Technical Report DIKU-TR-97/26
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 KBH Ø
DENMARK

November 1997

Poly-Logarithmic Deterministic Fully-Dynamic Graph Algorithms II: 2-edge and biconnectivity

Jacob Holm Kristian de Lichtenberg Mikkel Thorup

Poly-logarithmic deterministic fully-dynamic graph algorithms II: 2-edge and biconnectivity

Jacob Holm* Kristian de Lichtenberg* Mikkel Thorup*

Abstract

Deterministic fully dynamic algorithms are presented for 2-edge connectivity and biconnectivity. For 2-edge connectivity the amortized cost per operation is $O(\log^4 n)$ improving over the previous best deterministic bound of $O(\sqrt{n})$ and the previous best randomized bound of $O(\log^5 n)$.

For biconnectivity the amortized cost per operation is also $O(\log^4 n)$ improving over the previous best deterministic bound of $O(\sqrt{n \log n} \log\lceil m/n \rceil)$ and the alternative randomized bound of $O(\Delta \log^4 n)$ where Δ is the maximal degree. Thus our $O(\log^4 n)$ bound is the first polylogarithmic bound for biconnectivity.

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

1 Introduction

We consider the fully dynamic graph problems of 2-edge connectivity and biconnectivity. A graph is *connected* if there is a path between any two vertices. The *components* of a graph are the maximal connected subgraphs. 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 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. In the *fully dynamic 2-edge connectivity problem*, the updates may be interspersed with queries asking whether two given vertices are 2-edge connected. Similarly for the *fully dynamic biconnectivity problem*, queries are whether two given vertices are biconnected.

In this paper, we give deterministic algorithms solving the fully dynamic 2-edge and biconnectivity problem in $O(\log^4 n)$ amortized time per operation. It should be noted that biconnectivity has a history of being much harder than 2-edge connectivity, and that the biconnectivity result is considered the main contribution of this paper.

Relating to previous work In 1991 [5], Fredrickson succeeded in generalizing his $O(\sqrt{m})$ bound from 1983 [4] for fully dynamic connectivity to fully dynamic 2-edge connectivity. In 1992–1993 [3, 2], this was improved by Eppstein, Galil, Italiano, and Nissenzweig to $O(\sqrt{n})$. In 1995–1997 [7, 8], these bounds were improved to $O(\log^5 n)$ expected amortized time per operation, generalizing the randomized $O(\log^3 n)$ bound for connectivity from [7]. In 1996 [10], Henzinger and Thorup improved the randomized connectivity bound to $O(\log^2 n)$ but the improvement did not affect the randomized $O(\log^5 n)$ bound for 2-edge-connectivity. Here we present a deterministic fully dynamic 2-edge-connectivity algorithm with amortized operation cost $O(\log^4 n)$. Our algorithm is a careful generalization of a recent $O(\log^2 n)$ deterministic fully dynamic connectivity algorithm [11].

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 [9]. In 1994 [12], Rauch improved this bound to $O(\min\{\sqrt{m}\log n, n\})$. In 1995, (Rauch) Henzinger and Poutré further improved the deterministic bound to $O(\sqrt{n \log n}\log\lceil m/n\rceil)$.

In 1995 [6], Henzinger and King generalized their randomized algorithm from [7] to the biconnectivity problem to achieve an $O(\Delta \log^4 n)$ expected amortized cost per operation, where Δ is the maximal degree (In [6], the bound is incorrectly quoted as $O(\log^4 n)$ [Henzinger, personal communication, 1997]). Generalizing our approach for 2-edge connectivity, we present a deterministic fully dynamic biconnectivity algorithm with an amortized cost per operation of $O(\log^4 n)$. This is the first polylogarithmic bound for the problem, even when we include randomized algorithms.

Techniques We shall follow the general connectivity approach from [11] of amortizing cost over increases of internal edge levels, making sure that components induced by edges on level i or higher are of size at most $n/2^i$. Also, we follow the strategy from [6, 7] of organizing our information around some spanning forest. In [11], the amortization worked very simply for connectivity, and a decremental minimum spanning tree algorithm followed as a direct specialization. Generalizing to 2-edge connectivity and biconnectivity is a lot more subtle. The details are very different and the secrets are in the details, including two log-factors. In particular for biconnectivity, we need to make a careful recycling of information, leading to the first polylogarithmic algorithm for this problem.

2 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. 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. Our solution to the 2-edge connectivity also serves as a general framework for solving the more complicated biconnectivity problem. Most of the highest level routines will carry over directly, so to solve the biconnectivity problem, we will only have to replace certain subroutines.

We will maintain a spanning forest F of G, and the edges in F will be referred to as *tree edges*. If v and w are connected in F, $v \cdots w$ denotes the simple path from v to w in F. If they further are connected to u, $meet(u, v, w)$ denotes the intersection vertex between 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). Hence e is a bridge if and only if it is not covered by any non-tree edge. Thus two vertices x and y are 2-edge connected if and only if there is a path of covered edges between them.

