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, $O(\log^4 n)$ for minimum spanning forest, 2-edge connectivity, and $O(\log^5 n)$ biconnectivity.

1 Introduction

We consider the fully dynamic graph problems of connectivity, minimum spanning forest, 2-edge connectivity and biconnectivity. Here by *fully dynamic* we mean that the graph may be *updated* by insertion and deletion of edges. If we only allow insertions or only allow deletions, the graph is only *partially dynamic*. The updates are interspersed with *queries* to the current graph. The update and query *operations* are presented on-line, with no knowledge of future operations.

Like priority queues, dynamic graph algorithms may both be of direct interest, and of interest as data structures within algorithms for static problems. As an example of direct usage, Frederickson [7] suggests using a dynamic minimum spanning tree algorithm to maintain how heavily loaded links we need to use to get from one vertex to another in a communications networks. As an example of usage as data structures, Gabow, Kaplan, and Tarjan [11] use dynamic 2-edge connectivity to efficiently determine if a graph has a unique matching.

We will now formally define the problems considered and state our results. We are considering a fully dynamic graph G over a fixed vertex set V, $|V| = n$. Unless otherwise stated, we assume that we start with an empty edge set. Most of the time bounds presented are *amortized*, meaning that they are averaged over all operations performed. This is particularly justified when our fully dynamic algorithms are used as data structures inside static algorithms where we only care about the total running time. We are striving for time bounds that are polylogarithmic in n. Here polylogarithmic bounds are considered feasible for dynamic problems in the same way as polynomial bounds are considered feasible for static problems.

For the *fully dynamic connectivity problem*, the updates may be interspersed with *connectivity queries*, asking whether two given vertices are connected in G. The connectivity problem reduces

---

*A preliminary version of this work was presented at the *30th ACM Symposium on the Theory of Computing (STOC'98)* [21]. The conference version mistakenly claimed a $O(\log^4 n)$ instead of $O(\log^5 n)$ for biconnectivity.

[†]Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Kbh. Ø, Denmark. E-mail: `(samson,morat)@diku.dk`.

[‡]AT&T Labs–Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA. E-mail: `mthorup@research.att.edu`. Some of this work was done while the author was at the University of Copenhagen.

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 the dynamic trees of Sleator and Tarjan [25], 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 F in $O(\log^4 n)$ amortized time per operation. Applying the dynamic trees technique from [25] to the minimum spanning forest F, in $O(\log n/\log\log n)$ time, we can for any pair of vertices find the heaviest edge between them in F, which is also the heaviest edge needed to get between the vertices in G.

A *bridge* in a graph is an edge whose removal disconnects some component. A graph is *2-edge connected* if and only if it is connected and contains no bridges. 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. In the *fully dynamic 2-edge connectivity problem*, the edge updates may be interspersed with queries asking whether two given vertices are 2-edge connected. We present a deterministic algorithm supporting all operations in $O(\log^4 n)$ amortized time per operation. The algorithm is easily augmented with searches for bridges.

An *articulation point* in a graph is a vertex whose removal disconnects some component. A graph is *biconnected* if and only if it is connected and has no articulation points. 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 biconnectivity problem*, the edge updates may be interspersed with queries asking whether two given vertices are biconnected. We present a deterministic algorithm supporting all operations in $O(\log^5 n)$ amortized time per operation. The algorithm is easily augmented with searches for articulation points.

1.1 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 [7], Fredrickson 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. [5] improved the update time to $O(\sqrt{n})$ using the *sparsification technique*. Finally in 1997 Henzinger and King [16] gave an algorithm with $O(\sqrt[3]{n}\log n)$ amortized update time and constant time per connectivity query.

In 1995 [17], Henzinger and King used randomization to get the first feasible solution to the dynamic connectivity problem. They 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 $O(\log^2 n)$ in 1996 [19] by Henzinger and Thorup. No randomized technique was known for improving the deterministic $O(\sqrt[3]{n}\log n)$ amortized update cost for the minimum spanning tree problem.

In 1991 [8], Fredrickson succeeded in generalizing his $O(\sqrt{m})$ bound from 1983 [7] for fully dynamic connectivity to fully dynamic 2-edge connectivity. As for connectivity, the sparsification technique of Eppstein et.al. [5] 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 [15, 17]. It should be noted that the above mentioned improvement for connectivity of Henzinger and Thorup [19], 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 [12]. In 1994 [13], Rauch improved this bound to $O(\min\{\sqrt{m}\log n, n\})$. In 1995, (Rauch) Henzinger and Poutré further improved the deterministic bound to $O(\sqrt{n \log n}\log\lceil m/n\rceil)$ [18]. In 1995 [14], Henzinger and King generalized their randomized algorithm from [17] to the biconnectivity problem to achieve an $O(\Delta \log^4 n)$ expected amortized cost per operation, where Δ is the maximal degree at the moment the operation is performed (In [14], the bound is incorrectly quoted as $O(\log^4 n)$ [Henzinger, personal communication, 1997]).

Finally, for all of the above problems, there is a lower bound of $\Omega(\log n/\log\log n)$, proved independently by Fredman and Henzinger [10] and Miltersen, Subramanian, Vitter, and Tamassia [24].

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 [27]. Westbrook and Tarjan have obtained the same time bound for incremental 2-edge and biconnectivity [32]. Further Sleator and Tarjan have provided an $O(\log n)$ bound [25] 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 [29]. For decremental minimum spanning tree and biconnectivity, no better bounds were known than those for the fully dynamic case.

1.2 New contributions

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 that was used by Henzinger and King [17] 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 algorithm with an amortized cost of $O(\log^2 n)$ per operation for any sequence of $\Omega(m)$ deletions. Then we use a technique from [16] 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 an $O(\log^4 n)$ operation cost for 2-edge connectivity and an $O(\log^5 n)$ operation cost for biconnectivity. The generalization uses some of the ideas from [8, 14, 15] 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.

1.3 Implications

Using known reductions, our results imply improved fully dynamic algorithms for bipartiteness, k-edge witness, and maximal spanning forest decomposition [17], for geometric minimum spanning trees [4], and for approximate edge connectivity [31].

Our algorithms may also be used as improved subroutines in algorithms for the several static problems: randomly sampling spanning forests of a given graph [6], finding a color-constrained minimum spanning tree [9], and finding a consensus tree [20]. Very recently, our dynamic 2-edge connectivity has been used in providing efficient implementations of old constructive proofs in matching theory [2, 11].

1.4 Resent developments

Iyer et al. [22] have implemented and compared our connectivity algorithm with other fully-dynamic connectivity algorithms, and in these experiments, variants of our algorithm performed very well. Thorup [30] has found a linear space implementation of the fully-dynamic connectivity algorithm of this paper, which here is implemented in $O(m + n \log n)$ space. Also he improved the randomized amortized update time to $O(\log n(\log\log n)^2)$, thus getting very close the above mentioned cell-probe lower bound of $\Omega(\log n/\log\log n)$ [10, 24]. Finally, using bit parallelism as well as a kind of biased deletions, he has improved the time bounds for 2-edge and biconnectivity to $O(\log^3 n/\log\log n)$.

1.5 Contents

First we have a preliminary Section 2, reviewing notation and known tools for dealing with dynamic trees. Readers who are only interested in the general ideas of our dynamic connectivity algorithm can skip this preliminary section. Afterwards, we present the fully dynamic connectivity algorithm in Section 3, the generalization to decremental minimum spanning forest in Section 4, the fully dynamic minimum spanning forest algorithm in Section 5, the fully dynamic 2-edge connectivity algorithm in Section 6, and the fully dynamic biconnectivity algorithm in Section 7. Our presentation is focussed on getting good polylogarithmic bounds for all operations. In some cases, using more complicated algorithms, one can get better space and query time, but for ease of presentation, we will only sketch these improvements.

2 Preliminaries

As mentioned, this section can be skipped by readers only interested in the high level ideas of our dynamic connectivity algorithm.

In all the dynamic problems considered in this paper, we will be maintaining some spanning forest of the graph. If v and w are connected in our dynamic forest, $v \cdots w$ denotes the unique path from v to w. If $v = w$, $v \cdots w$ is just the vertex v. If $v \neq w$, $s^w(v)$ denotes the successor of v on the path $v \cdots w$. If u, v, and w are all connected, $meet(u, v, w)$ denotes the unique intersection vertex of the three paths $u \cdots v$, $u \cdots w$, and $v \cdots w$.

We will now review some data structures needed for maintaining our spanning forests. The first is very simple and suffices for connectivity and decremental minimum spanning trees. The second is more complicated, but is needed for our implementations of fully dynamic minimum spanning trees, 2-edge and biconnectivity. The data structures are themselves rooted trees so to keep things apart, *nodes* and *arcs* are in the data structure while *vertices* and *edges* are in the spanning forest.

2.1 ET-trees

We will now discuss the *ET-trees* of Henzinger and King [17]. We work on a dynamic forest where arbitrary edges can be cut and edges linking different trees in the forest can be inserted. A query connected(v, w) tells whether v and w are connected, and a query size(v) gives the number of vertices in the tree containing v. The forest can further be updated by adding or removing weighted keys from the vertices. A query min-key(v) returns a minimal key from the tree containing v, if any. If the keys are unweighted, min-key(v) returns an arbitrary key. In our connectivity and decremental minimum spanning algorithm, the keys will typically be incident non-tree edges.

All the above updates and queries are supported in $O(\log n)$ time using the ET-trees from [17], to which the reader is referred for a more detailed description. An ET-tree is a standard dynamic balanced binary tree over some Euler tour around a tree in the forest. Here an Euler tour around a tree is a maximal closed walk over the graph obtained from the tree replacing each edge by a directed edge in each direction. The walk uses each directed edge once so if T has n vertices, the cyclic Euler tour has length $2n - 2$. We will have such an ET-tree for each tree in our forest. The important point is that if trees in the 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 $O(\log n)$ ET-nodes.

