# Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity

Jacob Holm* Kristian de Lichtenberg* Mikkel Thorup*

## Abstract

Deterministic fully dynamic graph algorithms are presented for connectivity, minimum spanning forest, 2-edge connectivity, and biconnectivity. Assuming that we start with no edges in a graph with $n$ vertices, the amortized operation costs are $O(\log^2 n)$ for connectivity and $O(\log^4 n)$ for minimum spanning forest, 2-edge connectivity, and biconnectity.

## 1 Introduction

We consider the fully dynamic graph problems of connectivity, minimum spanning forest, 2-edge connectivity and biconnectivity. In a *fully dynamic graph problem*, we are considering a graph $G$ over a fixed vertex set $V$, $|V| = n$. The graph $G$ may be *updated* by insertions and deletions of edges. Unless otherwise stated, we assume that we start with an empty edge set.

For the *fully dynamic connectivity problem*, the updates may be interspersed with *connectivity queries*, asking whether two given vertices are connected in $G$. Both updates and queries are presented *on-line*, meaning that we have to respond to an update or query without knowing anything about the future. The connectivity problem reduces to the problem of maintaining a spanning forest (a spanning tree for each component) in that if we can maintain *any* spanning forest $F$ for $G$ at cost $O(t(n) \log n)$ per update, then, using dynamic trees [17], we can answer connectivity queries in time $O(\log n/ \log t(n))$. In this paper, we present a very simple deterministic algorithm for maintaining a spanning forest in a graph in amortized time $O(\log^2 n)$ per update. Connectivity queries are then answered in time $O(\log n/ \log\log n)$.

In the *fully dynamic minimum spanning forest problem*, we have weights on the edges, and we wish to maintain a minimum spanning forest $F$ of $G$. Thus, in connection with any update to $G$, we need to respond with the corresponding updates for $F$, if any. We will present a deterministic algorithm for maintaining a minimum spanning forest in a graph in $O(\log^4 n)$ amortized time per operation.

A graph is *2-edge connected* if and only if it is connected and no single edge deletion disconnects it. The 2-edge-connected components are the maximal 2-edge connected subgraphs, and two vertices $v$ and $w$ are 2-edge connected if and only if they are in the same 2-edge connected component, or equivalently, if and only if $v$ and $w$ are connected by two edge-disjoint paths. A graph is *biconnected* if and only if it is connected and no single vertex deletion disconnects it. The biconnected components are the maximal biconnected subgraphs, and two vertices $v$ and $w$ are biconnected if and only if they are in the same biconnected component, or equivalently, if and only if either $(v, w)$ is an edge or $v$ and $w$ are connected by two internally disjoint paths. In the *fully dynamic 2-edge and biconnectivity problems*, the updates may be interspersed with queries asking whether two given vertices are 2-edge or biconnected. We present $O(\log^4 n)$ algorithms for these two problems.

**Previous work** For deterministic algorithms, all the previous best solutions to the fully dynamic connectivity problem were also solutions to the minimum spanning tree problem. In 1985 [5], Frederickson introduced a data structure known as *topology trees* for the fully dynamic minimum spanning tree problem with a worst case cost of $O(\sqrt{m})$ per update, permitting connectivity queries in time $O(\log n/ \log(\sqrt{m}/ \log n)) = O(1)$. In 1992, Epstein et. al. [3] improved the update time to $O(\sqrt{n})$ using the *sparsification technique*. Finally in 1997 Henzinger and King [12] gave an algorithm with $O(\sqrt[3]{n} \log n)$ update time and constant time per connectivity query.

Randomization has been used to improve the bounds for the connectivity problem. In 1995 [10], Henzinger and King showed that a spanning forest could be maintained in $O(\log^3 n)$ expected amortized time per update. Then connectivity queries are supported in $O(\log n/ \log\log n)$ time. The update time was further improved to $O(\log^2 n)$

*E-mail: (samson,morat,mthorup)@diku.dk. Department of Computer Science, University of Copenhagen.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee
STOC 98 Dallas Texas USA
Copyright ACM 1998 0-89791-962-9 98 5...$5.00

in 1996 [14] by Henzinger and Thorup. No randomized technique was known for improving the deterministic $O(\sqrt[3]{n}\log n)$ update cost for the minimum spanning tree problem.

In 1991 [6], Fredrickson succeeded in generalizing his $O(\sqrt{m})$ bound from 1983 [5] for fully dynamic connectivity to fully dynamic 2-edge connectivity. As for connectivity, the sparsification technique of Eppstein et.al. [3] improved this bound to $O(\sqrt{n})$. Further, Henzinger and King generalized their randomization technique for connectivity to give an $O(\log^5 n)$ expected amortized bound [10, 11]. It should be noted that the above mentioned improvement for connectivity of Henzinger and Thorup [14], does not affect the $O(\log^5 n)$ bound for 2-edge connectivity.

For biconnectivity, the previous results are a lot worse. The first non-trivial result was a deterministic bound of $O(m^{2/3})$ from 1992 by Rauch [8]. In 1994 [16], Rauch improved this bound to $O(\min\{\sqrt{m}\log n, n\})$. In 1995, (Rauch) Henzinger and Poutré further improved the deterministic bound to $O(\sqrt{n\log n}\log\lceil m/n\rceil)$ [13]. In 1995 [9], Henzinger and King generalized their randomized algorithm from [10] to the biconnectivity problem to achieve an $O(\Delta\log^4 n)$ expected amortized cost per operation, where $\Delta$ is the maximal degree (In [9], the bound is incorrectly quoted as $O(\log^4 n)$ [Henzinger, personal communication, 1997]).

Finally, we note that for all of the above problems, we have a lower bound of $\Omega(\log n/\log\log n)$ which was proved independently by Fredman and Henzinger [7] and Miltersen, Subramanian, Vitter, and Tamassia [15].

For the incremental (no deletions) and decremental (no insertions) problems, the bounds are as follows. Incremental connectivity is the union-find problem, for which Tarjan has provided an $O(\alpha(m,n))$ bound [18]. Westbrook and Tarjan have obtained the same time bound for incremental 2-edge and biconnectivity [20]. Further Sleator and Tarjan have provided an $O(\log n)$ bound [17] for incremental minimum spanning forest.

Decrementally, for connectivity and 2-edge connectivity, Thorup has provided an $O(\log n)$ bound if we start with $\Omega(n\log^6 n)$ edges, and an $O(1)$ bound if we start with $\Omega(n^2)$ edges [19]. For decremental minimum spanning tree and biconnectivity, no better bounds were known than those for the fully dynamic case.

**Our results** First we present a very simple deterministic fully dynamic connectivity algorithm with an update cost of $O(\log^2 n)$, thus matching the previous best randomized bound and improving substantially over the previous best deterministic bound of $O(\sqrt[3]{n}\log n)$.

Our technique relies on some of the same intuition as was used in Henzinger and King [10] in their randomized algorithm. Our deterministic algorithm is, however, much simpler, and in contrast to their algorithm, it generalizes to the minimum spanning tree problem. More precisely, a specialization of our connectivity algorithm gives a simple decremental minimum spanning tree with an amortized cost of $O(\log^2 n)$ per operation for any sequence of $\Omega(m)$ operations. Then we use a technique from [12] to convert our deletions-only structure to a fully dynamic data structure for the minimum spanning tree problem using $O(\log^4 n)$ amortized time per update. This is the first polylogarithmic bound for the problem, even when we include randomized algorithms.

Finally, our connectivity techniques are generalized to 2-edge and biconnectivity, leading to $O(\log^4 n)$ algorithms for both of these problems. The generalization uses some of the ideas from [6, 9, 11] of organizing information around a spanning forest. However, finding a generalization that worked was rather delicate, particularly for biconnectivity, where we needed to make a careful recycling of information, leading to the first polylogarithmic algorithm for this problem.

The reader is referred to [2, 3, 4, 5, 6, 10] for discussions of problems that get improved by our new fully dynamic graph algorithms.

## 2 Connectivity

In this section, we present a simple $O(\log^2 n)$ time deterministic fully dynamic algorithm for graph connectivity. First we give a high level description, ignoring all problems concerning data structures. Second, we implement the algorithm with concrete data structures and analyze the running times.

### 2.1 High level description

Our dynamic algorithm maintains a spanning forest $F$ of a graph $G$. The edges in $F$ will be referred to as *tree-edges*. Internally, the algorithm associates with each edge $e$ a level $\ell(e) \leq L = \lfloor\log_2 n\rfloor$. For each $i$, $F_i$ denotes the sub-forest of $F$ induced by edges of level at least $i$. Thus, $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$. The following invariants are maintained.

**(i)** $F$ is a maximum (w.r.t. $\ell$) spanning forest of $G$, that is, if $(v,w)$ is a non-tree edge, $v$ and $w$ are connected in $F_{\ell(v,w)}$.

**(ii)** The maximal number of nodes in a tree in $F_i$ is $\lfloor n/2^i\rfloor$. Thus, the maximal relevant level is $L$.

Initially, all edges have level 0, and hence both invariants are satisfied. We are going to present an amortization argument based on increasing the levels of edges. The levels of edges are never decreased, so we can have at most $L$ increases per edge. Intuitively speaking, when the level of a non-tree edge is increased, it is because we have discovered that its end points are close enough in $F$ to fit in a smaller tree on a higher level. Concerning tree edges, note that increasing their level cannot violate (i), but it may violate (ii).