2.1 High level description

Internally, the algorithm associates with each non-tree edge e a level $\ell(e) \leq L = \lfloor \log_2 n \rfloor$. 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 $n/2^i$. Thus, the maximal relevant level is L.

Initially, all non-tree edges have level 0, and hence the invariant is satisfied. The point in the levels is that we will amortize our work over increases in the levels of non-tree edges. 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 $n/2^j$ 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 1 *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 in phase 2, and (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 $> n/2^{i+1}$

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 $n/2^i$ 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.

2.2 Implementation

2.2.1 Top-trees

In order to efficiently process information concerning paths in F, we shall use a variant from [1] of Frederickson's topology trees [4]. 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 [13].

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 ∂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 $\wp(C)$. 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 $\wp(A) \subseteq \wp(C)$. 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 on page 18).

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 $\mathcal{T}$.

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 2 ([1, 4]) *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 [13]:

Corollary 3 *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 Merge(A, B, S) has to assign $W_C := \max\{W_D | D \in \{A, B\} \text{ is a path-cluster}\}$ to the new root-cluster C, 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. □

2.2.2 2-edge connectivity by top-trees

The algorithm maintains the spanning forest in a top-tree data structure. For each cluster C we maintain $c_C = c(\wp(C))$. 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 $\wp(C)$ 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 $e \in \wp(C)$, 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 $\wp(C)$, 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 size$_{v,i}$:= 1 and let 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 $c(P \cap \wp(C)) \geq i$ and $c(P \setminus \wp(C)) \geq j$. Then 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 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 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 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 $j := -1, \ldots, L$: If A is a non-path cluster, set $X_{a,C,j} := X_{a,A,j} + X_{a,B,j}$. Otherwise set $X_{a,C,j} := X_{a,A,j,j}(+X_{c,j} + X_{c,B,j}$ 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 $X_{a,C,i,j} := X_{a,A,i,j}(+X_{c,B,j}$ if $c_A \geq i)$. Finally if both A and B are path-clusters, set $X_{a,C,i,j} := X_{a,A,i,j}(+X_{c,j} + X_{c,B,i,j}$ if $c_A \geq i)$.

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

```
• For $u := v, w$
    – Set $C$ :=Expose$(v, w)$.
    – While incident$_{u,C,-1,i}$+incident$_{u,i} > 0$ and not stopped,
        ∗ Set $(q, r)$ :=Find$(u, C, i)$.
        ∗ $D$ :=Expose$(q, r)$.
        ∗ If 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 incident$_{q,i}$ and incident$_{r,i}$ and
              increment incident$_{q,i+1}$ and incident$_{r,i+1}$.
            · Cover$(D, i + 1, (q, r))$.
        ∗ $C$ :=Expose$(v, w)$.
```

Find(a, C, i)**:** If 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 incident$_{a,A,i} > 0$ or A is a path cluster and 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 4 *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 2 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 ξ 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. $\square$

3 Biconnectivity

In this section we present an $O(\log^4 n)$ deterministic algorithm for the biconnectivity problem for a fully dynamic graph G.

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'.

Lemma 5 *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.*

3.1 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:

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

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 (ii), that is, if the biconnected component of e in $G_j \bigcup \{e\}$ has at most $n/2^j$ 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 update 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 6 *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 $> n/2^{i+1}$ nodes whose combined size is $\leq n/2^i$.

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 $c_{y,j}(x|z) \neq \emptyset$, and then we refer to x and z as the *uncovered neighbors* of y.

3.2 Implementation

The main difference between 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 $c_C = \wp(C)$.

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 $C \setminus \wp(C)$. 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 $\text{size}_{v,C,j,k}$ denote the number of internal nodes q of C such that either $q \in \wp(C)$ and $c(v \cdots q) \geq i$ or there exist a triple $u'uu'' \subseteq \wp(C)$ 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 $\text{incident}_{v,C,j,k}$ 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 $\text{size}_{v,C,k}$ be the number of internal nodes q of C such that $c(vw \cdots q) \geq k$, and let $\text{incident}_{v,C,k}$ 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 $\sum_{w \in W}(\text{Size}_{v,NC(v,w),i}$ 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, $\text{Incident}_{v,NC(v,w),i}$ if w cluster neighbor of v, and 0 otherwise).

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

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

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 $\text{incident}_{u,NC(u,z),i} > 0$.

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 $u' \in \wp(A)$, $\{u\} = \partial A \cap \partial B$ and B is the neighbor cluster of (u, u''). Then for $X \in \{\text{size,incident}\}$ 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 $\wp(A)$. 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 $X \in \{\text{size,incident}\}$ 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 $u' \in \wp(A)$, $\{u\} = \partial A \cup \partial B$, and $u'' \in \wp(B)$. 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 $u' \in \wp(A)$ and $u'' \in \wp(B)$ 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 7 *There exists a deterministic fully dynamic algorithm for maintaining biconnectivity in a graph, using $O(\log^4 n)$ amortized time per operation.* $\square$

It turns out that we can preserve the $O(\log^4 n)$ amortized update time while reducing the query time for "are v and w biconnected" to $O(\log n)$ worst case time. This is, however, beyond the scope of the current report.