Each vertex in our dynamic forest may occur several times in the Euler tour. Arbitrarily we select one of these occurrences as the representative. Now each ET-node represents the set of representative leaves below it. Let ET-root(v) denote the ET-tree root over v. Since the balanced ET-trees have height $O(\log n)$, we can find ET-root(v) in time $O(\log n)$. Now connected$(v, w) \iff$ ET-root$(v) =$ ET-root(w).

At each ET-node q, we maintain the number size(q) of representatives below it and the minimal key min-key(q) attached to a representative below it. Since links and cuts only affect $O(\log n)$ ET-nodes and since the ET-trees have height $O(\log n)$, this information is easily maintained in $O(\log n)$ time per update. Now, for a forest vertex v, size$(v) =$ size(ET-root(v)) and min-key$(v) =$ min-key(ET-root(v)).

Finally, as in [17], we note that if we are willing to settle for $O(\log^2 n/\log\log n)$ time for links and cuts, we can reduce the cost of the other operations to $O(\log n/\log\log n)$. The simple trick is instead of the balanced binary trees to use balanced $\Theta(\log n)$-ary trees over the Euler tours. Now, the height is reduced to $O(\log n/\log\log n)$, but link and cuts affect $O(\log n/\log\log n)$ ET-nodes each with $O(\log n)$ children.

2.2 Top trees

The ET-trees are very simple to implement, but they fail to maintain information about paths in trees, such as for example, what is the maximal weight on the path between two given vertices in a tree. Typically, a path will be completely spread over an Euler tour of a tree. In order to deal efficiently with paths, we shall use the top trees from [1].

A top tree is defined based on a pair consisting of a tree T and a set ∂T of at most 2 vertices from T, called *external boundary vertices*. Given $(T, \partial T)$, any connected subtree C of T has a set $\partial_{(T,\partial T)}C$ of *boundary vertices* which are the vertices of C that are either in ∂T or incident to an edge in T leaving C. The subtree C is called a *cluster* of $(T, \partial T)$ if it has at most two boundary vertices. Then T is itself a cluster with $\partial_{(T,\partial T)}T = \partial T$. Also, if A is a subtree of C, $\partial_{(C,\partial_{(T,\partial T)}C)}A = \partial_{(T,\partial T)}A$, so A is a cluster of $(C, \partial_{(T,\partial T)}C)$ if and only if A is a cluster of $(T, \partial T)$. Since $\partial_{(T,\partial T)}$ is a canonical generalization of ∂ from T to all subtrees of T, we will use ∂ as a shorthand for $\partial_{(T,\partial T)}$ in the rest

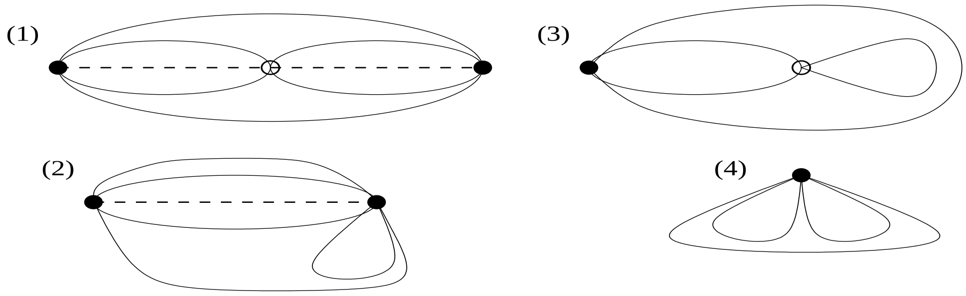

Figure 1: The cases of merging two neighboring clusters into one. The • are the boundary vertices of the merged cluster and the ∘ are the boundary vertices of the children clusters that did not become boundary vertices of the merged cluster. Finally the dashed line is the cluster path of the merged cluster.

of the paper. We say two clusters A and B are *neighbors* if they share a single vertex and $A \cup B$ is a cluster (see Figure 1).

A *top tree* $\mathcal{T}$ over $(T, \partial T)$ is a binary tree such that:

1. The nodes of $\mathcal{T}$ are clusters of $(T, \partial T)$.
2. The leaves of $\mathcal{T}$ are the edges of T.
3. If C is the parent of A and B in $\mathcal{T}$ then $C = A \cup B$ and A and B are neighbors.
4. The root of $\mathcal{T}$ is T itself.

For a cluster C, the vertices in $C \setminus \partial C$ are called *internal vertices.* If a and b are the (not necessarily distinct) boundary vertices of C, the path $a \cdots b$ is called the *cluster path* of C and is denoted $\pi(C)$. If $a \neq b$, the cluster is called a *path cluster.* The cluster C is said to be a *path ancestor* of the cluster D and D is called a *path descendant* of C if they are both path clusters and $\pi(D) \subseteq \pi(C)$. Note that each edge $e \in \pi(C)$ is a path descendant of C. A child which is a path descendant is a *path child*, so in Figure 1, we have 2 path children in (1), 1 path child in (2), and 0 path children in (3-4). If a is a boundary vertex of C and C has 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.

The top trees over the trees in our forest are maintained under the following operations:

Link(v, w) where v and w are in different trees, links these trees by adding the edge (v, w) to our dynamic forest.

Cut(e) removes the edge e from our dynamic forest.

Expose(v, w) returns **nil** if v and w are not in the same tree. Otherwise, it makes v and w external boundary vertices of the top tree containing them and returns the new root cluster.

Every update of the top trees is implemented as a sequence of the following two local operations:

Merge(A, B) where A and B are neighbor clusters and roots of two top trees $\mathcal{T}_A$ and $\mathcal{T}_B$. Creates a new cluster $C = A \cup B$ with children A and B, thus combining $\mathcal{T}_A$ and $\mathcal{T}_B$ in a top tree with root C. Finally, the new root cluster C is returned.

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 $\mathcal{T}_B$.

We will always start with sequence of Split. This includes a Split of all ancestor clusters of edges whose boundary change. Note that an end-point v of an edge has to be boundary vertex of the edge if v is not a leaf in the underlying forest, so each of Link, Cut, and Expose can change the boundary of at most two edges, excluding the edge being linked or cut, if any. Finally, we finish with a sequence of Merge.

Theorem 1 ([1, 7]) *For a dynamic forest we can maintain top trees of height $O(\log n)$ supporting each Link, Cut, or Expose with a sequence of $O(\log n)$ Split and Merge. Here the sequence itself is identified in $O(\log n)$ time. The space usage of the top trees is linear in the size of the dynamic forest.*

Note that since the height of any top tree maintained using this theorem is $O(\log n)$, we have that an edge is contained in at most $O(\log n)$ clusters. A vertex v of degree d can appear in $O(d \log n)$ clusters, but v is only internal to $O(\log n)$ clusters, and we assume a pointer $C(v)$ to the unique smallest cluster it is internal to. If v is an external boundary vertex, it is not internal to any cluster, and then $C(v)$ points to the root cluster containing v.

We refer to the algorithm of Theorem 1 which translates Link, Cut, and Expose operations into sequences of Merge and Split operations as the *top driver*. When using top trees, we have direct access to its representation, which is just a standard binary tree, whose nodes represent the clusters, and with each "top" node is associated a set of at most two boundary vertices. As users, we will typically associate extra information with the top nodes. Now, when the top driver has merged two clusters A and B into a new cluster C, we will be notified with pointers to the top nodes representing A, B, and C. We can then compute information for C based on the information we have associated with A and B. If C is later split, we may propagate information from C down to A and B. As an example of the power of the top machinery, we give a short proof of a result from [25]:

Corollary 2 *We can maintain a fully dynamic weighted forest F supporting queries about the maximum weight between any two vertices in $O(\log n)$ time per operation.*

Proof: For each path cluster C we maintain the maximum weight on $\pi(C)$ in the variable weight_C. For a path cluster consisting of an edge e, weight_e is just the weight of e. Now $C := \text{Merge}(A, B)$ sets $\text{weight}_C := \max\{\text{weight}_D | D \in \{A, B\} \text{ is a path child of } C\}$, while $\text{Split}(C)$ just deletes C. Both operations take constant time. To answer the query $\text{MaxWeight}(v \cdots w)$ we just set $C := \text{Expose}(v, w)$ and return weight_C. □

As a final observation, we note that it is easy to augment top trees with the following $O(\log n)$ time operation that we shall use in Subsection 5.2.

Find(v) returns a unique identifier for the tree containing v. Thus $\text{Find}(v) = \text{Find}(w)$ if and only if v and w are connected. The identifier is only changed when the tree containing v is changed by Link and Cut. It is not changed by Expose.

The identifiers are just stored at the root clusters, so $\text{Find}(v)$ is implemented by going to $C(v)$ and then move up $O(\log n)$ times till we find a root cluster, from which we return the associated identifier. In connection with $\text{Expose}(v, w)$, we first find and save the identifier of the root cluster containing v and w, then run Expose, and finally store the saved identifier at the new root cluster.

In connection with Link and Cut, we first free the identifiers at the root clusters involved, so that they can be reused. After the Link or Cut, new tree identifiers are allocated for the new root clusters.

3 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.

3.1 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.*

As mentioned in Section 2.1, it is easy to check if vertices are connected in a dynamic forest. Hence insertions are easy: when inserting an edge (v, w), we check if v and w are connected in F; if not, we add (v, w) to F. Also, we can easily deal with deletions of non-tree edges. Our challenge is to deal with the deletion of a tree edge (v, w). The deletion splits some tree in F, but if the corresponding component is not split, we have to find a replacement edge for F.

Internally, our algorithm associates with each edge e a level $\ell(e) \leq \ell_{\max} = \lfloor \log_2 n \rfloor$. For each i, F_i denotes the sub-forest of F induced by edges of level at least i. Thus, $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_{\ell_{\max}}$. The following invariants are maintained.

(i) F is a maximum (w.r.t. ℓ) 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 vertices in a tree in F_i is $\lfloor n/2^i \rfloor$. Thus, the maximal level is $\ell_{\max}$.