We are now ready for a high-level description of insert and delete.

**Insert($e$):** The new edge is given level 0. If the end-points were not connected in $F = F_0$, $e$ is added to $F_0$.

**Delete($e$):** If $e$ is not a tree-edge, it is simply deleted. If $e$ is a tree-edge, it is deleted and a *replacement edge*, reconnecting $F$ at the highest possible level, is searched for. Since $F$ was a maximum spanning forest, we know that the replacement edge has to be of level at most $\ell(e)$. We now call Replace($e, \ell(e)$). Note that when a tree-edge $e$ is deleted, $F$ may no longer be spanning, in which case (i) is violated until we have found a replacement edge. In the time in between, if $(v, w)$ is not a replacement edge, we still have that $v$ and $w$ are connected in $F_{\ell(v,w)}$.

**Replace($(v, w), i$):** Assuming that there is no replacement edge on level $> i$, finds a replacement edge of the highest level $\leq i$, if any.
Let $T_v$ and $T_w$ be the trees in $F_i$ containing $v$ and $w$, respectively. Assume, without loss of generality, that $|T_v| \leq |T_w|$. Before deleting $(v, w)$, $T = T_v \cup \{(v, w)\} \cup T_w$ was a tree on level $i$ with at least twice as many nodes as $T_v$. By (ii), $T$ had at most $\lfloor n/2^i \rfloor$ nodes, so now $T_v$ has at most $\lfloor n/2^{i+1} \rfloor$ nodes. Hence, preserving our invariants, we can take all edges of $T_v$ of level $i$ and increase their level to $i + 1$, so as to make $T_v$ a tree in $F_{i+1}$.
Now level $i$ edges incident to $T_v$ are visited one by one until either a replacement edge is found, or all edges have been considered.
Let $f$ be an edge visited during the search.
If $f$ does not connect $T_v$ and $T_w$, we increase its level to $i + 1$. This increase pays for our considering $f$.
If $f$ does connect $T_v$ and $T_w$, it is inserted as a replacement edge and the search stops.
If there are no level $i$ edges left, we call Replace($(v, w), i - 1$); except if $i = 0$, in which case we conclude that there is no replacement edge for $(v, w)$.

### 2.2 Implementation

For each $i$, we wish to maintain the forest $F_i$ together with all non-tree edges on level $i$. For any vertex $v$, we wish to be able to:

- Identify the tree $T_v$ in $F_i$ containing $v$.
- Compute the size of $T_v$.
- Find an edge of $T_v$ on level $i$, if one exists.
- Find a level $i$ non-tree edge incident to $T_v$, if any.

The trees in $F_i$ may be cut (when an edge is deleted) and linked (when a replacement edge is found, an edge is inserted or the level of a tree edge is increased). Moreover, non-tree edges may be introduced and any edge may disappear on level $i$ (when the level of an edge is increased or when non-tree edges are inserted or deleted).

All the above operations and queries may be supported in $O(\log n)$ time using the ET-trees from [10], to which the reader is referred for additional details. An ET-tree is a standard balanced binary tree over the Euler tour of a tree. Each node in the ET-tree represents the segment of the Euler tour below it. The point in considering Euler tours is that if trees in a forest are linked or cut, the new Euler tours can be constructed by at most 2 splits and 2 concatenations of the original Euler tours. Rebalancing the ET-trees affects only $O(\log n)$ nodes.

Here we have an ET-tree over each tree in $F_i$. Each node of the ET-tree contains a number telling the size of the Euler tour segment below it, a bit telling if any tree edges in the segment have level $i$, and a bit telling whether there is any level $i$ non-tree edges incident to a vertex in the segment.

Given a vertex $v$ we can find the tree $T_v$ containing $v$ by moving $O(\log n)$ steps up till we find a root of an ET-tree. This root represents the Euler tour of $T_v$. The size $s$ of the Euler tour of a tree is twice the number of edges, so the number of vertices is $s/2 + 1$. To find a tree edge of level $i$ or an incident non-tree edge, if any, we move $O(\log n)$ steps down the ET-tree, using the bits telling us under which nodes such edges are to be found. If a tree edge $(v, w)$ is moved from level $i$, we only need to update the bits on the paths from $(v, w)$ and $(w, v)$ to the root, using $O(\log n)$ time. If a non-tree edge $(v, w)$ is introduced/disappear, we only need to update the bits on the paths from $v$ and $w$ to their respective roots. This takes $O(\log n)$ time. When the trees are cut or linked, only $O(\log n)$ nodes are affected, and the information in each node is updated in constant time.

It is now straightforward to analyze the amortized cost of the different operations. When an edge $e$ is inserted on level 0, the direct cost is $O(\log n)$. However, its level may increase $O(\log n)$ times, so the amortized cost is $O(\log^2 n)$.

Deleting a non-tree edge $e$ takes time $O(\log n)$. When a tree edge $e$ is deleted, we have to cut all forests $F_j$, $j \leq \ell(e)$, giving an immediate cost of $O(\log^2 n)$. We then have $O(\log n)$ recursive calls to Replace, each of cost $O(\log n)$ plus the cost amortized over increases of edge levels. Finally, if a replacement edge is found, we have to link $O(\log n)$ forests, in $O(\log^2 n)$ total time.

Thus, the cost of inserting and deleting edges from $G$ is $O(\log^2 n)$. The balanced binary tree over $F_0 = F$ immediately allows us to answer connectivity queries between arbitrary nodes in time $O(\log n)$. In order to reduce this time to $O(\log n/\log\log n)$, as in [10], we introduce an extra balanced $\Theta(\log n)$-ary B-tree over the Euler tour of each tree in $F$. The B-tree has depth $O(\log n/\log\log n)$, which is hence the time it takes for a connectivity query. Each delete or insert gives rise to at most one cut and one link in $F$, and for $\Theta(\log n)$-ary B-trees, such operations can be supported in $O(\log^2 n/\log\log n)$ time. Thus, we conclude:

**Theorem 1** *Given a graph $G$ with $m$ edges and $n$ vertices, there exists a deterministic fully dynamic algorithm that answers connectivity queries in $O(\log n/\log\log n)$ time worst case, and uses $O(\log^2 n)$ amortized time per insert or delete.*

## 3 Minimum spanning forest

We will now expand on the ideas from the previous section to the problem of maintaining a minimum spanning forest (MSF). First we present an $O(\log^2 n)$ deletions-only algorithm, and then we apply a general construction from [12] transforming a deletions-only MSF algorithm into a fully dynamic MSF algorithm.

### 3.1 Decremental minimum spanning forests

It turns out that if we only want to support deletions, we can obtain an MSF-algorithm from our connectivity algorithm by some very simple changes. The first is, of course, the initial spanning forest $F$ has to be a minimal spanning forest. The second is that when in replace (cf. page 3). we consider the level $i$ non-tree edges incident to $T_v$, instead of doing it in an arbitrary order, we should do it in order of increasing weights. That is, we repeatedly take the lightest incident level $i$ edge $e$: if $e$ is a replacement edge, we are done; otherwise, we move $e$ to level $i+1$, and repeat with the new lightest incident level $i$ edge, if any. For the above changes to work, it is crucial, that *all weights are distinct.* To ensure this, we associate a unique number with each edge. If two edges have the same weight, it is the one with the smaller number that is the smaller.

To see that the above simple changes suffice to maintain that $F$ is a minimum spanning forest, we will prove that in addition to (i) and (ii), the following invariant is maintained:

(iii) If $e$ is the heaviest edge on a cycle $C$, then $e$ has the lowest level on $C$.

The original replace function found a replacement edge on the highest possible level, but now, among the replacement edges on the highest possible level, we choose the one of minimum weight. Using (iii), we will show that this edge has minimum weight among all replacement edges.

**Lemma 2** *For any tree edge e, among all replacement edges, the lightest edge is on the maximum level.*

**Proof:** Let $e_1$ and $e_2$ be replacement edges for $e$. Let $C_i$ be the cycle induced by $e_i$; then $e \in C_i$. Suppose $e_1$ is lighter than $e_2$. We want to show that $\ell(e_1) \geq \ell(e_2)$.

Consider the cycle $C = (C_1 \cup C_2) \setminus (C_1 \cap C_2)$. Since $F$ is a minimum spanning forest, we know that $e_i$ is the heaviest edge on $C_i$. Hence $e_2$ is the heaviest edge on $C$. By (iii) this implies that $e_2$ has the lowest level on $C$. In particular, $\ell(e_1) \geq \ell(e_2)$. □

Since our algorithm is just a specialized version of the decremental connectivity algorithm, we already know that (i) and (ii) are maintained.

**Lemma 3** *(iii) is maintained.*

**Proof:** Initially (iii) is satisfied since all edges are on level 0. We will now show that (iii) is maintained under all the different changes we make to our structure during the deletion of an edge. If an edge $e$ is just deleted, any cycle in $G \setminus \{e\}$ also existed in $G$, so (iii) is trivially preserved. Also note that replacing a deleted tree-edge cannot in itself violate (iii) since it does not change the levels or weights of any edges.

