each time **Test_Path** runs to completion, having covered a path by successful sampling, two leaves are removed from a marked subtree, subtracted 2 from Z. In addition, some degree 3 nodes may become leaves but this doesn't affect Z. While Each time **Test_Path** terminates without covering the complete path, a marked subtree is unmarked which does not increase Z. ▮

We now explain how we charge for the costs on level $j \leq i$ during a **Delete**(e, i), where $e = (u, v)$, i.e., the cost of **Test_Path**(j, u, v). If $\pi(u, v)$ was successfully covered with sampled edges by **Test_Path**(j, u, v), the incurred cost on level j is $O(\log^4 n)$ and no calls of **Test_Path** on levels $> j$ occur. In this case we charge the cost to the deletion of edge e. Note that at most one level charges to the *delete* operation, i.e., the charge per operation is $O(\log^4 n)$.

If $\pi(u, v)$ was not successfully covered with sampled edges by **Test_Path**(j, u, v), then incurred cost on level j is $O((1 + d_j) \log^4 n)$ where d_j is the number of edges moved from level j to level $j + 1$ during the **Test_Path**. In this case there was a sequence of splits of the tree T containing u and v on level j.

Consider one of these splits. Let S^c be the set of edges moved to level $j + 1$ by the split and let T_1 be the subtree searched after the split. As observed earlier, the weight of T_1 is at most half the weight of its tree before the split. We charge the cost of moving the edges of S^c to level $j + 1$ to the nontree edges incident to T_1. Thus, the charge per nontree edge is $O(|S^c| \log^4 n / w(T_1)) = O(log^3 n)$. This accounts for all the cost occurred during **Test_Path**(j, u, v).

Since $\sum w(T_1) \leq 8m_i \log n$, the total expected charge that a nontree edge ever receives on level i is $O(\log^4 n)$.

3 A Dynamic 2-Edge Connectivity Algorithm

The basic idea is to insert edges into the last level and rebuild levels as necessary. Let F be a spanning forest of G. All nontree edges of G are put into E_1 and the other E_i are empty. As in the deletions-only algorithm, edges may move to higher levels.

Throughout the algorithm, the nontree edges of G are partitioned into levels $E_1, \ldots, E_l$, $l = \lceil 2 \log n \rceil$. We let F_i denote a forest of 2-edge connected components of $G_i = (V, (\cup_{j \leq i} E_j) \cup F)$, where $F_i \subseteq F_{i+1}$. The level of a tree edge e is defined to be the smallest i such that e is covered in G_i.

We represent F_i for each level i in as labelled subtrees in a dynamic tree data structure representing F. Unlike the deletions-only algorithm, we also keep a compressed version of F_i^c 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 compressed forest $F_i^c = (V_i^c, E^c)$ is initially constructed using E_i. Suppose for each edge $\{x, y\}$ in E_i, the path $\{x, y\}$ in F is marked. Let F^m denote the subforest of marked edges in F. 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 E^c iff $x, y \in V_i^c$, the path between x and y is in F^c and there are no other nodes in V_i^c which are on the path $\pi(x, y)$. We say that *path* $\pi(x, y)$ *is represented by superedge* $\{x,y\}$.

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.

Note that as the level number increases, the 2-edge connected components of F_i may contain more nodes, but their compressed version, whose size is linear in the size of E_i is smaller.

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 labelled with its name.
- a dynamic tree data structure $C(i)$ storing F_i where an edge e has cost $c_i(e)$ if it is represented by $c_i(e)$ superedges in F_j, $j \leq i$.
- for each 2-edge connected component of F_i^c 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 *plus* the number of superedges contained in it.

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 keep the following invariants.

Invariants:

1. All edges of F_i which are covered by edges in E_i are represented in F_i^c, by exactly one superedge.
2. All edges of F which are represented by a superedge in F_i^c must be in F_i.
3. If two nodes 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.
4. In $C(i)$, an edge of F has cost $c(e)$ if it is represented by exactly $c(e)$ superedges in F_j^c, $j \leq i$.