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 $\ell_{\max}$ 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. Clearly, neither (i) nor (ii) is violated.

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 intermediate time, 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 vertices as T_v. By (ii), T had at most $\lfloor n/2^i \rfloor$ vertices, so now T_v has at most $\lfloor n/2^{i+1} \rfloor$ vertices. 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).

3.2 Implementation

To implement the above abstract algorithm, for each i, we apply the ET-trees from Subsection 2.1 to the forest F_i. With each vertex, we associate a key for each incident level i edge. The keys for tree edges and for the non-tree edges are separated so that we can search tree edges and non-tree edges independently.

Note that each tree edge on level i appears in all F_h, $h \leq i$, hence in $O(\log n)$ levels. On the other hand, we only have m keys, as each edge only appears as a key on its own level. Hence, our space usage is $O(m + n \log n)$.

It is now straightforward to analyze the amortized cost of the different operations. When an edge is inserted on level 0, the direct cost is $O(\log n)$. However, its level may increase $O(\log n)$ times. For each increase, we spend $O(\log n)$ time, both for finding the edge, and for making the appropriate changes to the ET-trees. Thus, the total amortized cost of inserting an edge, including all subsequent level increases, 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 ET-trees over $F_0 = F$ immediately allows us to answer connectivity queries between arbitrary vertices in time $O(\log n)$. In order to reduce this time to $O(\log n / \log \log n)$, we simply apply the $\Theta(\log n)$-ary ET-trees menioned in Subsection 2.1 to our spanning forest F. This gives us an added cost of $O(\log^2 n / \log \log n)$ time per changes in F, but this is subsumed by our $O(\log^2 n)$ cost from above. Hence we conclude:

Theorem 3 *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.*

As mentioned, our space bound for connectivity $O(m + n \log n)$. The main challenge in getting the space further down is that Replace, for each level i, needs to determine which of T_v and T_w is the smaller. However, Thorup [30] has recently found a quite different linear space implementation of our algorithm.

4 Decremental minimum spanning forests

We will now expand on the ideas from the previous section to the problem of decrementally maintaining a minimum spanning forest (MSF). In the next section, we will apply what is essentially a construction from [16], transforming a deletions-only MSF algorithm into a fully dynamic MSF algorithm.

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, that the initial spanning forest F has to be a minimal spanning forest. The second is that when in replace, 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.

To see that the above simple change suffices 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) Every cycle C has a non-tree edge e with $w(e) = \max_{f\in C} w(f)$ and $\ell(e) = \min_{f\in C}\ell(f)$.

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 4 *Assume (iii) and that F is a minimum spanning forest. Then, 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 a heaviest edge on C_i. Hence e_2 is the unique heaviest non-tree 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 5 *The algorithm maintains (iii), i.e. that every cycle C has a non-tree edge e with $w(e) = \max_{f\in C} w(f)$ and $\ell(e) = \min_{f\in C}\ell(f)$.*

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.

Our real problem is to show that (iii) is preserved during Replace when an edge e either gets its level increased, or becomes a tree edge. We may assume that e is a unique lowest heaviest non-tree edge on some cycle C, for otherwise (iii) cannot get violated. Based on these assumptions we will show (1) that all other edges from C incident to T_v have level $> \ell(e)$, and using (1) we will show (2) that C cannot leave T_v. From (2) we conclude that e cannot be a replacement edge, and from (1) and (2) we conclude that e has strictly lower level than all other edges in C, hence that e still satisfies (iii) when increased.

We will now prove (1). Let i be the level of e. When e is next to be considered by Replace we know that all edges in the tree T_v have level $> i$. Also, we know that any non-tree edge incident to

T_v and strictly lighter than e has level $> i$. Further, since e was lowest on C and since e was the unique heaviest lowest non-tree edge on C, we know that any non-tree edge in C as heavy as e has level $> i$. Thus we conclude (1), that all edges from C incident to T_v have level $> i$.

To prove (2), suppose that C leaves T_v. Then C has at least two edges leaving T_v, one of which is not e. Call this edge f. If f has level $\geq i$, it would be a replacement edge, and from the previous section, we know that there is no replacement edge on level $> i$. Hence $\ell(f) \leq i$, contradicting (1), so we conclude (2), that C never leaves T_v.

As discussed above, (1) and (2) implies that (iii) does not get violated. □

We have now established that our invariants (i), (ii), and (iii) are maintained. Hence, given that we start with a minimum spanning forest, Lemma 4 ascertains that if a tree edge is deleted, it is replaced by a lightest replacement edge. Thus our spanning forest will remain minimal, as desired.

The ET-trees from Subsection 2.1 are already made to return a minimum weight key, which here corresponds to the desired lightest non-tree edge incident to a tree. Hence our time bounds are the same as for connectivity starting with m edges, each with a potential cost of $O(\log^2 n)$. We conclude

Theorem 6 *There exists a deletions-only MSF algorithm that can be initialized on a graph with n vertices and m edges and support any sequence of deletions in $O(m \log^2 n)$ total time.* □

As for connectivity, the space bound is $O(m + n \log n)$.

5 Fully dynamic MSF

To obtain a fully dynamic MSF algorithm we apply a general reduction from a fully dynamic MSF problem to a series of decremental MSF problems. Essentially our reduction is that of Henzinger and King [16, pp. 600-603]. Their reduction requires, however, that the decremental structure can support inserting certain batches of edges while we want to reduce directly to purely decremental MSF problems. To obtain such a reduction, we combine the above mentioned technique of Henzinger and King with a contraction technique of theirs presented in [15]. Our reduction can be formally characterized as follows.

Theorem 7 *Suppose we have a deletions-only MSF algorithm that for any k, l, can be initialized on a graph with k vertices and l edges and support any sequence of 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 vertices starting with no edges, supports m insertions and deletions in amortized time*

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

Combining Theorem 6 and Theorem 7, we conclude

Theorem 8 *There is a fully dynamic MSF algorithm that for a graph with n vertices, starting with no edges, maintains a minimum spanning forest in $O(\log^4 n)$ amortized time per edge insertion or deletion.*

The rest of this section presents a construction proving Theorem 7.

5.1 High level description

We will support insertions via a logarithmic number of decremental MSF structures. When an edge is deleted, it will be deleted from all the decremental structures, and a replacement edge will be sought among the replacement edges returned by these. When an edge is inserted, we will union it with some of the decremental structures into a new decremental structure.

More precisely, besides maintaining a minimum spanning forest F of G, we will maintain a set $\mathcal{A} = \{A_0, \ldots, A_L\}$, $L = \lceil \log_2 m \rceil$, of subgraphs of G, and for each A_i, we will maintain a minimum spanning forest F_i. We will refer to edges of F as *global tree edges* and the edges in the F_i as *local tree edges*. All edges of G will be in at least one A_i, so $F \subseteq \bigcup_i F_i$. Further, we have the following invariant:

(iv) For each global non-tree edge $f \in G \setminus F$, there is exactly one i such that $f \in A_i \setminus F_i$ and if $f \in F_j$, then $j > i$.

The following lemma states that when a global tree edge e is deleted, we can find its replacement by deleting e from all A_i it occurs in.

Lemma 9 *If f is the lightest replacement edge for a global tree edge e, then f is the lightest replacement edge for e in at least one A_i.*

Proof: Since f is not in F yet, by (iv) there is an i such that $f \in A_i \setminus F_i$. When e has been deleted, f is a global tree edge in $G \setminus \{e\}$, but then f must also be a local tree edge in the subgraph $A_i \setminus \{e\} \subseteq G \setminus \{e\}$. □

Before presenting the details of a deletion, we describe insertions

Insert(e) Let v and w be the end points of e. If v and w are not connected by F, we just add e to F. Otherwise, if e is lighter than the heaviest edge f on the path from v to w, we replace f with e in F, and update $\mathcal{A}$ with $\{f\}$ as described below. Finally, if the path from v to w does not contain an edge heavier than e, we update $\mathcal{A}$ with $\{e\}$.

Delete(e) First we delete e from all A_i it appears in. Let R be the set of returned replacement edges. If $e \in F$, we delete e from F. Subsequently, we check R for edges reconnecting F. If R contains any such edges, we pick the lightest such edge f. By Lemma 9, f is the correct replacement edge for e, so we insert f in F. Finally, no matter whether e was a global tree edge or not, we update $\mathcal{A}$ with R as described below.

Update $\mathcal{A}$ with the edge set D First we find the smallest j such that $|(D \cup \bigcup_{h \leq j}(A_h \setminus F_h)) \setminus F| \leq 2^j$. Then we set

$$A_j := F \cup D \cup \bigcup_{h \leq j} (A_h \setminus F_h), \tag{1}$$

initializing A_j as a new decremental MSF structure. Finally we set $A_h := \emptyset$ for all $h < j$.

By definition of j, we have $m \geq |(D \cup \bigcup_{h \leq j}(A_h \setminus F_h)) \setminus F| > 2^{j-1}$, so $j \leq \lceil \log_2 m \rceil = L$, and hence we are not introducing any new A_j. The correctness of Insert and Delete now follows from:

Lemma 10 *Both Insert and Delete restore (iv).*

Proof: The proof divides into two steps.

(a) First we show that before updating $\mathcal{A}$ with D, it is only the edges in D for which (iv) may not be satisfied.

(b) Second we show that updating $\mathcal{A}$ with D establishes (iv) for the edges in D while not destroying (iv) for any other edge.

The two steps together establish the restoration of (iv).

(a) With Insert, since we do not change $\mathcal{A}$ before the update, our only concern is a new global non-tree edge. This is either e if e does not go in F, or an edge f that e replaces in F. In either case $\mathcal{A}$ is subsequently updated with the new global non-tree edge. With Delete, we get no new non-tree edges, so our only concern is edges changing status in some A_i. The edges changing status before the update of $\mathcal{A}$ are exactly the replacement edges in R that we later update with. This completes (a).