Our real problem is to show that (iii) is preserved during Replace when the level of an edge $e$ is increased. This cannot violate (iii) if $e$ is not the heaviest edge on some cycle, so assume that $e$ is the heaviest edge on a cycle $C$. To prove that (iii) is not violated, we want to show that before the increase, all other edges in $C$ have level $\geq i+1$.

No tree edge is heaviest on any cycle, so $e$ is a non-tree edge. When $\ell(e)$ is to be increased from $i$ to $i+1$, we know it is the lightest level $i$ edge incident to $T_v$ (cf. the description of replace on page 3). Moreover, by (iii), all other edges on $C$ have level at least $i$. Thus, all other edges from $C$ incident to $T_v$ have level at least $i+1$.

To complete the proof, we show that all edges in $C$ are incident to $T_v$. Suppose, for a contradiction, that $C$ contained an edge $f$ leaving $T_v$. Since $e$ is to be increased, $e \neq f$. Also, the call to Replace requires that there is no replacement edge of level $> i$, so $\ell(f) \leq i$. This contradicts that all edges $\neq e$ from $C$ incident to $T_v$ have level $\geq i+1$. □

It has now been established that the above change in replace suffice to maintain a minimum spanning forest. A last point is that we need to modify our ET-trees to give us the lightest non-tree edge incident to a tree. So far, for each node in the ET-trees, we had a bit telling us whether the Euler tour segment below it had an incident non-tree edge. Now, with the node, we store the minimum weight of a non-tree edge incident to the Euler tour segment below it. Clearly, we can still support the different operations in $O(\log n)$ time. We conclude

**Theorem 4** *There exists a deletions-only MSF algorithm that can be initialized on a graph with n nodes and m edges and support any sequence of $\Omega(m)$ deletions in total time $O(m \log^2 n)$.* □

### 3.2 Fully dynamic MSF

To obtain a fully dynamic minimum spanning forest algorithm we apply a general reduction, which is a slight generalization of the one provided by Henzinger and King [12, pp. 600-603]. The reduction is described as follows.

**Lemma 5** *Suppose we have a deletions-only MSF algorithm that for any k, l, can be initialized on a graph with k nodes and l edges and support any sequence of $\Omega(l)$ deletions in total time $O(l \cdot t(k,l))$ where t is non-decreasing. Then there exists a fully-dynamic MSF algorithm for a graph on n nodes starting with no edges, that for m edges, supports an update in amortized time*

$$O\left(\log^3 n + \sum_{i=1}^{\log_2 m} \sum_{j=1}^{i} t\left(\min\{n, 2^j\}, 2^j\right)\right).$$

**"Proof":** Essentially, we combine the reduction from [12] with a contraction idea from [11]. We will only sketch the changes needed in [12]. As in [12], we operate on a series of graphs $A_i$, where $A_i$ has $2^i$ non-tree edges. In [12], $A_i$ may have $n-1$ MSF-edges, and this forces them to introduce a special efficient operation for adding a batch of edges. Here, instead, when we first create $A_i$, we contract all MSF-paths

that are not incident to any non-tree edge. The "super" edge $e$ replacing a MSF-path $P$ gets the minimum weight on $P$. Moreover, if any edge from $P$ is deleted, we have to delete $e$ in $A_i$. As a result, we can base our fully-dynamic algorithm directly on deletions-only algorithms. □

From Theorem 4, we get $t(k,l) = O(\log^2 k)$, and hence we get a fully dynamic algorithm with update cost

$$O\left(\log^3 n + \sum_{i=1}^{\log_2 m} \sum_{j=1}^{i} \log^2\left(\min\{n, 2^j\}\right)\right) = O\left(\log^4 n\right).$$

Note for comparison, that in [12], Henzinger and King had $t(k,l) = O(\sqrt[3]{l}\log k)$, giving them an update cost of $O(\sqrt[3]{m}\log n)$. Then sparsification [3] reduces the cost to $O(\sqrt[3]{n}\log n)$. From the combination of Theorem 4 and Lemma 5, we conclude

**Theorem 6** *There is a fully-dynamic MSF algorithm that for a graph with n nodes and starting with no edges maintains a minimum spanning forest in $O(\log^4 n)$ amortized time per edge insertion or deletion.*

## 4 2-edge connectivity

In this section we present an $O(\log^4 n)$ deterministic algorithm for the 2-edge connectivity problem for a fully dynamic graph $G$. An important secondary goal is to present ideas and techniques that will be reused in the next section for dealing with the more complex case of biconnectivity.

As in the previous sections we will maintain a spanning forest $F$ of $G$. If $v$ and $w$ are connected in $F$, $v \cdots w$ denotes the simple path from $v$ to $w$ in $F$. If they are further connected to $u$, *meet*$(u, v, w)$ denotes the intersection vertex of the three paths $u \cdots v$, $u \cdots w$, and $v \cdots w$.

A tree edge $e$ is said to be *covered* by a non-tree edge $(v, w)$ if $e \in v \cdots w$, that is if $e$ is in the cycle induced by $(v, w)$. A *bridge* is an edge $e$ whose removal disconnects a component, or, equivalently, an edge whose end-points are not 2-edge connected. Hence $e$ is a bridge if and only if it is a tree edge not covered by any non-tree edge. Since 2-edge connectivity is a transitive relation on vertices, it follows that two vertices $x$ and $y$ are 2-edge connected if and only if they are connected in $F$ and all edges in $x \cdots y$ are covered [6].

Recall from connectivity that our spanning forest $F$ was a certificate of connectivity in $G$ in that vertices were connected in $G$ if and only if they were so in $F$. If an edge from $F$ was deleted, we needed to look for a replacement edge reconnecting $F$, if possible. An amortization argument paid for all non-replacement edges considered.

Now, for 2-edge connectivity we have a certificate consisting of $F$ together with a set $C$ consisting of a covering edge for each non-bridge edge in $F$. Thus two vertices are 2-edge connected in $G$ if and only if they are so in $F \cup C$. However, if an edge $f \in C$ is deleted, we may need to add several "replacement edges" to $C$ in order regain a certificate. Nevertheless, by carefully choosing the order in which potential replacement edges are considered, we will be able to amortize the cost of considering all but two of them.

### 4.1 High level description

The algorithm associates with each non-tree edge $e$ a level $\ell(e) \leq L = \lfloor \log_2 n \rfloor$. However, in contrast to connectivity, the tree edges do not have associated levels. For each $i$, let $G_i$ denote the subgraph of $G$ induced by edges of level at least $i$ together with the edges of $F$. Thus, $G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_L \supseteq F$. The following invariant is maintained:

**(i')** The maximal number of nodes in a 2-edge connected component of $G_i$ is $\lfloor n/2^i \rfloor$. Thus, the maximal relevant level is $L$.

Initially, all non-tree edges have level 0, and hence the invariant is satisfied. As for connectivity, we will amortize work over level increases. We say that it is *legal* to increase the level of a non-tree edge $e$ to $j$ if this does not violate (i'), that is, if the 2-edge connected component of $e$ in $G_j \bigcup \{e\}$ has at most $\lfloor n/2^j \rfloor$ vertices.

For every tree edge $e \in F$, we implicitly maintain the *cover level* $c(e)$ which is the maximum level of a covering edge. If $e$ is a bridge, $c(e) = -1$. The definition of a cover level is extended to paths by defining $c(P) = \min_{e \in P} c(e)$. During the implementation of an edge deletion or insertion, the $c$-values may temporarily have too small values. We say that $v$ and $w$ are *c-2-edge connected on level $i$* if they are connected and $c(v \cdots w) \geq i$. Assuming that all $c$-values are updated, we have our basic 2-edge connectivity query:

**2-edge-connected**$(v, w)$**:** Decides if $v$ and $w$ are c-2-edge connected on level 0.

Further note that with updated $c$-values, $e \in F$ is a bridge in $G_i$ if and only if $c(e) < i$. For basic updates of $c$-values, we need

**InitTreeEdge**$(v, w)$**:** $c(v, w) := -1$.

**Cover**$(v, w, i)$**:** Where $v$ and $w$ are connected. For all $e \in v \cdots w$, if $c(e) < i$, set $c(e) := i$.

**Uncover**$(v, w, i)$**:** Where $v$ and $w$ are connected. For all $e \in v \cdots w$, if $c(e) \leq i$, set $c(e) := -1$.

We can now compute $c$-values correctly by first calling InitTreeEdge$(v, w)$ for all tree edges $(v, w)$, and then calling Cover$(q, r, \ell(q, r))$ for all non-tree edges $(q, r)$. Inserting an edge is straightforward:

**Insert**$(v, w)$**:** If the end-points of $(v, w)$ were not connected in $F$, $(v, w)$ is added to $F$ and InitTreeEdge$(v, w)$ is called. Otherwise set $\ell(v, w) := 0$ and call Cover$(v, w, 0)$. Clearly (i') is not violated in either case.

In connection with deletion, the basic problem is to deal with the deletion of a non-tree edge. If a non-bridge tree edge $(v, w)$ is to be deleted, we first swap it with a non-tree edge as described in Swap below. The sub-routine **FreeTreeEdge** is dummy for now, but is included so that Swap can be reused directly for the biconnectivity problem.