3.0.1 Subroutines

We introduce the following subroutines for operations on compressed forests:

remove(i,e): This assumes $e = \{u, v\}$ is a superedge in 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 $N(i)$.

insert(i,e): This inserts the superedge $e = \{u, v\}$ into F_i^c and assumes no part of $\pi(u, v)$ is represented in F_i^c.

For each level $j \geq i$ do
(1) Insert 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 $N(i)$.
(4) Add u and v to V_i^c if they are not in V_i^c.

connect(i,e): 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.

Case A: *No portion of* $\pi(u, v)$ *is represented in* F_i^c*:* $insert(i, e)$.

Case B: *Portions of $\pi(u, v)$ are represented but not the whole path.*

If node u is not in V_i^c but the edge in $\pi(u, v)$ containing u is represented: Let x and y be the nodes which are in V_i^c and which lie closest to u on either side of the labelled path. Do $remove(i, \{x, y\})$; $insert(i, \{x, u\})$; $insert(i, \{u, y\})$.

Repeat the following steps until all of $\pi(u, v)$ is represented:

1. Let u' be the node closest to u such that $\{u', v'\}$ is in $\pi(u, v)$ and is not represented. If u' is not in V_i^c, add it to V_i^c. If u' lies on a labelled path let x and y be the nodes which are in V_i^c and which lie closest to u' on either side of the represented path. Do $remove(i, \{x, y\})$, do $insert(i, \{x, u'\})$, and $insert(\{u', y\})$.
2. Find the closest node z to u' in a represented portion of the path $\pi(u', v)$.
3. If z is not in V_i^c, add it to V_i^c. Then z lies on a labelled path. Let x and y be the nodes which are in V_i^c and which lie closest to z on either side of the represented path. Do $remove(i, \{x, y\})$,
4. Do $insert(i, \{x, z\})$, $insert(\{z, y\})$, and $insert(\{u', z\})$.

- Repeat case B for v substituted for u.

3.1 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, i.e., $N(i)$, and $C(i)$ for each level i. If e is a nontree edge then do $insert(l, e)$.

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 $b_0, ..., b_{l-1}$, where b_0 is the most significant bit, then a rebuild of level i occurs each time the i^{th} bit flips to 1.

3.2 Rebuilding level i:

During a level i rebuild, we remove all nontree edges from E_j $j > i$ and put them into E_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 $remove(j, e)$; also discard the ET-trees. Construct the new compressed graph F_i^c by applying $connect(i, e)$ for each edge of E_i. Store the edges of E_i with the appropriate nodes in the new ET-trees which result.

3.3 Deletions

Here, we may sometimes need to insert superedges on level i to connect components of F_i which are not connected in F_i^c.

- *Executing* **Test_Path(i,u,v)***:*

  In **Sample**, the threshold for determining if a cut is sparse is lowered by a small factor (and the amount sampled is increased by a small factor) so that each level receives no more than 1/4 of the edges in the previous level. Then 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 $|E_i| < (3/4)n^2/2^i$.

Sample uses the ET-tree for F_i^c instead of the one for F_i, in order to obtain nontree edges in E_i. Note that when a nontree edge $\{u, v\}$ is deleted from a level i, there is only one ET-tree on level i which contains edges that might cover path $\pi(u, v)$, since that path was covered by an edge in E_i and every edge in E_i which covers a part of the path must lie in the same connected component of F_i^c. If **Test_Path(j,u,v)** is subsequently carried out on level $j > i$ then we may need to add superedges to F_j^c in order to connect portions of the path $\pi(u, v)$ which were (before the deletion) connected in F_j. Thus, before performing **Test_Path(j,u,v)**, we perform $connect(j, \{u, v\})$.

In the coverage data structure used by **Test_Path(i,u,v)** we use $C(i-1)$ in place of $D(i)$. In $C(i-1)$ a tree edge has cost greater than 0 iff it is covered on a level j, $j > i$. After incrementing the paths covered by the sampled edges, we test if $\pi(u, v)$ is covered.