(b) We will now first show that the update of $\mathcal{A}$ establishes (iv) for the edges in D. This follows if we can show that when the update starts, the edges of D are not non-tree edges in any A_i. In case of Insert, our concern is if e replaces an edge f which is then used for the update. However, f was a global tree edge, so f must be a local tree edge in any A_i it appears in. In connection with Delete, let $r \in D = R$. Then r was a replacement edge from some A_i, meaning that it was a non-tree edge in A_i. However, by (iv) this means that r is not a non-tree edge in any other A_j. Further, since r has replaced e in A_i, r is no longer a non-tree edge in any A_i. Thus Update establishes (iv) for the edges in D.

To see that Update does not destroy (iv) for any edge outside D, let f be a non-tree edge satisfying (iv) before the update and let i be the A_i in which f appears as a non-tree edge. If $j < i$, by (iv), f does not appear in any A_h, $h \leq j$, and hence f is not affected by the update. If, on the other hand, $i \leq j$, then after the update, f is still not a non-tree edge in any A_h, $h > j \geq i$, and clearly f becomes a non-tree edge in A_j if f is a global non-tree edge. This completes step (b). $\Box$

We are going to represent the local tree edges implicitly, so for efficiency, our main concern is the number of local non-tree edge initializations in (1). These are amortized over global edge deletions.

Note for our analysis of efficiency that it is valid to assume that all edges end up being deleted; for we always start with an empty graph, so given any operation sequence S, if we continue it by deleting all edges we get a sequence S' which is at most twice as long. Hence, if we can show for S' that the amortized operation cost is T, it follows that the amortized operation cost for S was at most $2T$.

Lemma 11 *For each edge deletion, for each $i = 0, \ldots, L$, and for each $j = 0, \ldots, i$, we make at most 2 local non-tree edge initializations in A_j in (1).*

Proof: All initializations happen in connection with an update when we set $A_j := F \cup D \bigcup_{h \leq j}(A_h \setminus F_h)$. We are only going to count the initializations with non-tree edges which either come from D or for some A_h where $h < j$. By definition of j, $|(D \cup \bigcup_{h \leq j}(A_h \setminus F_h)) \setminus F| > 2^{j-1}$ while $|(D \cup \bigcup_{h<j}(A_h \setminus F_h)) \setminus F| \leq 2^j$, so we know we are counting at least half the local non-tree edge initializations.

Consider the life cycle of some edge e in $\mathcal{A}$. An incarnation of an edge e is live in A_j if it is non-tree in A_j, and dead otherwise. By (vi) each edge has only one live incarnation.

Suppose during an update of $\mathcal{A}$ with D that e is initialized as a local non-tree edge into A_j. If $e \in D$, this is the birth, or rebirth of e. Otherwise, e comes from some A_h, $h \leq j$. If $h = j$, we do not count the initialization. Otherwise, we claim that there is a live incarnation of e among the A_h, $h < j$, and it is this live incarnation that we view as being moved up from A_h to A_j. Now e can only be initialized as local non-tree edge in A_j if it is also a global non-tree edge, but then by (vi) the first instance of e in some A_h is live, so indeed the update is moving a live instance of e up from some A_h to A_j where $h < j$.

Our final step is to note that the only way e can die from some A_i is if either e is deleted globally, in which case e escapes the cycle of rebirth, or if it becomes a local tree edge in A_i. The latter requires that some other edge is deleted from A_i and e comes in as a replacement edge. Then e will be moved to R from which it will be reborn in the subsequent update. We attribute the latest rebirth of e and all the progressive moves of e from A_h to A_j, $h < j$ to the death of e. This is at most one initialization in each A_j for $j = 0, ..., i$.

Since edges only die in connection with global deletions, and since each global deletion kills at most one edge from each A_i, the result follows. □

At present the number of initializations of local tree edges is not efficiently bounded. As mentioned previously, to resolve this, we will only maintain F implicitly. Instead of adding F to A_j in an update, we will add a forest F' of *super edges* e_P, each representing a path P in F. The super edges represent the minimal set of non-overlapping paths connecting the end points of the non-tree edges in A_j. Thus F' is an unrooted tree where degree 1 or 2 vertices are end points of non-tree edges in A_i. It follows that there are less than twice as many edges in F' as there are end-points, and for each non-tree edge, we have two end-points. Thus,

Lemma 12 *For each non-tree edge initialized in A_j, there are at most most 4 super edges initialized in A_j.*

A super edge e_P representing the path P is assigned the maximum weight in P, and e_P is deleted if any edge from P is deleted. Since deleting edges from A_i cannot turn tree edges into non-tree edges, our replacement of A_j with A'_j in the deletions only structures is valid. From Lemma 11 and 12, we conclude

Lemma 13 *The total cost of the decremental MSF structures is $O(\sum_{i=1}^{\log_2 m} \sum_{j=1}^{i} t(\min\{n, 2^j\}, 2^j)$.*

5.2 Implementation

For our implementation of the above reduction, we shall use the top tree data structure from Subsection 2.2. Our main challenges are in connection with super edges. First, to implement Update, we need to identify the super edges for A'_j. Also, when an edge e is to be deleted from A_i, we need to check whether it is part of a path P representing a super edge e_P, which is to be deleted from A'_i.

For each A_i we maintain a copy F^i of F from when A_i was last initiated. Let S denote the set of end points of non-tree edges in A_i from the initiation. To identify the super edges, we take the vertices from S one at a time, incrementally adding *super vertices* to the tree of super-edges. The paths between the super vertices in F^i are then *super paths* to be contracted to super edges.

More precisely, every time we add a vertex $v \in S$, we do as follows. First v is marked as a super vertex. If v is not connected in G to any other super vertex, we are done. Otherwise, we find the nearest vertex x on some super path. If x is a super vertex, we simply add the new super path $v \cdots x$. Otherwise, since the super paths only intersect in the super vertices, x must be internal to a unique current super path $a \cdots b$. We now mark x as a new super vertex, delete the super path $a \cdots b$ and add the three super paths $v \cdots x$, $a \cdots x$, and $b \cdots x$. To facilitate an efficient implementation, note

Lemma 14 *Suppose r is a super vertex, and let v be any vertex. Then the vertex x on a super path which is nearest to v is on the path $v \cdots r$. Further, if x is internal to a super path P, then P contains the edge leaving x on the path directed from v to r.*

Proof: First, suppose for a contradiction that x is not on $v \cdots r$. Since x is on a super path and r is a super vertex, all vertices on $x \cdots r$ must be on super paths. In particular this means that the first intersection x' between $v \cdots r$ and $x \cdots r$ is on a super path, but x' is closer to v than x, contradicting the choice of x. □

As mentioned, our implementation is based on the top trees from Subsection 2.2. When we start the process of finding the super paths, we assume that we already have a top tree over F^i. This is achieved by updating F^i to F every time we start initializing A_i. Hence, between initializations of A_i, we have to record the changes to F, that is, we maintain the difference between F^i and F. When A_i is initialized, we first delete all edges deleted from F since last initialization, and second we insert all the edges inserted in F. Since F has remained a tree during the updates, so does F^i. As a consequence, each change of F cause up to L top tree updates, leading to a cost per change of F of $O((\log n)^2)$.

With each tree T, we associate a variable super-root$_T$ which is **nil** if T does not contain a super vertex, and otherwise contains an arbitrary super vertex from T. Thus v is connected to a super vertex if and only if super-root$_{\mathrm{Find(v)}} \neq$ **nil**.

For each cluster C and boundary vertex a of C, we will maintain the variables nearest-super-path$_{C,a}$ and nearest-super-path-vertex$_{C,a}$ defined as follows. If there is no super path containing an edge from $\pi(C)$, nearest-super-path$_{C,a}$ = nearest-super-path-vertex$_{C,a}$ = **nil**. Otherwise, let P be the super path containing an edge of $\pi(C)$ nearest to a, and let x be the vertex of P on $\pi(C)$ closest to a. Then nearest-super-path$_{C,a} = P$ and nearest-super-path-vertex$_{C,a} = x$.

Suppose the above variables are properly maintained for the root clusters. Assuming this, we can implement the routine for adding a new point $v \in S$ as follows. First, we mark v as a super vertex. Next, we set $r :=$ super-root$_{\mathrm{Find(v)}}$. If $r =$ **nil**, v is not connected to any other current super vertices, so we complete by setting super-root$_{\mathrm{Find(v)}} := v$. Otherwise, we want to find the super path vertex x closest to v. This is done by setting $C :=$ Expose(v, r) and $P =$ nearest-super-path$_{C,v}$. If $P =$ **nil**, $x = r$ by Lemma 14. Otherwise, we set $x =$ nearest-super-path-vertex$_{C,v}$. Having identified x, we mark x as a new super vertex.

Assume we are in the complicated case where x was not already a super vertex. We still need to replace P with the three super paths $v \cdots x$, $a \cdots x$, and $b \cdots x$ where a and b are the end points. Here we represent P as identifier with which we have associated the pair of points (a, b). Further, with P, we associate a list of pointers to all references to P in the top tree, so that we can easily erase all information about P. Since the erasing the information is as quick as inserting it, we can ignore it in the asymptotic analysis of our algorithm. Having erased the information about P, we free the identifier for later reuse.

We now allocate three new super path identifiers $P_{(v,x)}$, $P_{(a,x)}$, and $P_{(b,x)}$. Inserting the information about these is done using a variable lazy-super-path$_C$ that for a cluster C is either **nil** or contains an identifier of a super path containing $\pi(C)$. Now, for any cluster D, either D has a path ancestor C with lazy-super-path$_C \neq$ **nil**, and then $\pi(D) \subseteq \pi(C) \subseteq$ lazy-super-path$_C$, or nearest-super-path$_{D,a}$ and nearest-super-path-vertex$_{D,a}$ have the correct information for each boundary vertex $a \in \partial D$. Since a root cluster has no ancestors, it always has the correct information.

To insert the three new super paths, for $y = v, a, b$, we insert $P_{(y,x)}$ as follows. First we set $C :=$ Expose(y, x) and then we call Add-path$(C, P_{(y,x)})$ defined below.