**Swap**$(v, w)$**:** Where $(v, w)$ is a tree-edge which is not a bridge. Let $(x, y)$ be a non-tree edge covering $(v, w)$ with

$\ell(x, y) = c(v, w) = i$, and set $\ell(v, w) := i$. Call FreeTreeEdge$(v, w)$. Replace $(v, w)$ by $(x, y)$ in $F$. Call InitTreeEdge$(x, y)$ and Cover$(v, w, i)$.

To see that the above updates the cover information, note that it is only the edges being swapped whose covering is affected. We are now ready to describe delete.

**Delete$(v, w)$:** If $(v, w)$ is a bridge, we simply delete it. If $(v, w)$ is a tree edge, but not a bridge, we call Swap$(v, w)$. Thus, if $(v, w)$ is not a bridge, we are left with the problem of deleting a non-tree edge $(v, w)$ on level $i = \ell(v, w)$. Now call Uncover$(v, w, i)$ and delete the edge $(v, w)$. This may leave some c-values on $v \cdots w$ to low and thus for $i = \ell(v, w), \ldots, 0$, we call Recover$(v, w, i)$.

**Recover$(v, w, i)$:** We divide into two symmetric phases. Set $u := v$ and let $u$ step through the vertices of $v \cdots w$ towards $w$. For each value of $u$, consider, one at the time, the non-tree edges $(x, y)$ with $meet(x, v, w) = u$ and $\forall e \in u \cdots x, c(e) \geq i$. If legal, increase the level of $(x, y)$ to $i + 1$ and call Cover$(x, y, i + 1)$. Otherwise, we call Cover$(x, y, i)$ and stop the phase.
If the first phase was stopped, we have a second symmetric phase, starting with $u = w$, and stepping through the vertices in $w \cdots v$ towards $v$.

The problem in seeing that the above algorithm is correct, is to check that the calls to Recover computes the correct $c$-values on $v \cdots w$. We say that $v \cdots w$ is *fine* on level $i$ if all $c$-values in $F$ are correct, except that $c$-values $< i$ on $v \cdots w$ may be too low. Clearly, $v \cdots w$ is fine on level $\ell(e) + 1$ when we make the first call Recover$(v, w, \ell(v, w))$. Thus, correctness follows if we can prove

**Lemma 7** *Assuming that $v \cdots w$ is fine on level $i + 1$. Then after a call Recover$(v, w, i)$, $v \cdots w$ is fine on level $i$.*

**Proof:** First note that we do not violate $v \cdots w$ being fine on level $i + 1$ if we take a level $i$ edge $(x, y)$ and either call Cover$(x, y, i)$ directly, or first increase the level to $i + 1$, and then call Cover$(x, y, i + 1)$.

Given that $v \cdots w$ remains fine on level $i + 1$, to prove that it gets fine on level $i$, we need to show that for any remaining level $i$ non-tree edge $(x, y)$, all edges $e$ in $x \cdots y$ have $c(e) \geq i$. In particular, it follows that $v \cdots w$ does become fine on level $i$ if phase 1 runs through without being stopped.

Now, suppose phase 1 is stopped. Let $u_1$ be the last value of $u$ considered, and $(x_1, y_1)$ be the last edge considered, thus increasing the level of $(x_1, y_1)$ is illegal. Then phase 2 will also stop, for otherwise, it would end up illegally increasing the level of $(x_1, y_1)$. Let $u_2$ be the last value of $u$ considered, and let $(x_2, y_2)$ be the last edge considered in phase 2.

Since the phases were not interrupted for non-tree edges $(x, y)$ covering edges $u$ before $u_1$ or after $u_2$, we know that if $(x, y)$ remains on level $i$, it is because $x \cdots y \cap v \cdots w \subseteq u_1 \cdots u_2$. Hence, we prove fineness of level $i$, if we can show that all $c$-values in $u_1 \cdots u_2$ are $\geq i$.

For $k := 1, 2$, from the illegality of increasing the level of $(x_k, y_k)$, it follows that the 2-edge connected component $C_k$ of $x_k$ in $G_{i+1} \cup \{(x_k, y_k)\}$ has $> \lfloor n/2^{i+1} \rfloor$ nodes. However, we know that before the deletion of $(v, w)$, $C_1$ and $C_2$ where both part of a 2-edge connected component $D$ of $G_i$, and this component had at most $\lfloor n/2^i \rfloor$ nodes. Hence $C_1 \cap C_2 \neq \emptyset$. Thus, they are contained in the same 2-edge connected component $C$ of $G_{i+1} \cup \{(x_1, y_1), (x_2, y_2)\}$. Since covering is done for all level $i + 1$ edges, it follows that our calls Cover$(x_1, y_1, i)$ and Cover$(x_2, y_2, i)$ imply that all tree-edges in $C$ has got $c$-values $\geq i$. Moreover $u_k \in C_k$, so $u_1 \cdots u_2 \subseteq C$, and hence all edges in $u_1 \cdots u_2$ have $c$-values $\geq i$. □

After the last call Recover$(v, w, 0)$, we now know that $v \cdots w$ is fine on level 0, that is, all $c$-values in $F$ are correct, except that $c$-values $< 0$ on $v \cdots w$ may be too low. However, since $-1$ is the smallest value, we conclude that all $c$-values are correct, and hence our fully dynamic 2-edge-connectivity algorithm is correct.

### 4.2 Implementation

#### 4.2.1 Top-trees

In order to efficiently process information concerning paths in $F$, we shall use a variant from [1] of Frederickson's topology trees [5]. The original topology trees are defined for ternary trees which can then be used to encode trees of unbounded degrees. This is often quite technical, so instead we use a variant from [1], called *top-trees*, which works directly for trees of unbounded degree, and which gives rise to much fewer cases. For our purposes, top-trees are also easier to use than the dynamic trees of Sleator and Tarjan [17].

The top-tree is a data structure for dynamic trees that allows simple divide and conquer algorithms. The basic idea is to maintain a balanced binary tree $\mathcal{T}$ representing a recursive subdivision of the tree $T$ into *clusters*, which are subtrees of $T$ that are connected to the rest of $T$ through at most two *boundary nodes*. Each leaf of $\mathcal{T}$ represents a unique edge of $T$ and each internal node of $\mathcal{T}$ represents the cluster that is the union of the clusters represented by its children.

The set of boundary nodes of a given cluster $C$ is denoted $\partial C$, and a node in $C \setminus \partial C$ is called an *internal node* of $C$. If $\partial C = \{a, b\}$ then the path $a \cdots b$ is called the *cluster path* of $C$ and is denoted $\pi(C)$. If $a \neq b$ then the cluster is called a *path-cluster*. The cluster $C$ is said to be a *path-ancestor* of the cluster $A$ and $A$ is called a *path-descendant* of $C$ if they are both path-clusters and $\pi(A) \subseteq \pi(C)$. If $C$ is also the parent of $A$ then $A$ is called a *path-child* of $C$. If $a$ is a boundary node of $C$ and $C$ as two children $A$ and $B$, then $A$ is considered *nearest* to $a$ if $a \notin B$ or if $\partial A = \{a\}$. If $\partial C = \partial A = \partial B = \{a\}$, the nearest cluster is chosen arbitrarily (see figure 1).

As a slight generalization from the above description we may have up to two *external boundary nodes* for each top-tree $\mathcal{T}$. These nodes are considered boundary nodes of any cluster in which they appear. In particular, they are the only boundary nodes of the root cluster of $\mathcal{T}$.

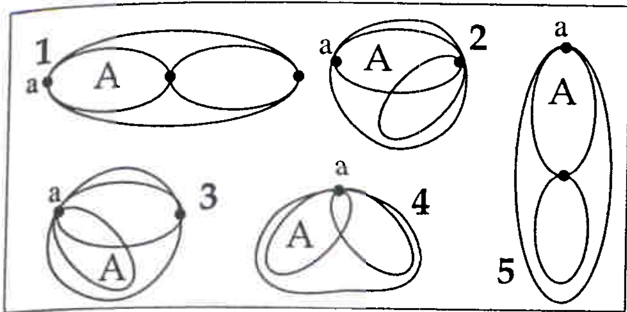


Figure 1: *The 5 cluster compositions. Cluster A is nearest to a. In 4 the choice of nearest is arbitrary.*

The top-tree supports the following update operations:

**Link**$(v, w)$**:** Where $v$ and $w$ are in different top-trees $\mathcal{T}_v$ and $\mathcal{T}_w$. Creates a single new top-tree $\mathcal{T}$ representing $\mathcal{T}_v \cup \mathcal{T}_w \cup \{(v, w)\}$.

**Cut**$(e)$**:** Removes the edge $e$ from the top-tree $\mathcal{T}$ containing it, thus separating the endpoints of $e$.

**Expose**$(v, w)$**:** Makes $v$ and $w$ external boundary nodes of the tree $\mathcal{T}$ containing them and returns the new root cluster.

Every update of the top-tree can be implemented as a sequence of the following two operations:

**Merge**$(A, B, S)$**:** Where $A$ and $B$ are the root-clusters of two top-trees $\mathcal{T}_A$ and $\mathcal{T}_B$, $A \cup B$ is a cluster, $(\partial A \cup \partial B) \setminus (\partial A \cap \partial B) \subseteq S \subseteq \partial A \cup \partial B$ and $|S| \leq 2$. Creates a new cluster $C = A \cup B$ with $\partial C = S$ and makes it the common root of $A$ and $B$, thus turning $\mathcal{T}_A$ and $\mathcal{T}_B$ into a single new top-tree $\mathcal{T}$ with (possibly external) boundary nodes $S$.