To implement exhaustive search of a tree T in F_i^c, we again use $C(i-1)$ and increment all paths covered by nontree edges incident to T. For each superedge in T we check if the path it represents is covered in this augmented $C(i-1)$. Any superedge corresponding to a path that is not wholly covered is moved down (see below).

- *Moving a superedge $e = \{x, y\}$ from level i:* (This occurs when $\pi(x, y)$ is no longer wholly covered in $\cup_{j \le i} E_j$. This replaces the operation in the deletions-only algorithm in which a tree edge on level i is removed from F_i.)

  Do $remove(i, e)$; $connect(i + 1, e)$.

- *Moving a nontree edge $e = \{x, y\}$ from level $i - 1$ to level i:* Remove the edge from the ET-tree of level $i - 1$. Do $connect(i, e)$. Add the edge to the ET - tree of level i.

- *Updating when a tree edge $e = \{x, y\}$ is replaced by an edge $e' = \{x', y'\}$:* Let i be the level of the tree edge, i.e., the minimum j such that e is in F_j.
  (1) If there is a superedge $e_j = \{u, v\}$ in F_j^c which represents a path containing e, then do $remove(j, e_j)$ for all $j \ge i$;
  (2) Remove e from F as represented in $C(j)$ and $N(j)$ for each j.
  (3) Add e' to F as represented in $C(j)$ and $N(j)$ for each j and do $connect(j, e')$ for all $j \ge i$.

3.4 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.

3.5 Proof of correctness

We will show in this section that the invariants hold.

Lemma 3.1 *All edges of F_i that are covered by edges in E_i are represented in F_i^c by exactly one superedge.*

Proof: Note that if the claim holds before a call to $connect(i, e)$, 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. It follows that the claim holds right after rebuilding level i.

Consider next a deletion. There are 3 cases to consider: Adding an edge to E_i, removing a superedge from F_i^c, and updating whenever a tree eege is replaced by e'. Whenever a nontree edge e is added to level i, $connect(i, e)$ is called and the claim holds. Whenever a superedge is removed from level i, all edges in E_i on its cut are removed as well. When e is replaced by e', $connect(i, e)$ is called in the updated F, guaranteeing that for every edge $\{x, y\} \in E$; that lies on the cut of e, $\pi(x, y)$ is represented by a path of superedges. Thus the claim holds. ▮

Lemma 3.2 *All edges of F that are represented by a superedge in F_i^c must be in F_i.*

Proof: During a rebuild all superedges added to F_i^c represent edges of F covered by an edge of E_i. Thus the claim holds. Assume the claim holds before a $connect(i, e)$ operation, where $e = \{u, v\}$. Then afterwards all edges of F represented by a superedge in F_i^c either are in F_i or on $\pi(u, v)$. The deletion of an edge does not add any superedges to F_i^c, except in the following four cases:

(A) Moving a superedge from level $i - 1$ to level i. Note that the claim might not hold immediately after such a move. However, for each moved superedge $\{u, v\}$ either (a) there is a call **Test_path**(i, a, b) with $\{u, v\} \in \pi(a, b)$ which removes $\{u, v\}$ if it is not covered by $E_i \cup \cup_{j \le i} F_j^c$, or (b) a nontree edge $\{a, b\}$ was moved to E_i at the same time as $\{u, v\}$ and $\{u, v\} \in \pi(a, b)$.

(B) Moving a nontree edge from level $i - 1$ to level i. Obviously the claim holds after moving a nontree edge to level i.

(C) Updating when a tree edge is replaced by an edge $\{x', y'\}$ on level $j \le i$. Note that all tree edges on $\pi(u, v)$ are connected on level j and, thus, belong to F_i. As shown above, after the update every edge of F represented by a superedge in F_i^c either is in F_i or on $\pi(u, v)$. Thus the claim holds.