Add-path(C, P) Set lazy-super-path$_C := P$, and for each $a \in \partial C$, set nearest-super-path$_{C,a} := P$ and nearest-super-path-vertex$_{C,a} := a$.

To complete the description of the procedure for finding the super edges, we need to tell how to update information in connection with Merge and Split.

$C :=$ **Merge**(A, B) If C is not a path cluster (Figure 1(3-4)), we just set all variables of C to **nil** and return.

Otherwise, first we set we set lazy-super-path$_C :=$ **nil**.

If C has exactly one path child A (Figure 1(2)), let $\partial C = \{a, b\} = \partial A$. We then copy the information from A directly to C. That is, for $c = a, b$, we set nearest-super-path$_{C,c}$:= nearest-super-path$_{A,c}$ and nearest-super-path-vertex$_{C,c}$:= nearest-super-path-vertex$_{A,c}$.

If C has two path children (Figure 1(1)), let $\partial C = \{a, b\}$, $\partial A = \{a, c\}$, and $\partial B = \{c, b\}$. If nearest-super-path$_{A,a} \neq$ **nil**, set nearest-super-path$_{C,a}$:= nearest-super-path$_{A,a}$ and nearest-super-path-vertex$_{C,a}$:= nearest-super-path-vertex$_{A,a}$. Otherwise, if nearest-super-path$_{A,a} =$ **nil**, set nearest-super-path$_{C,a}$:= nearest-super-path$_{B,c}$ and nearest-super-path-vertex$_{C,a}$:= nearest-super-path-vertex$_{B,c}$. The values of nearest-super-path(C, b) and nearest-super-path-vertex(C, b) are found symmetrically with b and B replacing a and A.

Split(C) We only have to update information if lazy-super-path$_C \neq$ **nil**. In this case, for each path child $\mathcal{A}$ of $\mathcal{C}$, call Add-path$(A$, lazy-super-path$_C)$.

We have now shown the set of super edges can be constructed. When an edge (v, w) is globally deleted, for each i, we want to check if (v, w) is on a super path corresponding to a super edge in A'_i. This can only be the case if (v, w) is an edge in F^i. If so, we take the top tree over F^i, set $C :=$ Expose(v, w) and $P :=$ nearest-super-path(C, v). If $P =$ **nil**, (v, w) is not in a super path. Otherwise, P is the identifier for the super path containing (v, w), and with P we can easily store information about whether the corresponding super edge (a, b) has been deleted from A'_i. If not, we delete (a, b) from A'_i.

Lemma 15 *For a set S of end-points of non-tree edges, the corresponding set of super edges is found in time $O(|S| \log n)$ time. Further, identifying a potential super edge from A'_i covering an edge e to be deleted takes $O(\log n)$ time.*

Proof: Each Merge and Split takes constant time, so by Theorem 1 each top tree operation takes $O(\log n)$ time. □

Proof of Theorem 7 From Lemma 13, we have already accounted for the cost of the decremental MSF structures. It remains to show that the remaining cost of the other operations is $O((\log n)^3)$.

First, consider Insert and Delete without the call to Update. In connection with Insert(e), we need to check if there is a path in F between the end points of e, and if so, what is the heaviest edge on such a path. As described in Corollary 2, this can be done in time $O(\log n)$.

In connection with Delete(e), first, for each i we need to check if e is in a super edge e' of A'_i. By Lemma 15 this takes $O(\log n)$. Second, when e or e' has been deleted from A'_i, a replacement edge f may be returned, and then we have to check if the end points of f are connected in F. This takes $O(\log n)$ time using the Find operation of the top trees. Hence the cost of Delete ignoring Update is $O((\log n)^2)$.

Finally, for Update, our problem is to find the super edges. By Lemma 11 and 15, this has a cost of $O(\sum_{i=1}^{\log_2 m} \sum_{j=1}^{i} \log n) = O((\log n)^3)$ per delete. This completes the proof of Theorem 7. □

The total number of edges in the decremental MSF structures is $O(m)$, so their total space is $O(m \log m)$. To store the $O(\log n)$ trees F^i, including their top trees and their difference from F, we need $O(n \log n)$ space, so the total space for our fully dynamic MSF algorithm is $O(m \log n)$.

6 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.

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 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 [8].

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 containing an edge covering 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.

6.1 High level description

The algorithm associates with each non-tree edge e a level $\ell(e) \leq \ell_{\max} = \lceil \log_2 n \rceil$. 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, $G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_{\ell_{\max}} \supseteq F$. The following invariant is maintained:

(i') The maximal number of vertices in a 2-edge connected component of G_i is $\lceil n/2^i \rceil$. Thus, the maximal relevant level is $\ell_{\max}$.

It should be noted that we here round up instead of down for the component sizes. This will be significant for biconnectivity, where we will use that $\lceil n/2^i \rceil \leq 2\lceil n/2^{i+1} \rceil$

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 $\lceil n/2^j \rceil$ 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. Hence $c(e)$ is also the maximal level for which e is in a 2-edge connected component. 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) Decide if v and w are c-2-edge connected on level 0.

For basic updates of c-values, we have

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

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

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 v and w are 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 will first swap it with a non-tree edge as described in Swap below.

Swap(v, w) where (v, w) is a tree edge which is not a bridge, that is, $c(v, w) \geq 0$. Set $\alpha = c(v, w)$, and let (x, y) be a non-tree edge covering (v, w) with $\ell(x, y) = \alpha$. Set $\ell(v, w) := \alpha$. Replace (v, w) by (x, y) in F. Call InitTreeEdge(x, y) and Cover(v, w, α).

To see that Swap does not violate invariant (i'), we argue much stronger that Swap does not change the 2-edge connected components of any G_i. If $i \leq \alpha$, then both (v, w) and (x, y) are in G_i both before and after the call, so G_i is unchanged. If $i > \alpha$, (v, w) was a bridge in G_i. Then (x, y) must be a new bridge in G_i, for otherwise there should be a level i non-tree edge (q, r) covering (x, y) after the swap, but this edge would have covered (v, w) before the swap. However, replacing one bridge with another, does not affect any 2-edge components of G_i. Thus no 2-edge connected component is affected in any G_i.

Next we need to argue that the cover information is correctly updated. All edges covered by (v, w) after the swap, except for (x, y), were covered by (x, y) before the swap, and hence their c-values where at least $\ell(x, y) = \alpha$. Thus, when we call Cover(v, w, α) we do not affect any of these c-values. This lack of change is correct since the 2-edge connected components are not changed in any G_i. Concerning (x, y), Cover(v, w, α) sets $c(x, y) := \alpha$, which is correct since (x, y) is a bridge in any G_i with $i > \alpha$.

We are now ready to describe how to delete an edge.

Delete(v, w) If (v, w) is a bridge, we simply delete it and return.

Assuming that (v, w) is not a bridge, if (v, w) is a tree edge, we first call Swap(v, w), turning (v, w) into a non-tree edge. We then call Uncover$(v, w, \ell(v, w))$ as defined below, delete the edge (v, w), and finally, for $i := \ell(v, w), \ldots, 0$, we call Recover$(v, w, i)$ as defined below.

The point in Uncover is to remove all cover information potentially stemming from (v, w). This is done as follows.

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

Our problem now is that the c-values on $v \cdots w$ may have become too low. Formally, 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. After the call Uncover$(v, w, \ell(v, w))$ in Delete, $v \cdots w$ is fine on level $\ell(v, w) + 1$.

The procedure Recover(v, w, i) defined below assumes that $v \cdots w$ is fine on level $i + 1$, and then it makes $v \cdots w$ fine on level i. Thus, given a correct implementation of Recover, all cover information will be correct after the final call Recover$(v, w, 0)$ in Delete.

A correct implementation of Recover(v, w, i) would be to take all level i edges (q, r) covering edges in $v \cdots w$, and call Cover(v, w, i). As in basic connectivity, we would like to amortize the calls by first increasing the level of (q, r) to $i + 1$ and then call Cover$(q, r, i + 1)$. This time we need to be a lot more careful, however. The basic problem is that we are not just trying to recover a single component with a single replacement edge, but rather we have to recover a hole chain of 2-edge connected components along $v \cdots w$.

Before presenting our implementation of Recover, consider a level i edge (q, r) covering some edge in $v \cdots w$. By definition, $q \cdots r \cap v \cdots w \neq \emptyset$. Since cover information is correct outside $v \cdots w$, it follows that $c(\mathit{meet}(q, v, w) \cdots q) \geq i$. This condition is, in fact, equivalent to (q, r) being in the same 2-edge connected component as (v, w) in G_i before the deletion of (v, w).

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 level i non-tree edges (q, r) with $\mathit{meet}(q, v, w) = u$ and $c(u \cdots q) \geq i$. If legal, increase the level of (q, r) to $i + 1$ and call Cover$(q, r, i + 1)$. Otherwise, we call Cover(q, r, i) and stop the phase.

If the first phase was stopped, we have a second symmetric phase, starting with $u = w$, and stepping u through the vertices in $w \cdots v$ towards v.

Our final step is to establish the correctness of Recover.

Lemma 16 *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: For brevity, we say that a level i non-tree edge is *relevant* if $c(\mathit{meet}(q, v, w) \cdots q) \geq i$.

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

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 relevant edges (q, r), all edges e in $q \cdots r \cap v \cdots w$ have $c(e) \geq i$. This is trivially the case if phase 1 runs through without being stopped, for then there are no remaining relevant edges.

Now, suppose phase 1 is stopped. Let u_1 be the last value of u considered, and (q_1, r_1) be the last edge considered. Then increasing the level of (q_1, r_1) is illegal. Hence phase 2 will also stop, for otherwise, it would end up illegally increasing the level of (q_1, r_1). Let u_2 be the last value of u considered, and let (q_2, r_2) be the last edge considered in phase 2.