**Split**$(C)$**:** Where $C$ is the root-cluster of a top-tree $\mathcal{T}$ and has children $A$ and $B$. Deletes $C$, thus turning $\mathcal{T}$ into the two top-trees $\mathcal{T}_A$ and $\mathcal{T}_B$.

**Theorem 8** **([1, 5])** *We can maintain a top-tree of height $O(\log n)$ supporting each of the operations Link, Cut and Expose, using a sequence of at most $O(\log n)$ Merges and Splits per operation. In addition this sequence can be computed in $O(\log n)$ time.*

Note that since the height of any top-tree is $O(\log n)$, we have that an edge is contained in at most $O(\log n)$ clusters. A node is internal to at most $O(\log n)$ clusters, and we assume pointers from each node to the unique smallest cluster it is internal to.

To illustrate the power of our machinery, we now give a short proof of a result from [17]:

**Corollary 9** *We can maintain a fully dynamic forest $F$ and support queries about the maximum weight between any two nodes in $O(\log n)$ time per operation.*

**Proof:** For each path-cluster $C$ we maintain the maximum weight $W_C$ on the cluster path. Then $C$ :=Merge$(A, B, S)$ sets $W_C := \max\{W_D | D \in \{A, B\}$ is a path-cluster$\}$, while Split$(C)$ just deletes $C$. Both operations take constant time. To answer the query MaxWeight$(v \cdots w)$ we just call $C$ :=Expose$(v, w)$ and return $W_C$. □

#### 4.2.2 2-edge connectivity by top-trees

The algorithm maintains the spanning forest in a top-tree data structure. For each cluster $C$ we maintain $c_C = c(\pi(C))$. Thus, 2-edge connectivity queries are implemented by:

**2-edge-connected**$(v, w)$**:** Set $C$ :=Expose$(v, w)$. Return $(c_C \geq 0)$.

In connection with Swap, for a given tree edge $(v, w)$, we need a covering edge $e$ with $\ell(e) = c(v, w)$. This is done, by maintaining for each cluster $C$ a non-tree edge $e_C$ covering an edge on $\pi(C)$ with $\ell(e_C) = c_C$. Then the desired edge $e$ is found by setting $C$ :=Expose$(v, w)$ and returning $e_C$. Calls to cover and uncover also reduces to operations on clusters:

**Cover**$(v, w, i)$**:** Set $C$ :=Expose$(v, w)$. Call Cover$(C, i, (v, w))$.

**Uncover**$(v, w, i)$**:** Set $C$ :=Expose$(v, w)$. Call Uncover$(C, i)$.

The point is, of course, that we cannot afford to propagate the cover/uncover information the whole way down to the edges. When these operations are called on a path-cluster $C$, we will implement them directly in $C$, and then store *lazy information* in $C$ about what should be propagated down in case we want to look at the descendants of $C$. The precise lazy information stored is

- $c_C^+$, $c_C^-$ and $e_C^+$, where $c_C^+ \leq c_C^-$ and $\ell(e_C^+) = c_C^+$. This represents that for all edges $e \in \pi(C)$, if $c(e) \leq c_C^-$, we should set $c(e) := c_C^+$ and $e(e) := e_C^+$.

The lazy information has no effect if $c_C^+ = c_C^- = -1$. Trivially, the cover information in a root cluster is always correct in the sense that there cannot be any relevant lazy information above it. Moreover, note that the lazy cover information only effects $\pi(C)$, hence only path descendants of $C$. Thus, the cover information is always correct for all non-path clusters.

In order to guide Recover, we need two things: first we need to find the level $i$ non-tree edges $(q, r)$, second we need to find out if increasing the level of $(q, r)$ to $i+1$ will create a too large level $i+1$ component. Thus, we introduce counters **size** and **incident** that are further defined so as to facilitate efficient local computation of all of Cover, Uncover, Split, and Merge.

- For any node $v$ and any level $i$, let $\text{size}_{v,i} := 1$ and let $\text{incident}_{v,i}$ be the number of level $i$ non-tree edges with an endpoint in $v$.
- Let $i$ and $j$ be levels, and let $v$ be a boundary node of a path-cluster $C$. Let $X_{v,C,i,j}$ be the set of internal nodes from the cluster $C$ that are reachable from $v$ by a path $P$ where $c(P \cap \pi(C)) \geq i$ and $c(P \setminus \pi(C)) \geq j$. Then $\text{size}_{v,C,i,j} = (\sum_{w \in X_{v,C,i,j}} \text{size}_{w,i})$ is the number of nodes in $X_{v,C,i,j}$ and $\text{incident}_{v,C,i,j} = (\sum_{w \in X_{v,C,i,j}} \text{incident}_{w,i})$ is the number of (directed) level $j$ non-tree edges $(q, r)$ with $q \in X_{v,C,i,j}$. By directed we mean that $(q, r)$ is counted twice if $r$ is also in $X_{v,C,i,j}$.
- Similarly for any level $i$ and any non-path cluster $C$ with $\partial C = \{v\}$ let $X_{v,C,i}$ be the set of internal nodes $q$ from $C$ such that $c(v \cdots q) \geq i$. Then $\text{size}_{v,C,i} = (\sum_{w \in X_{v,C,i}} \text{size}_{w,i})$ is the number of nodes in $X_{v,C,i}$ and

$\text{incident}_{v,C,i} = (\sum_{w \in X_{v,C,i}} \text{incident}_{w,i})$ is the number of (directed) level $i$ non-tree edges $(q,r)$ with $q \in X_{v,C,i}$.

We are now ready to implement all the different procedures:

**Cover**$(C,i,e)$**:** If $c_C < i$, set $c_C := i$ and $e_C := e$. If $i < c_C^+$, do nothing. If $c_C^- \geq i \geq c_C^+$, set $c_C^+ := i$ and $e_C^+ := e$. If $i > c_C^-$, set $c_C^- := i$ and $c_C^+ := i$ and $e_C^+ := e$. For $X \in \{\text{size,incident}\}$ and for all $-1 \leq j \leq i$ and $-1 \leq k \leq L$ and for $v \in \partial C$ set $X_{v,C,j,k} := X_{v,C,-1,k}$.

**Uncover**$(C,i)$**:** If $c_C \leq i$, set $c_C := -1$ and $e_C :=$ **nil**. If $i < c_C^+$, do nothing. If $i \geq c_C^+$, set $c_C^+ := -1$ and $c_C^- := \max\{c_C^-, i\}$ and $e_C^+ :=$**nil**. For $X \in \{\text{size,incident}\}$ and for all $-1 \leq j \leq i$ and $-1 \leq k \leq L$ and for $v \in \partial C$ set $X_{v,C,j,k} := X_{v,C,i+1,k}$.

**Clean**$(C)$**:** For each path-child $A$ of $C$, call Uncover$(A, c_C^-)$ and Cover$(A, c_C^+, e_C^+)$. Set $c_C^+ := -1$ and $c_C^- := -1$ and $e_C^+ :=$**nil**.

**Split**$(C)$**:** Call Clean$(C)$. Delete $C$.

**Merge**$(A, B, \{a\})$**:** Where $a \in \partial A$. Create a parent $C$ of $A$ and $B$ with $\partial C = \{a\}$. Let $c$ be the node in $\partial A \cap \partial B$.
For $X \in \{\text{size,incident}\}$ and for $j := 0, \ldots, L$: If $A$ is a non-path cluster, set $X_{a,C,j} := X_{a,A,j} + X_{a,B,j}$. Otherwise set $X_{a,C,j} := X_{a,A,j,j}(+X_{c,j} + X_{c,B,j} \text{ if } c_A \geq i)$.

**Merge**$(A, B, \{a,b\})$**:** Where $a \in \partial A$ and $b \in \partial B$. Create a parent $C$ of $A$ and $B$ with $\partial C = \{a, b\}$. Let $c$ be the node in $\partial A \cap \partial B$. Let $D$ be the path-child of $C$ minimizing $c_D$, then set $c_C := c_D$ and $e_C := e_D$. Set $c_C^+ := -1$ and $c_C^- := -1$ and $e_C^+ :=$**nil**. For $X \in \{\text{size,incident}\}$ and for $i,j := -1, \ldots, L$ compute $X_{a,C,i,j}$ as follows ($X_{b,C,i,j}$ is symmetrical): If $A$ is a non-path cluster, set $X_{a,C,i,j} := X_{a,A,j} + X_{a,B,i,j}$. Otherwise if $B$ is a non-path cluster, set $X_{a,C,i,j} := X_{a,A,i,j}(+X_{c,B,j} \text{ if } c_A \geq i)$. Finally if both $A$ and $B$ are path-clusters, set $X_{a,C,i,j} := X_{a,A,i,j}(+X_{c,j} + X_{c,B,i,j} \text{ if } c_A \geq i)$.