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] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Improved sparsification. Technical Report 93-20, Univ. of California, Irvine, Dept. Information and Computer Science, 1993.

[3] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. In *Proc. 33rd Symp. Foundations of Computer Science*, pages 60–69. IEEE, 1992.

[4] Greg 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.

[5] Greg 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.

[6] 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.

[7] 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.

[8] 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 [7].

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

[10] Monika Rauch Henzinger and Mikkel Thorup. Improved sampling with applications to dynamic graph algorithms. In *Proceedings of the 23rd International Colloquium on Automata Languages, and Programming (ICALP), LNCS 1099*, pages 290–299, 1996.

[11] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic graph algorithms I: connectivity and minimum spanning tree. Technical Report DIKU-TR-97/17, University of Copenhagen, Dept. of Computer Science, 1997. Also submitted to STOC'98.

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

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

A Illustration of cluster compositions

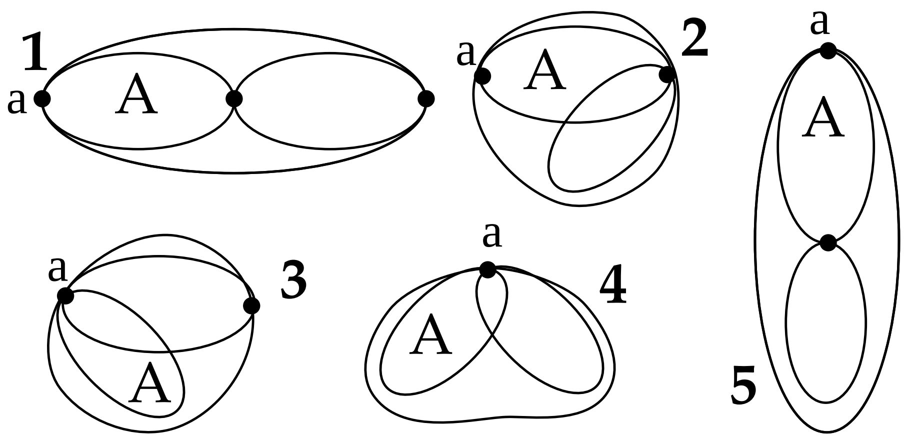

Figure 1: The 5 cluster compositions. Cluster A is nearest to a. In 4 the choice of nearest is arbitrary.

B Top-trees implemented through topology trees

For completeness, we sketch the reduction from our top-trees to Frederickson's topology trees, giving Theorem 2 from [1]. It should be noted that in practice it is much more efficient to implement our top-trees directly.

Let T be a dynamic tree for which we wish to maintain a top-tree $\mathcal{T}$. This may be done directly, in a manner very similar to the maintainance of the topology trees, but the proof is rather technical. Here we show how to maintain $\mathcal{T}$ by a simple reduction to the topology trees by Frederickson [4, 5].

The basic idea of the reduction is to maintain a ternary tree T_Δ for T, where each node represents a (possibly empty) edge-cluster of T. Using the algorithm from Frederickson [5] we then maintain a topology tree $\mathcal{T}_\Delta$ for T_Δ. Finally we show that every node of $\mathcal{T}_\Delta$ represents a cluster in T and thus $\mathcal{T}$ is maintained.

Let T_Δ be a tree derived from T as follows:

- For every edge $e \in T$, T_Δ contains a node t_e representing the edge-cluster $\{e\}$.
- For every node $v \in T$ having edges $e_1, \ldots, e_d$, T_Δ contains:
  - nodes $v_{e_1}, \ldots, v_{e_d}$ representing empty edge-clusters.
  - edges $(v_{e_1}, t_{e_1}), \ldots, (v_{e_d}, t_{e_d})$.
  - edges $(v_{e_1}, v_{e_2}), \ldots, (v_{e_{d-1}}, v_{e_d})$ if $d > 1$.

T_Δ has $3n-3$ nodes and any link or cut in T corresponds to a constant number of links and cuts in T_Δ. Thus by using the algorithm from [5] we can maintain a topology tree $\mathcal{T}_\Delta$ for T_Δ. The desired result then follows by this lemma:

Lemma 8 *Any node-cluster in $\mathcal{T}_\Delta$ corresponds to an edge-cluster of T.*

Proof: From [5] we know that the node-clusters of $\mathcal{T}_\Delta$ have the following property: Either the external degree of the cluster (the number of edges with exactly one endpoint in the cluster) is ≤ 2 or the cluster consists of a single node.

This means that if the external degree is ≤ 2 then the node-cluster trivially corresponds to an edge-cluster of T. Otherwise the cluster consists of a single node of degree 3 and all nodes in T_Δ of degree 3 represent empty edge-clusters. $\square$

It should be noted that Frederickson has no equivalent to our Expose, which hence does not follow from the reduction. Implementing Expose is, however, very similar to implementing cut and link.