Since the phases were not interrupted for non-tree edges (q, r) covering edges before u_1 or after u_2, we know that if (q, r) remains on level i, it is because $q \cdots r \cap v \cdots w \subseteq u_1 \cdots u_2$. 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 (q_k, r_k), it follows that the 2-edge connected component C_k of q_k in $G_{i+1} \cup \{(q_k, r_k)\}$ has $> \lceil n/2^{i+1} \rceil$ vertices. However, we know that before the deletion of (v, w), C_1 and C_2 where both part of the 2-edge connected component C of G_i containing (v, w), and this component had at most $\lceil n/2^i \rceil$ vertices. Hence $C_1 \cap C_2 \neq \emptyset$. Thus, C_1 and C_2 are contained in the same 2-edge connected component D of $G_{i+1} \cup \{(q_1, r_1), (q_2, r_2)\}$. Since covering is done for all level $i + 1$ edges, it follows that our calls Cover(q_1, r_1, i) and Cover(q_2, r_2, i) imply that all tree edges in D get 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.

6.2 Implementation

The algorithm maintains the spanning forest in the top tree data structure from Subsection 2.2. For each cluster C we maintain $\text{cover}_C = c(\pi(C))$. Thus, 2-edge connectivity queries are implemented by:

2-edge-connected(v, w) Set $C :=$ Expose(v, w). Return $(\text{cover}_C \geq 0)$.

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 cover-edge$_C$ covering an edge on $\pi(C)$ with $\ell(\text{cover-edge}_C) = \text{cover}_C$. Then the desired edge e is found by setting $C :=$ Expose(v, w) and returning cover-edge$_C$. Calls to cover and uncover also reduce to operations on clusters:

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

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

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

- cover_C^+, cover_C^- and cover-edge$_C^+$, where $\text{cover}_C^+ \leq \text{cover}_C^-$. This represents that for each path descendant D of C, if $\text{cover}_D \leq \text{cover}_C^-$, we should set $\text{cover}_D := \text{cover}_C^+$ and cover-edge$_D :=$ cover-edge$_C^+$.

The lazy information has no effect if $\text{cover}_C^+ = \text{cover}_C^- = -1$. 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 Cover, Uncover, Split, and Merge.

- For any vertex v and any level i, let $\text{incident}_{v,i}$ be the number of level $\geq i$ non-tree edges with an endpoint in v.

- Let i and j be levels, and let v be a boundary vertex of a path cluster C. Let $I_{C,v,i,j}$ be the set of internal vertices of the cluster C that are reachable from v by a path P in F where $c(P \cap \pi(C)) \geq i$ and $c(P \setminus \pi(C)) \geq j$. Then $\text{size}_{C,v,i,j} = |I_{C,v,i,j}|$ and $\text{incident}_{C,v,i,j} = (\sum_{w \in I_{C,v,i,j}} \text{incident}_{w,j})$ is the number of (directed) level j non-tree edges (q,r) with $q \in I_{C,v,i,j}$. By directed we mean that (q,r) is counted twice if r is also in $I_{C,v,i,j}$.

- Similarly for any level i and any non-path cluster C with $\partial C = \{v\}$ let $I_{C,v,i}$ be the set of internal vertices q from C such that $c(v \cdots q) \geq i$. Then $\text{size}_{C,v,i} = |I_{C,v,i}|$ and $\text{incident}_{C,v,i} = (\sum_{w \in I_{C,v,i}} \text{incident}_{w,i})$ is the number of (directed) level i non-tree edges (q,r) with $q \in I_{C,v,i}$.

For an edge (v,w), we maintain $\text{cover}_{(v,w)}$ no matter whether (v,w) is a path cluster or not. If (v,w) is a path cluster, it has no internal vertices, so all of the above size and incidence counters are zero. However, if v is the only boundary vertex of (v,w), $\text{size}_{(v,w),v,i} = 1$ if $i \leq \text{cover}_{(v,w)}$; 0 otherwise. Similarly, $\text{incident}_{(v,w),v,i} = \text{incident}_{v,i}$ if $i \leq \text{cover}_{(v,w)}$; 0 otherwise.

When a non-tree edge gets inserted or deleted, or its level changes, we always expose its end-points so that they are not internal to any clusters. This has the convinient effect that we do not affect any of the incident-counters at the clusters until we start covering or uncovering the path between the end-points.

We are now ready to implement all the different procedures: For any vertex v and any level i, let $\text{size}_{v,i} := 1$ and

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

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

Clean(C) For each path child A of C, call $\text{Uncover}(A, \text{cover}_C^-)$ and $\text{Cover}(A, \text{cover}_C^+, \text{cover-edge}_C^+)$. Set $\text{cover}_C^+ := -1$ and $\text{cover}_C^- := -1$ and $\text{cover-edge}_C^+ :=$ **nil**.

Split(C) Call $\text{Clean}(C)$. Delete C.

C:=Merge(A, B) Suppose $\partial C = \{a\}$ and $a \in \partial A$ (Figure 1(3-4)). For $X := \text{size,incident}$ and $j := 0, \ldots, \ell_{\max}$ we compute $X_{C,a,j}$ as follows. If A is a non-path cluster, so is B (Figure 1(4)), and then $\partial A = \partial B = \{a\}$. In this case, we set $X_{C,a,j} := X_{A,a,j} + X_{B,a,j}$. Otherwise (Figure 1(3)), $\partial A = \{a,b\}$ and $\partial B = \{b\}$, in which case we set $X_{C,a,j} := X_{A,a,j,j}(+X_{b,j} + X_{B,b,j}$ if $\text{cover}_A \geq j)$.

Suppose $\partial C = \{a,b\}$, $a \in \partial A$, and $b \in \partial B$ (Figure 1(1-2)). Let D be the path child of C minimizing cover_D. Then set $\text{cover}_C := \text{cover}_D$ and $\text{cover-edge}_C := \text{cover-edge}_D$. Set $\text{cover}_C^+ := -1$ and $\text{cover}_C^- := -1$ and $\text{cover-edge}_C^+ :=$ **nil**. For $X := \text{size,incident}$ and $i, j := -1, \ldots, \ell_{\max}$, we compute $X_{C,a,i,j}$ as follows ($X_{C,b,i,j}$ is symmetric). If A is a non-path cluster (Figure 1(2)), $\partial A = \{a\}$ and $\partial B = \{a,b\}$. In this case, we set $X_{C,a,i,j} := X_{A,a,j} + X_{B,a,i,j}$. Otherwise, if B is a non-path cluster, $\partial A = \{a,b\}$ and $\partial B = \{b\}$, and we set $X_{C,a,i,j} := X_{A,a,i,j}(+X_{B,c,j}$ if $\text{cover}_A \geq i)$. Finally if both A and B are path clusters (Figure 1(1)), $\partial A = \{a,c\}$ and $\partial B = \{c,b\}$, and we set $X_{C,a,i,j} := X_{A,a,i,j}(+X_{c,j} + X_{B,c,i,j}$ if $\text{cover}_A \geq i)$.

Recover(v, w, i)

- Repeat once with $u = v$ and once with $u = w$,
  - Set $C :=$ Expose(v, w).
  - While incident$_{C,u,-1,i}$+incident$_{u,i} > 0$ and not stopped,
    * Set $(q, r) :=$ Find(u, C, i).
    * $D :=$ Expose(q, r).
    * If size$_{D,q,-1,i+1} + 2 > n/2^{i+1}$, '+2' adds the two external boundary vertices.
      - 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 vertex nearest to a in B, return Find(b, B, i).

Theorem 17 *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) also takes $O(\log^2 n)$ time we have by theorem 1 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 $O(\log^3 n)$ time plus $O(\log^3 n)$ time per 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. □

The space usage of our fully dynamic 2-edge connectivity algorithm is $O(m + n \log^2 n)$ due to the $O(\log^2 n)$ counters $X_{C,v,i,j}$ stored with each path cluster. It is possible to reduce the space to $O(m + n \log n)$: using a more complicated merge, it suffices that we only store the $O(\log n)$ counters $X_{C,v,0,j}$ for the path clusters, that is, we ignore the covering of the cluster path. The main complication is then the merge in Figure 1(3) of a path cluster A and a non-path cluster B into a non-path cluster C where we now need to determine how much of the cluster path of A that is covered on different levels. The time bounds are not affected by this change.

The query time for 2-edge connectivity above is $O(\log^3 n)$, but it can be reduced to $O(\log n)$. The basic idea is to leave the top tree unchanged. The point is that with the general Expose, we perform $O(\log n)$ merges and splits, each at cost $O(\log^2 n)$. However, for our query, we only need to check if some covering is positive, and then we really only need to spend constant time per cluster considered.

Thorup [30] has recently observed that the time bound for the updates can be improved by a factor $O(\log n/ \log \log n)$. The essential point is that it suffices to maintain our size- and incident-counters approximately, using $O(\log \log n)$ bits, and then we can operate on $O(\log n/ \log \log n)$ of them in constant time. This only works if we store all of the $O(\log^2 n)$ counters $X_{C,v,i,j}$ for the path clusters, that is, it does not work tother with the above mentioned space improvement. Summing up, Thorup gets an amortized operation time of $O(\log^3 n \log\log n)$ using $O(m + n\log n\log\log n)$ space.

6.3 Bridges

We note that the data already stored in the top trees make it easy to search for bridges. First we show how to augment our 2-edge connectivity query to provide a bridge between v and w if they are connected but not 2-edge connected. That is, we have just set $C := \text{Expose}(v, w)$, but found $\text{cover}_C = -1$. We then run

- While C is not an edge,
  - Clean(C).
  - Let A be a path child of C with $\text{cover}_A = -1$.
  - Set $C := A$.
- Return the edge C.