**Recover**$(v, w, i)$**:**

- For $u := v, w$
  - Set $C$ :=Expose$(v, w)$.
  - While $\text{incident}_{u,C,-1,i} + \text{incident}_{u,i} > 0$ and not stopped,
    - Set $(q,r)$ :=Find$(u, C, i)$.
    - $D$ :=Expose$(q, r)$.
    - If $\text{size}_{q,D,-1,i} + 2 > n/2^i$,
      - Cover$(D, i, (q,r))$.
      - Stop the while loop.
    - Else
      - Set $\ell(q,r) := i+1$, decrement $\text{incident}_{q,i}$ and $\text{incident}_{r,i}$ and increment $\text{incident}_{q,i+1}$ and $\text{incident}_{r,i+1}$.
      - Cover$(D, i+1, (q,r))$.
    - $C$ :=Expose$(v, w)$.

**Find**$(a, C, i)$**:** If $\text{incident}_{a,i} > 0$ then return a non-tree edge incident to $a$ on level $i$. Otherwise call Clean$(C)$ and let $A$ and $B$ be the children of $C$ with $A$ nearest to $a$. If $A$ is a non-path cluster and $\text{incident}_{a,A,i} > 0$ or $A$ is a path cluster and $\text{incident}_{a,A,-1,i} > 0$, then return find$(a, A, i)$. Else, let $b$ be the boundary node nearest to $a$ in $B$, return find$(b, B, i)$.

**Theorem 10** *There exists a deterministic fully dynamic algorithm for maintaining 2-edge connectivity in a graph, using $O(\log^4 n)$ amortized time per operation.*

**Proof:** Cover$(C,i,e)$ and Uncover$(C,i)$ both take $O(\log^2 n)$ time. This means that Clean$(C)$ and thus Split$(C)$ takes $O(\log^2 n)$ time. Since Merge$(A,B,S)$ also takes $O(\log^2 n)$ time we have by theorem 8 that Link$(v,w)$, Cut$(e)$ and Expose$(v,w)$ takes $O(\log^3 n)$ time. This again means that FindCoverEdge$(v,w)$, 2-edge-connected$(v,w)$, Cover$(v \cdots w, i, e)$ and Uncover$(v \cdots w, i)$ take $O(\log^3 n)$ time. Find$(a,C,i)$ calls Clean$(C)$ $O(\log n)$ times and thus takes $O(\log^3 n)$ time. Finally Recover$(v,w,i)$ takes $O(\xi \log^3 n)$ time where $\xi$ is the number of non-tree edges whose level is increased. Since the level of a particular edge is increased at most $O(\log n)$ times we spend at most $O(\log^4 n)$ time on a given edge between its insertion and deletion. □

## 5 Biconnectivity

In this section we present an $O(\log^4 n)$ deterministic algorithm for the biconnectivity problem for a fully dynamic graph $G$. We will follow the same pattern as was used for 2-edge connectivity. Historically, such a generalization is difficult. For example, it took several years to get sparsification to work for biconnectivity [3, 13]. Furthermore, the generalization in [9] of the $O(\log^5 n)$ randomized 2-edge connectivity algorithm from [11] has an expected bound of $O(\Delta \log^4 n)$, where $\Delta$ is the maximal degree [Henzinger, pc 1997]. Our main new idea for preserving the $O(\log^4 n)$ bound for biconnectivity, is an efficient recycling of the information as described in Lemma 12 below.

A *triple* is a length two path $xyz$ in the graph $G$, and a *tree triple* $xyz$ in $F$ is said to be *covered* by a non-tree edge $(v,w)$ if $xyz \subseteq v \cdots w$, that is if $xyz$ is a segment of the cycle induced by $(v,w)$. Covered triples are also *transitively covered*, and if $xyz$ and $x'yz$ are transitively covered, then so is $xyx'$. An *articulation point* is a vertex whose removal disconnect a component of $G$.

**Lemma 11** *$v$ is an articulation point if and only if there is an uncovered tree triple $uvw$. Moreover, $v$ and $w$ are biconnected if and only if for all $xyz \subseteq v \cdots w$, $xyz$ is transitively covered.*

### 5.1 High-level

As with 2-edge connectivity, with each non-tree edge $e$, we associate a level $\ell(e) \in \{0, \ldots, L\}$, $L = \lfloor \log_2 n \rfloor$, and for each $i$, we let $G_i$ denote the subgraph of $G$ induced by edges of level at least $i$ together with the edges of $F$. Thus $G = G_0 \supseteq G_1 \supseteq \ldots \supseteq G_L \supseteq F$. Here, for biconnectivity, we will maintain the invariant:

(i") The maximal number of nodes in a biconnected component of $G_i$ is $\lfloor n/2^i \rfloor$.

As for 2-edge connectivity, the invariant is satisfied initially, by letting all non-tree edges have level 0. We say that it is *legal* to increase the level of a non-tree edge $e$ to $j$ if this

does not violate (i"), that is, if the biconnected component of $e$ in $G_j \bigcup\{e\}$ has at most $\lfloor n/2^j \rfloor$ vertices.

For each vertex $v$ and each level $i$, we implicitly maintain the disjoint sets of neighbors biconnected on level $i$. If $u$ is a neighbor of $v$, the set of neighbors of $v$ biconnected to $u$ on level $i$ is maintained as $c_{v,i}(u)$. As for 2-edge connectivity, the $c$-values may temporarily not be fully updated. If $P$ is a path in $G$, $c(P)$ denotes the maximal $i$ such that for all triples $xyz \subseteq P$, $z \in c_{y,i}(x)$. If there is no such $i$, $c(P) = -1$. Thus $c(P) \geq i$ witnesses that the end points of $P$ are biconnected on level $i$. Typically $P$ will be a tree path, but in connection with Recover, we will consider paths where the last edge $(q, r)$ is a non-tree edge. We say that $v$ and $w$ are *c-biconnected on level* $i$ if they are connected and $c(v \cdots w) \geq i$. If all $c$-values are updated, we therefore have

**biconnected**$(v, w)$: Decides if $v$ and $w$ are $c$-biconnected on level 0.

To update $c$-values from scratch, we need

**InitTreeEdge**$(v, w)$: For $i := 0, \cdots, L$, set $c_{x,i}(y) := \{y\}$ and $c_{y,i}(x) := \{x\}$.

**FreeTreeEdge**$(v, w)$: Remove $y$ from $c_{x,\cdot}(\cdot)$ and remove $x$ from $c_{y,\cdot}(\cdot)$.

**Cover**$(xyz, i)$: Where $xyz$ is a tree triple, unions $c_{y,j}(x)$ and $c_{y,j}(z)$ for $j := 0, \ldots, i$.

**Cover**$(v, w, i)$: Calls Cover$(xyz, i)$ for all $xyz \subseteq v \cdots w$.

Now, as for 2-edge connectivity, we can compute all $c$-values by first calling InitTreeEdge$(v, w)$ for all tree edges $(v, w)$, and then calling Cover$(q, r, \ell(q, r))$ for all non-tree edges $(q, r)$. The above routines immediately complete the descriptions of Insert and Swap. In order to describe Delete, we need to define both Uncover and Recover. To do this efficiently, we have to recycle cover information using the following:

**Lemma 12** *Let $(v, w)$ be a level $i$ non-tree edge covering a tree triple $xyz \subseteq v \cdots w$. Suppose $s$ is a neighbor to $y$ biconnected on level $j \leq i$ to $x$, and hence to $y$, and $z$. Then, if $(v, w)$ is deleted, afterwards, $s$ is biconnected on level $j$ to $x$ or $z$, and it may be biconnected to both.*

The lemma suggests, that when $(v, w)$ is deleted, we should store the neighbors $s$ mentioned. This is done in $c_{y,j}(x|z)$ by Uncover. More precisely, $c_{y,j}(x|z)$ will be the set of neighbors to $y$ that we know are biconnected to $x$ or $z$, but that are not yet $c$-biconnected to either. This will be used in one of two ways. Either $x$ and $z$ get $c$-biconnected on level $j$, in which case we just restore $c_{y,j}(x)$ and $c_{y,j}(z)$ by setting $c_{y,j}(x) := c_{y,j}(z) := c_{y,j}(x|z) \cup c_{y,j}(x) \cup c_{y,j}(z)$ and $c_{y,j}(x|z) := \emptyset$. Alternatively, suppose we know we have finished updating $c_{y,j}(x)$ and that $z \notin c_{y,j}(x)$. Then we can set $c_{y,j}(z) := c_{y,j}(x|z) \cup c_{y,j}(z)$ and $c_{y,j}(x|z) := \emptyset$.

**Uncover**$(xyz, i)$: where $xyz$ is a tree triple $c$-biconnected on level $i$, if it is also $c$-biconnected on level $i+1$, do nothing; otherwise, for $j := i, \ldots, 0$, set
$c_{y,j}(x|z) := c_{y,j}(x) \setminus (c_{y,j+1}(x) \cup c_{y,j+1}(z))$,
$c_{y,j}(x) := c_{y,j+1}(x)$, and $c_{y,j}(z) := c_{y,j+1}(z)$.

Strictly speaking, above, we should also set $c_{y,j}(s) := c_{y,j+1}(s)$ for all $s \in c_{y,j}(x|z)$, but our algorithm will never query any subset of $c_{y,j}(x|z)$.

**Uncover**$(v, w, i)$: Calls Uncover$(xyz, i)$ for all $xyz \subseteq v \cdots w$.

**Cover**$(xyz, i)$: Where $xyz$ is a tree triple. For $j = 0, \ldots, i$, if $c_{y,j}(x|z) \neq \emptyset$, union $c_{y,j}(x)$, $c_{y,j}(z)$, and $c_{y,j}(x|z)$, and set $c_{y,j}(x|z) := \emptyset$. Otherwise, union $c_{y,j}(x)$ and $c_{y,j}(z)$, subtracting them from any $c_{y,j}(\cdot|\cdot)$ they might appear in.