(D) Before a call to **Test_Path (i, u, v)**. Superedges covering all edges of $\pi(u, v)$ are added. **Test_Path** removes all superedges on $\pi(u, v)$ that are not covered by $E_i \cup \cup_{j \le i} F_j^c$. ▮

Lemma 3.3 *If two nodes 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, $j \le i$, is rebuilt.*

Proof: Note that neither an $insert(i, e)$ nor a $connect(i, e)$ disconnects previously connected nodes in F_i^c, only a $remove(i, e)$ does. A $remove(i, e)$ is executed either (a) during a rebuild on level $j \le i$, (b) when moving a superedge from level i to level $i + 1$, or (c) when a tree edge is replaced by a nontree edge. In the latter case, the two nodes are reconnected in F_i^c by the following connect. In case (b), the removed superedge $\{x, y\}$ is no longer covered by edges of $E_i \cup \bigcup_{j<i} F_j^c$. To finish the proof of the lemma, we need to show in this case that x and y are no longer connected in F_i. Assume by contradiction that x and y are still connected. Then each edge e' of $\pi(x, y)$ is covered by at least one edge in $\bigcup_{j \le i} E_j$. By Lemma 3.1 e' is represented by a superedge in $\bigcup_{j<i} F_j^c$ or an edge in E_i. Contradiction. ▮

Lemma 3.4 *In $C(i)$, an edge of F has cost $c(e)$ if it is represented by exactly $c(e)$ superedges in F_j^c, $j \le i$.*

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. ▮

3.6 Analysis of the Running Time.

The routines *insert(i,e)* and *remove(i,e)* take $O(\log n)$ time per level or $O(\log^2 n)$ time. The routine *connect(i,e)* 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 $O(\log n)$ to do one $insert(l, e)$. If a tree edge is inserted then every dynamic tree containing F must be modified for a total cost of $O(\log^2 n)$.

A rebuild requires a $remove$ for each superedge and a $connect$ for each edge in E_i. We will show the number of superedges in F_i^c is proportional to 2^{l-i}. Thus the total cost is $O((\log^2 n)2^{l-i})$.

The analysis of **Test_Path** is almost the same as in the deletions-only algorithm, except that we need to execute $connect$ once. Unlike the deletions-only algorithm, we use weight of an ET-tree to mean the number of nontree edges plus the size of the ET-tree (number of superedges in it). Here, the cost of exhaustive search includes the cost of checking each superedge to determine if it is still covered after nontree edges are moved down.

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 heavier component of F_i, rather than the lighter. Yet the weight of this portion must be no greater than the weight 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 charged to a level i between two consecutive rebuilds of levels i or lower is the same as in the deletions-only algorithm. That is, the cost is proportional to the number of nontree edges in E_i plus the number of superedges in F_i times a factor of $O(\log^4 n)$.

The cost of removing a superedge or moving a nontree edge from a lower level to a higher level is the cost of $remove$ plus the cost of $connect$ or $O(\log^2 n)$ plus $O(\log^2 n)$ times the number of superedges inserted into the lower level.

The cost of updating when a tree edge on level i is swapped is the cost of doing a $remove$ on each level $j \geq i$ and a $connect$ on each level $j \geq i$. The cost of the $remove$'s is $O(\log^3 n)$ plus $O(\log^2 n)$ times the number of superedges added.

Lemma 3.5 *The total number of superedges inserted into a level i between two consecutive rebuilds of any levels $j, j' \leq i$ is proportional to the size of E_i plus the number of operations since the rebuild.*

Proof: Initially, there are no more than $4\,|E_i|$ superedges in F_i^c since there are no more than $2|E_i|$ leaves in the marked tree subtree of F and every node of V_i^c is either a leaf or has degree at least 3. Each operation adds no more than a constant number of superedges, unless it is connecting up components of F_i^c each of which already contain superedges. By invariant (3) two components can only be connected once in F_i^c, as they remain connected until they are no longer connected in F_i. Therefore the total number of superedges in a level i is $big-O$ of the number of operations executed between rebuilds plus $|E_i| = O(2^{l-i})$. ▮

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 $O(\log^5 n)$.

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

4 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 $O(\log n / b)$.