Another scenario considered by Gabow et al. [11] is that we are given a vertex v, and want to determine if it is connected to a bridge. In order to facilitate a recursive search, we consider the existential problem of checking if a cluster A has a bridge, i.e. if it is not 2-edge connected. If A is a non-path cluster with $\partial A = \{a\}$, A has a bridge if and only if $\text{size}_{A,a,0} < \text{size}_{A,a,-1}$. If instead A is a path cluster with $\partial A = \{a, b\}$, A has a bridge if and only if $\text{cover}_A = -1$ or $\text{size}_{A,a,0,0} < \text{size}_{A,a,-1,-1}$.

Find-bridge(v) Finds bridge connected to v, if any.

- Set $C := \text{Expose}(v, v)$.
- If C has no bridge, return "The component of v is 2-edge connected"
- While C is not an edge,
  - Clean(C).
  - Let A be a child of C containing a bridge.
  - Set $C := A$.
- Return the edge C.

Both of the above bridge finding procedures take $O(\log^3 n)$ time. As for the 2-edge connectivity query, this can be reduced to $O(\log n)$ time with a more complicated algorithm that does not change the top tree.

7 Biconnectivity

In this section we present an $O(\log^5 n)$ 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 [5, 18]. Furthermore, the generalization in [14] of the $O(\log^5 n)$ randomized 2-edge connectivity algorithm from [17] has an expected bound of $O(\Delta \log^4 n)$, where

Δ is the maximal degree [Henzinger, pc 1997]. Our main new idea for getting a $O(\log^5 n)$ bound for biconnectivity is an efficient recycling of the information as described in Lemma 19 below.

One of the obstacles for biconnectivity is that it is not a transitive relation over vertices. However, it is a transitive relation over edges in the sense that for edges e, f, g, if e and f are in a biconnected component and f and g are in a biconnected component, then all of e, f, and g are in the same biconnected component. Our algorithm makes use of the transitivity over edges.

More particularly, in 2-edge connectivity we used that when an edge (v, w) was deleted, the 2-edge connected components to be recovered were linearly ordered along the path $v \cdots w$. Our amortization worked for all but one large middle component, and hence when we had reached it from both v and w, we knew we had visited all other components. In biconnectivity, we can have different biconnected components meeting in each vertex $u \in v \cdots w$, and our problem is that we cannot define a corresponding order for these. We circumvent this problem by recycling some information, allowing us to skip some components.

A *triple* is a length two path xyz in the graph G, and a *tree triple* xyz is a triple in F. Let (v, w) be a non-tree edge. Then (v, w) *covers* all triples on the cycle induced by (v, w) in F, that is, (v, w) covers all triples $xyz \subseteq v \cdots w$ plus the triples $wvs^w(v)$ and $vws^v(w)$. Recall here that $s^w(v)$ is the vertex succeeding v in $v \cdots w$. The covering of triples is symmetric, so when covering xyz, it is understood that we also cover zyx.

We now define *transitively covered triples* as follows. All covered triples are transitively covered. Further, if xyz and zyx' are transitively covered, then xyx' is transitively covered.

Lemma 18

(a) Biconnectivity is a transitive relation over the neighbors of a vertex u, and if two neighbors of u are biconnected, u is in the biconnected component containing them.

(b) A triple xyz is transitively covered if and only if x and z are biconnected.

(c) A vertex y is an articulation point if and only if there is a tree triple xyz which is not transitively covered.

(d) Two vertices v and w are biconnected if and only if for all $xyz \subseteq v \cdots w$, xyz is transitively covered.

Proof: (a) Let v and w be biconnected neighbors of u. By definition, either (v, w) is an edge, or we have two internally vertex disjoint paths from v to w. In either case, we find a path P from v to w not containing u, and then Pu is a simple cycle showing that all of u, v, and w are in the same biconnected component. Moreover, the cycle shows that the edges (u, v) and (u, w) are biconnected. Since biconnectivity is a transitive relation over edges, it follows that biconnectivity is a transitive relation over the neighbors of u.

(b) First we show that we can restrict our attention to the case were xyz is a tree triple. If x is not a tree neighbor of y, let x' be the tree neighbor $s^x(y)$ of y. Then (x, y) covers xyx', so xyz is transitively covered if and only if $x'yz$ is transitively covered. The cycle induced by (x, y) contains x', so x and x' are biconnected. By (a), biconnectivity over neighbors of y is transitive, so y is biconnected to x if and only if it is biconnected to x'. It follows that we can replace x by the x' in (b). Similarly, if z is not a tree neighbor of y, it can be replaced by the tree neighbor $s^z(y)$. Thus we may assume that xyz is a tree triple when proving (b).

Now, assume the two tree neighbors x and z of y are biconnected. As in (a), we can find a path P from x to z not containing y. Let $T_1, ..., T_k$ be the subtrees of $F \setminus \{y\}$ that P passes on the way from x to z. Further let x_i be the vertex of T_i that is a tree neighbor of y in F. Then $x_1 = x$ and $x_k = z$. Now, for $i = 1, ..., k-1$, P contains an edge between T_i and T_{i+1} which covers $x_i y x_{i+1}$. It follows that xyz is transitively covered.

For the other direction, assume that the tree triple xyz is transitively covered. We then have a sequence $x_1, ..., x_k$ of neighbors to y such that $x_1 = x$, $x_k = z$, and $x_i y x_{i+1}$ is covered by an edge e_i. Let T_i be the subtree of $F \setminus \{y\}$ containing x_i. Let P_1 be the path in T_1 from x_1 to e_1. For $i = 2, ..., k-1$, let P_i be path in T_i connecting e_{i-1} to e_i, and let P_k be the path in T_k connecting e_{k-1} to x_k. Then $P_1 \cdots P_k$ is a path from x to z disjoint from xyz.

(c) If y is not an articulation point then any two of its neighbors are biconnected. In particular, for any tree triple xyz, x and z are biconnected, so by (b), xyz is transitively covered. Conversely, if all tree triples are covered and x and z are arbitrary neighbors of y, then $s^x(y)ys^z(y)$ is transitively covered, and hence xyz is transitively covered. Thus x and z are biconnected by (b).

(d) Suppose that all triples $xyz \subseteq v \cdots w$ are transitively covered. By (b), each edge pair (x, y) and (y, z) is biconnected, so by transitivity of biconnectivity on edges, the first and the last edge of $v \cdots w$ are in the same biconnected component. Hence v and w are in this component.

Conversely, suppose v and w are biconnected. By definition there are two internally vertex disjoint paths P_1 and P_2 between v and w. For each $xyz \subseteq v \cdots w$, there is a P_i not containing y. Now, $P_i \cup v \cdots w$ must contain a cycle containing xyz, so x and z are biconnected, and hence xyz is transitively covered by (b). □

7.1 High-level

As with 2-edge connectivity, with each non-tree edge e, we associate a level $\ell(e) \in \{0, \ldots, \ell_{\max}\}$, $\ell_{\max} = \lceil \log_2 n \rceil$, 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 $G = G_0 \supseteq G_1 \supseteq \ldots \supseteq G_{\ell_{\max}} \supseteq F$. Here, for biconnectivity, we will maintain the invariant:

(i") The maximal number of vertices in a biconnected component of G_i is $\lceil n/2^i \rceil$.

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 $G_j \cup \{e\}$ has at most $\lceil n/2^j \rceil$ vertices.

For each vertex v and each level i, we will maintain a partitioning of all neighbors of v in G such that u and w are in the same set if and only if uvw is a transitively covered triple in G_i. The set in the partitioning containing u is denoted $c^*_{v,i}(u)$. By Lemma 18(b), u and w belong in the same set if and only if they are biconnected neighbors of v in G_i. Note that a neighbor in G of v which is not a neighbor in G_i of v is a singleton element in the partition. It might seem more natural to exclude such neighbors from the partitioning, but leaving them in saves on administration, and gives the advantage that the partitioning does not depend on the bridges of G_i.

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.

As for 2-edge connectivity, the c^*-values may temporarily be too low. 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) Decide if v and w are c^*-biconnected on level 0.

In connection with deletions, we are going to reuse the swap procedure from 2-edge connectivity. Recall from our analysis of Swap that it only affects the G_i by swapping bridges. Hence Swap does not affect the c^*-values. While Swap does not affect the transitive covering of triples, it may strongly affect which triples are covered, and this is one of the reasons why we maintain transitive covering, not worrying about which covered triples are currently generating the transitive covering.

For Swap we need the c-values from 2-edge connectivity. That is, for each tree edge e, $c(e)$ should be the minimum level a non-tree edge covering e. We will maintain the c-values via the procedures from 2-edge connectivity, prefixing the procedure names by '2e-'.

For basic manipulation of c- and c^*-values, we have

InitEdge(v, w) For $i := 0, \cdots, \ell_{\max}$, set $c^*_{v,i}(w) := \{w\}$ and $c^*_{w,i}(v) := \{v\}$. Moreover, if (v, w) is a tree edge, call 2e-InitTreeEdge(v, w).

Cover(xyz, i) Union $c^*_{y,j}(x)$ and $c^*_{y,j}(z)$ for $j := 0, \ldots, i$.

Cover(v, w, i) Calls Cover$(vws^v(w), i)$, Cover$(wvs^w(v), i)$, and Cover(xyz, i) for all $xyz \subseteq v \cdots w$. Finally, we call 2e-Cover(v, w, i).

We can now compute all c- and c^*-values by first calling InitEdge(v, w) for all edges (v, w), and second calling Cover$(v, w, \ell(v, w))$ for all non-tree edges (v, w).

Inserting an edge is now straightforward.

Insert(v, w) If v and w are not connected in F, (v, w) is added to F and InitEdge(v, w) is called. Otherwise, call InitEdge(v, w), set $\ell(v, w) := 0$, and call Cover$(v, w, 0)$.

On the high level, Delete is almost identical to 2e-Delete.

Delete(v, w) If (v, w) is a bridge, we simply delete it, deleting w from $c^*_{v,\cdot}(\cdot)$ and v from $c^*_{w,\cdot}(\cdot)$, and then we return.