To complete the description of Delete, we need to define Recover.

**Recover**$(v, w, i)$: We divide into two symmetric phases. Phase 1 goes as follows:

Set $u := v$ and let $u'$ be the successor of $u$ in $v \cdots w$.

(*) While there is a level $i$ non-tree edge $(q, r)$ such that $u = meet(q, v, w)$ and $c(u'\, u \cdots q\, r) \geq i$, if legal, increase the level of $(q, r)$ to $i+1$ and call Cover$(q, r, i+1)$; otherwise, just call Cover$(q, r, i)$ and stop Phase 1.

While $\exists u'uu'' \subseteq v \cdots w$ with $c_{y,j}(x|z) \neq \emptyset$,

Let $u'uu''$ be such a triple nearest to $v$.
Run (*) again with the new values of $u$ and $u'$.
Union $c_{u,j}(u'')$ and $c_{u,j}(u'|u'')$, and set $c_{u,j}(u'|u'') := \emptyset$.
Run (*) again with $u''$ in place of $u'$.

If Phase 1 was stopped in (*), we have a symmetric Phase 2, which is the same except that we start with $u = w$ and in the loop choose the triple $u'uu'' \subseteq w \cdots v$ nearest to $w$.

The proof of correctness is essentially the same as for 2-edge connectivity. As a small point, note that different biconnected components may overlap in one vertex. Nevertheless, we cannot have two different biconnected components with $> \lfloor n/2^{i+1} \rfloor$ nodes whose combined size is $\leq \lfloor n/2^i \rfloor$.

Note that at the end of Recover$(v, w, j)$, all sets $c_{y,j}(x|z)$, $xyz \subseteq v \cdots w$, will be empty. Hence, for each $y$, there can be at most one pair $x$ and $z$ with $c_{y,j}(x|z) \neq \emptyset$. Then we refer to $x$ and $z$ as the *uncovered neighbors* of $y$.

### 5.2 Implementation

The main difference between implementing biconnectivity and 2-edge connectivity, is that we need to maintain the biconnectivity of the neighbors of all vertices efficiently. For each vertex $y$, we will maintain $c_{y,\cdot}(\cdot)$ as a list with weights on the links between succeeding elements such that $c(xyz)$ is the minimum weight of a link between $x$ and $z$ in $c_{y,\cdot}(\cdot)$. Then $c_{y,i}(x)$ is a segment of $c_{y,\cdot}(\cdot)$ and using standard techniques for manipulating lists, we can easily find $c(xyz)$ or identify $c_{y,i}(x)$ in time $O(\log n)$.

Now, if $c_{y,j-1}(x) = c_{y,j-1}(z)$, we can union $c_{y,j}(x)$ and $c_{y,j}(z)$ without affecting $c_{y,j-1}(x)$, simply by *moving $c_{y,j}(z)$ to $c_{y,j}(x)$ on level $j$* as follows. First we *extract* $c_{y,j}(z)$, replacing it by the minimal link to its neighbors. Since both of these links are at most $j-1$, this does not affect the minimum weight between elements outside $c_{y,j}(z)$. Second we *insert* $c_{y,j}(z)$ after $c_{y,j}(x)$ with link $j$ in between. The link after $c_{y,j}(z)$ becomes the link we had after $c_{y,j}(x)$. Note that if $x \in c_{u,\cdot}(u'|u'')$ and we move $c_{u,j}(x)$ to $c_{u,j}(u')$,

then, implicitly, we delete $c_{u,j}(x)$ from $c_{u,j}(u'|u'')$, as required.

**InitTreeEdge**$(v, w)$: Link $w$ to $c_{v,\cdot}(\cdot)$ on level -1 and $v$ to $c_{w,\cdot}(\cdot)$ on level -1.

**FreeTreeEdge**$(v, w)$: Extract $w$ from $c_{v,\cdot}(\cdot)$ and $v$ from $c_{w,\cdot}(\cdot)$.

**Cover**$(xyz, i)$: Where $xyz$ is a tree triple. For $j = 0, \ldots, i$, if $x$ and $z$ are uncovered neighbors of $y$ and $c_{y,j}(x|z) \neq \emptyset$, move $c_{y,j}(x|z)$ and $c_{y,j}(z)$ to $c_{y,j}(x)$. Else, if $x$ is an uncovered neighbor of $y$, move $c_{y,j}(z)$ to $c_{y,j}(x)$. Else move $c_{y,j}(x)$ to $c_{y,j}(z)$.

**Uncover**$(xyz, i)$: where $c(xyz) \geq i$, if $c(xyz) > i$, do nothing; otherwise, for $j := i, \ldots, 0$, first extract $c_{y,j}(x)$ and set $c_{y,j}(x|z) := c_{y,j}(x)$. Then move $c_{y,j+1}(x)$ and $c_{y,j+1}(z)$ back to the neighbor list $c_{y,\cdot}(\cdot)$ on level -1.

**Biconnectivity by top-trees** As for 2-edge connectivity, the algorithm maintains the spanning forest in a top-tree data structure. For each cluster $C$ we maintain $c_C = c(\pi(C))$.

**Biconnected**$(v, w)$: Set $C$ :=Expose$(v, w)$. Return $(c_C \geq 0)$.

Also, $e_C$, $c_C^+$, $c_C^-$, and $e_C^+$ are defined analogously to in 2-edge connectivity. The cover edges $e_C$ and $e_C^+$ are exactly the same, while $c_C^+$ and $c_C^-$, like $c_C$, now refer to covering of triples instead of edges.

A main new idea is that we overrule the top-trees by using the neighbor lists $c_{y,\cdot}(\cdot)$ to propagate information from minimal non-path clusters to path clusters. Recall that in 2-edge, the information in non-path clusters is never missing any lazy information. Let $v$ be the boundary node of a path cluster $C$, and let $w$ be any neighbor to $v$ in $C \setminus \pi(C)$. Then we call $w$ a *cluster neighbor of* $v$. It is easy to see that there is then a non-path cluster $A \subseteq C$ with $\{v\} = \partial A$ and $w \in A$. We call the minimal such cluster $A$ the *neighbor cluster of* $(v, w)$, and denote it $NC(v, w)$. Note that the ordering of $v$ and $w$ matters. It is easy to see that there cannot be another $(v, w')$ with $NC(v, w') = NC(v, w)$. Hence, for any neighbor cluster $NC(v, w)$, we can uniquely talk about the *neighbor edge* $(v, w)$. We are going to use the neighbor lists to propagate counters directly from neighbor clusters to the minimal path clusters containing them, skipping all non-path clusters in between.

We are now ready for the rather delicate definitions of the counters **size** and **incident** for path clusters and neighbor clusters.

- Let $j$ and $k$ be levels, and let $C$ be a path-cluster with $\partial C = \{v, w\}$. Let size$_{v,C,j,k}$ denote the number of internal nodes $q$ of $C$ such that either $q \in \pi(C)$ and $c(v \cdots q) \geq i$ or there exist a triple $u'uu'' \subseteq \pi(C)$ with $u = meet(v, w, q)$ and $(u, x) \in u \cdots q$ such that $c(v \cdots u) \geq j$, $c(u \cdots q) \geq k$ and either $c(u'ux) \geq k$ or $c(u'uu'') \geq j$ and $x \in c_{u,k}(u'') \cup c_{u,k}(u'|u'')$. Let incident$_{v,C,j,k}$ be the number of (directed) non-tree edges $(q, r)$ with the path $v \cdots qr$ satisfying the conditions from above for the path $v \cdots q$.
- Similarly let $k$ be a level and let $C$ be a neighbor cluster with neighbor edge $(v, w)$. Let size$_{v,C,k}$ be the number of internal nodes $q$ of $C$ such that $c(vw \cdots q) \geq k$, and let incident$_{v,C,k}$ be the number of (directed) non-tree edges $(q, r)$ where $q$ is an internal node of $C$ and $c(vw \cdots qr) \geq k$.

To get from neighbor clusters to path clusters, and vice versa, we need the following functions:

**Size**$(v, W, i)$: where $W$ is a set of neighbors of $v$, returns $\sum_{w \in W}$(Size$_{v,NC(v,w),i}$ if $w$ cluster neighbor of $v$, 0 otherwise).

**Incident**$(v, W, i)$: where $W$ is a set of neighbors of $v$, returns $\sum_{w \in W}$(1 if $w$ non-tree neighbor of $v$, Incident$_{v,NC(v,w),i}$ if $w$ cluster neighbor of $v$, and 0 otherwise).

**Neighbor**$X(u, u', i)$: $X \in$ {**Size, Incident**}, $X(u, c_{u,i}(u'), i)$

**Neighbor**$X(u, u' \vee u'', i)$: $X \in$ {**Size, Incident**}, $X(u, c_{u,i}(u') \cup c_{u,i}(u'') \cup c_{u,i}(u'|u''), i)$.

**NeighborFind**$(u, u', i)$: Finds $z \in c_{u,i}(u')$ such that $z$ is either a non-tree neighbor of $u$ or a cluster neighbor with incident$_{u,NC(u,z),i} > 0$.

Whenever the counters of a neighbor cluster $NC(v, w)$ are updated, we update corresponding counters of $w$ in the neighbor list $c_{v,\cdot}(\cdot)$ of $v$. Standard list data structures allow us, in $O(\log n)$ logarithmic time, to update any one of the $2L$ counters of a neighbor, or to answer a query Neighbor$X$. The remaining operations are implemented analogously to in 2-edge connectivity.

**Cover**$(C, i, e)$: First we do as in 2-edge connectivity. If $C$ has path children $A$ and $B$ and $\{u\} = \partial A \cap \partial B \not\subseteq \partial C$ and $u'uu''$ is the triple with $u' \in A$ and $u'' \in B$, then we call Cover$(u'uu'', i)$.

**Uncover**$(C, i)$: First we do as in 2-edge connectivity. If $C$ has path children $A$ and $B$ and $\{u\} = \partial A \cap \partial B \not\subseteq \partial C$ and $u'uu''$ is the triple with $u' \in A$ and $u'' \in B$, then we call Uncover$(u'uu'', i)$.

**Merge**$(A, B, \{a\})$: Where $a \in \partial A$. Create a parent $C$ of $A$ and $B$ with $\partial C = \{a\}$. If $A$ is a non-path cluster, we are done. Otherwise, let $u'uu''$ be the unique triple such that $u' \in \pi(A)$, $\{u\} = \partial A \cap \partial B$ and $B$ is the neighbor cluster of $(u, u'')$. Then for $X \in$ {size,incident} and $k := -1, \ldots, L$, $X_{a,C,k} := X_{a,A,k,k}$ if $c_A < k$, $X_{a,C,k} := X_{a,A,k,k}$+Neighbor$X(u, u', k)$ if $c_A \geq k \wedge c(u'uu'') < k$, and finally $X_{a,C,k} := X_{a,A,k,k}$+Neighbor$X(u, u' \vee u'', k) + X_{u,B,k}$ if $c_A \geq k \wedge c(u'uu'') \geq k$. Let $a'$ be the successor of $a$ in $\pi(A)$. Then $C = NC(a, a')$, so we have to update the $2L$ counters associated with $a'$ in $a$'s neighbor list $c_{a,\cdot}(\cdot)$.

**Merge**$(A, B, \{a, b\})$: Where $a \in \partial A$ and $b \in \partial B$. Create a parent $C$ of $A$ and $B$ with $\partial C = \{a, b\}$. $c_C, e_C, c_C^+, c_C^-$ and $e_C^+$ are maintained as in 2-edge connectivity. For $X \in$ {size,incident} and $j, k := -1, \ldots, L$ compute $X_{a,C,j,k}$ as follows ($X_{b,C,j,k}$ is symmetrical): If $A$ is a non-path cluster, set $X_{a,C,j,k} := X_{a,B,j,k}$. Otherwise if $B$ is a non-path cluster, set $X_{a,C,j,k} := X_{a,A,j,k}$. Finally if both $A$ and $B$ are path-clusters, let $u'uu''$ be the triple such that $u' \in \pi(A)$, $\{u\} = \partial A \cup \partial B$, and $u'' \in \pi(B)$. Then $X_{a,C,j,k} := X_{a,A,j,k}$ if $c_A < j$,
$X_{a,C,j,k} := X_{a,A,j,k}$+Neighbor$X(u, u', k)$ if $c_A \geq j \wedge c(u'uu'') < j$, and finally
$X_{a,C,j,k} := X_{a,A,j,k}$+Neighbor$X(u, u' \vee u'', k) + X_{u,B,j,k}$ if $c_A \geq j \wedge c(u'uu'') \geq j$.

**Recover**$(v, w, i)$: We divide into two symmetric phases. Phase 1 goes as follows:

Set $C$ :=Expose$(v, w)$.

Set $u := v$ and let $u'$ be the successor of $u$ on $u \cdots w$.

(*) While NeighborIncident$(u, u', i) > 0$,

- Set $(q, r)$ :=VertexFind$(u, C, i, u')$.
- $D$ :=Expose$(q, r)$.
- Let $(q, q')$ and $(r', r)$ be edges on $q \cdots r$
- If size$_{q,D,-1,i}$ + 2+ NeighborSize$(q, q', i)$+ NeighborSize$(r, r', i) > n/2^i$,
  - Cover$(D, i, (q, r))$.
  - Stop the phase.
- Else
  - Set $\ell(q, r) := i + 1$, updating the corresponding incidence counters $c_{q,\cdot}(\cdot)$ and $c_{r,\cdot}(\cdot)$.
  - Move $c_{q,i+1}(r)$ to $c_{q,i+1}(q')$ and $c_{r,i+1}(q)$ to $c_{r,i+1}(r')$ on level $i + 1$.
  - Cover$(D, i + 1, (q, r))$.
- $C$ :=Expose$(v, w)$.

$u$ :=FindBranch$(v, C, i)$.

While $u \neq$ **nil**,

Let $u'$ be the predecessor, and let $u''$ be the successor of $u$ in $v \cdots w$.
Run (*) again with the new values of $u$ and $u'$.
Move $c_{y,j}(x|z)$ to $c_{y,j}(z)$ and set $c_{y,j}(x|z) := \emptyset$.
Run (*) again with $u''$ in place of $u'$.
$u$ :=FindBranch$(v, C, i)$.

If Phase 1 was stopped in (*), we have a symmetric Phase 2 with the roles of $v$ and $w$ interchanged.

**FindBranch**$(a, C, i)$: If incident$_{a,C,-1,i} = 0$ return **nil** else call Clean$(C)$. If $C$ has only one path-child $a$ then return FindBranch$(a, A, i)$. Otherwise let $A$ and $B$ be the children of $C$ with $A$ nearest to $a$ and let $u'uu''$ be the triple such that $u' \in \pi(A)$ and $u'' \in \pi(B)$ and $u \in \partial A \cup \partial B$. If incident$_{a,A,-1,i} > 0$ then return FindBranch$(a, A, i)$. Otherwise if $c_{u,i}(u'|u'') \neq \emptyset$ then return $u$ else return FindBranch$(u, B, i)$.

**VertexFind**$(u, C, i, u')$: Call Clean$(C)$. Let $z$ :=NeighborFind$(u, u', i)$. If $z$ is a non-tree neighbor, return $(u, z)$. Otherwise $z$ is a cluster neighbor and then $NC(u, z)$ has two children $A$ and $B$ with $u \in A$, $A \cap B = \{b\}$. If incident$(u, A, i, i) > 0$, return PathFind$(u, A, i)$. Otherwise, return VertexFind$(b, B, i, b')$ where $b'$ is the predecessor of $b$ in $u \cdots b$.

**PathFind**$(a, C, i)$: Call VertexFind$(a, C, i, a')$ where $a'$ is the successor of $a$ on $a \cdots b$. If no edge was returned, let $A$ and $B$ be the children of $C$ with $A$ nearest to $a$. If incident$_{a,A,-1,i} > 0$ then return PathFind$(a, A, i)$. Else let $b$ be the boundary node nearest to $a$ in $B$, return PathFind$(b, B, i)$. If no edge was found return VertexFind$(b, C, i, b')$, where $b'$ is the predecessor of $b$ on $a \cdots b$.

**Theorem 13** *There exists a deterministic fully dynamic algorithm for maintaining biconnectivity in a graph, using $O(\log^4 n)$ amortized time per operation.*

## References


[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 270–280, 1997.

[2] D. Eppstein. Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:237–250, 1995.

[3] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification — a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, September 1997. See also FOCS'92.

[4] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, January 1981.

[5] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985. See also STOC'83.

[6] G. N. Frederickson. Ambivalent data structures for dynamic 2-Edge-Connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26(2):484–538, April 1997. See also FOCS'91.

[7] M. Fredman and M. R. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. Technical Report TR95-1523, Cornell University, Computer Science, 1995. To appear in *Algorithmica*. See also STOC'94 [16].

[8] M. R. Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, June 1995. See also FOCS'92.

[9] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th IEEE Symp. Foundations of Computer Science*, pages 664–672, 1995.

[10] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing*, pages 519–527, 1995.

[11] M. R. Henzinger and V. King. Fully dynamic 2-edge connectivity algorithm in polygarithmic time per operation. Technical Report SRC 1997-004a, Digital, 1997. A preliminary version appeared as [10].

[12] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 594–604, 1997.

[13] M. R. Henzinger and H. La Poutré. Certificates and fast algorithms for biconnectivity in fully-dynamic graphs. In *Proc. 3rd European Symp. Algorithms, LNCS 979,*, pages 171–184, 1995.

[14] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: With applictions to fully dynamic graph algorithms. *Random Structures and Algorithms*, 11:369–379, 1997. See also ICALP'96.

[15] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130(1):203–236, 1994.

[16] M. Rauch. Improved data structures for fully dynamic biconnectivity. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, may 1994.

[17] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. of Computer and System Sciences*, 26:362–390, 1983.

[18] R. E. Tarjan. Efficiency of a good but not linear set union algorithms. *J. Assoc. Comput. Mach.*, 22:215–225, 1975.

[19] M. Thorup. Decremental dynamic connectivity. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 305–313, 1997.

[20] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.