Assuming that (v, w) is not a bridge, if (v, w) is a tree edge, we first call Swap(v, w), turning (v, w) into a non-tree edge. We then call Uncover$(v, w, \ell(v, w))$ as defined below, delete the edge (v, w), and finally, for $i := \ell(v, w), \ldots, 0$, we call Recover$(v, w, i)$ as defined below.

As in 2-edge connectivity, the basic goal of Uncover is to remove the cover information stemming from the edge (v, w). However, since we are maintaining transitive covering, it is not quite so obvious what to do. We could of course, remove all transitive covering that (v, w) had been part of generating, but then we would not be able to recover the correct cover information efficiently. Our key insight is expressed in the lemma below.

Lemma 19 *Let (v, w) be a level i non-tree edge covering a tree triple $xyz \subseteq v \cdots w$. Consider G_j, $j \leq i$. Suppose s is a neighbor of y biconnected to x, hence to y and z. Then, if (v, w) is deleted, afterwards, s is biconnected to x or z, possibly to both.*

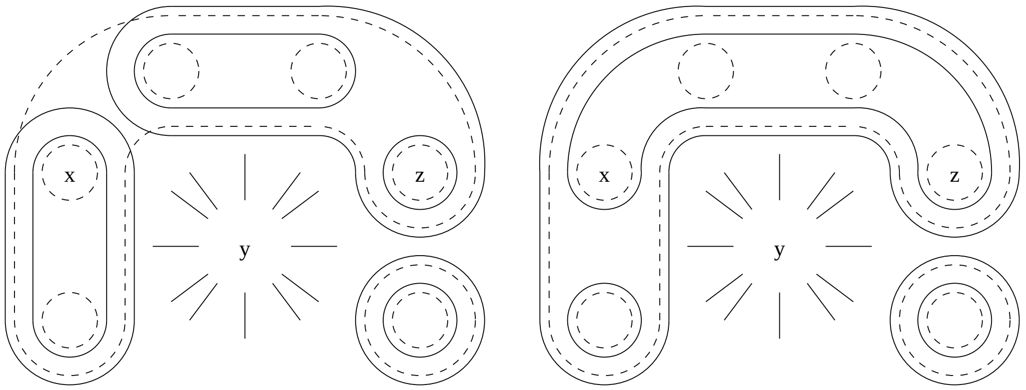

Figure 2: Application of Lemma 19. The dashed lines show the transitive covering before the edge deletion, which is the same on both figures. The solid lines show our recovery. The inner bubbles represent level $i+1$ while the outer bubbles represent level i.

Proof: From Lemma 18 (b), we know that biconnectivity of neighbors of y is the transitive closure over pairs (x', z') where $x'yz'$ is covered. Removing one pair (x, z) can either not change the transitive closure, or split it in two, with one part containing x and the other containing z. $\square$

The lemma suggests, that when (v, w) is deleted, we should store the neighbors s mentioned. From before the deletion, these neighbors form the set $c^*_{y,j}(x) = c^*_{y,j}(z)$. Generally, we will use the set $c^*_{y,j}(x|z)$ to store the set of neighbors to y that we know are biconnected to x or z on level j, but that are not yet c^*-biconnected to either. The application of the lemma is illustrated in Figure 2. We will be expanding $c^*_{y,j+1}(x)$ until either we get stuck with $c^*_{y,j+1}(x) = c^*_{y,j}(x)$ and $z \notin c^*_{y,j}(x)$, or we include z in $c^*_{y,j+1}(x)$. In the former case, we will set $c^*_{y,j}(z) := c^*_{y,j}(x|z) \cup c^*_{y,j}(z)$ and $c^*_{y,j}(x|z) := \emptyset$. In the latter case, we will set $c^*_{y,j}(x) := c^*_{y,j}(x|z) \cup c^*_{y,j}(x) \cup c^*_{y,j}(z)$ and $c^*_{y,j}(x|z) := \emptyset$. When (v, w) is deleted, it is only triples $xyz \in v \cdots w$ that are affected in the above way. In particular, for any y', there can only be one affected tree triple $x'y'z'$. If there is such a triple $x'y'z'$ we say that it is *fuzzy covered*. We can now describe the uncovering of a tree triple:

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 $c^*_{y,j}(x|z) := c^*_{y,j}(x) \backslash (c^*_{y,i+1}(x) \cup c^*_{y,i+1}(z))$, $c^*_{y,j}(x) := c^*_{y,i+1}(x)$, and $c^*_{y,j}(z) := c^*_{y,i+1}(z)$. Finally set $c^*_{y,j}(s) := c^*_{y,j+1}(s)$ for all $s \in c^*_{y,j}(x|z)$

For the sake of recovering, we need a matching redefinition of our covering procedure.

Cover(xyz, i) For $j = i, \ldots, 0$, we do as follows. If there is no fuzzy covered triple $x'yz'$ with $c^*_{y,j}(x'|z') \neq \emptyset$, we just union $c^*_{y,j}(x)$ and $c^*_{y,j}(z)$. If there is a fuzzy covered triple $x'yz'$ and $c^*_{y,j}(x'|z') \neq \emptyset$, we divide into cases. If neither of x' and z' are in $c^*_{y,j}(x)$ or $c^*_{y,j}(z)$, we again just union $c^*_{y,j}(x)$ and $c^*_{y,j}(z)$. Otherwise, by symmetry, we may assume that $c^*_{y,j}(x)$ contains x'. Since $c^*_{y,j}(x'|z') \neq \emptyset$, $c^*_{y,j}(x)$ cannot also contain z'. If $z' \in c^*_{y,j}(z)$, we union $c^*_{y,j}(x)$, $c^*_{y,j}(z)$, and $c^*_{y,j}(x'|z')$, and set $c^*_{y,j}(x|z) := \emptyset$. Otherwise, we union $c^*_{y,j}(x)$ and $c^*_{y,j}(z)$, and subtract them from $c^*_{y,j}(x'|z')$.

Before giving the final description of how to Uncover a non-tree edge (v, w), recall that (v, w) covers the triples $wvs^w(v)$ and $wvs^w(v)$ where $(v, s^w(v))$ and $(w, s^v(w))$ are tree edges. There are no other non-tree edges that can cover a triple containing the non-tree edge (v, w). Hence removing the covering of $wvs^w(v)$ corresponds to removing the transitive covering of all triples $wv\cdot$, thus singling w out from its set in the partitioning of the neighbors of v. Since the deletion of (v, w) also stops w from being a neighbor of v, we can simply remove w from $c^*_{v,\cdot}(\cdot)$. Similarly, v is removed from $c^*_{w,\cdot}(\cdot)$.

Uncover(v, w, i) Call Uncover(xyz, i) for all $xyz \subseteq v \cdots w$. Delete w from $c^*_{v,\cdot}(\cdot)$ and v from $c^*_{w,\cdot}(\cdot)$. Call 2e-Uncover(v, w, i).

To complete the description of Delete, we need to define Recover(v, w, i). Our general assumption is that all c^*-information is correct, except that we need to resolve the sets $c^*_{y,j}(x|z)$ with $xyz \subseteq v \cdots w$. When Recover(v, w, i) is called, we assume these sets are empty for $j > i$, and now we need to empty them for $j = i$. The definition is rather subtle and we defer the explanation to the subsequent proofs.

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

Let u step through the vertices of $v \cdots w$ towards w, starting with $u = v$.

1. if $u \neq v$, set $u' := s^v(u)$ and run
   (*) While there is a level i non-tree edge (q, r) such that $u = \mathit{meet}(q, v, w)$ 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.
2. if $u \notin \{v, w\}$ and $c^*_{u,i}(s^v(u)|s^w(u)) \neq \emptyset$,
   Union $c^*_{u,i}(s^w(u))$ and $c^*_{u,i}(s^v(u)|s^w(u))$, and set $c^*_{u,i}(s^v(u)|s^w(u)) := \emptyset$.
3. if $u \neq w$, set $u' := s^w(u)$ and run (*).

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

As a first step in proving correctness, we prove

Lemma 20 *For each value of u, after step 2, $c^*_{u,i}(s^v(u)|s^w(u))$ is correctly resolved.*

Proof: Since the two phases are symmetric, we may assume that we are in Phase 1.

For each value of u, we want to prove that $c^*_{u,i}(s^v(u)|s^w(u))$ gets correctly distributed between $c^*_{u,i}(s^v(u))$ and $c^*_{u,i}(s^w(u))$. Given that Cover is correct, the critical point is step 2 where we union $c^*_{u,i}(s^w(u))$ and $c^*_{u,i}(s^v(u)|s^w(u))$, setting $c^*_{u,i}(s^v(u)|s^w(u)) := \emptyset$. This requires that $c^*_{u,i}(s^v(u))$ is completed with no elements left in $c^*_{u,i}(s^v(u)|s^w(u))$. Inductively, we prove correctness assuming correctness for each previous value u^* of u.

If $c^*_{u,i}(s^v(u))$ is not completed, it is because there is a level i non-tree edge (q, r) covering a triple xuz with $x \in c^*_{u,i}(u')$ and $z \notin c^*_{u,i}(u')$. Here, by symmetry, we assume that x is nearest to q, i.e. that $s^q(u) = x$ and $s^r(u) = z$.

Suppose $\mathit{meet}(q, v, w) \neq u$, and set $u^* = \mathit{meet}(q, v, w)$. Then (q, r) covers $s^w(u^*)u^* \cdots qr$. By induction, $c^*_{u^*,i}(\cdot)$ was correct after step 2, and $u^* \cdots qr$ has been correctly covered all along as it does not contain a triple from $v \cdots w$. Thus $c^*(s^w(u^*)u^* \cdots qr) \geq \ell(q, r) = i$, and hence we should have found and incremented the level of (q, r) in step 3.

Suppose instead $\mathit{meet}(q, v, w) = u$. Since $s^q(u) \in c^*_{u,i}(s^v(u))$, $c^*(s^v(u)us^q(u)) \geq i$. Moreover, $u \cdots qr$ is covered by (q, r) and $u \cdots qr$ does not contain any triples from $v \cdots w$, so $c^*(u \cdots qr) \geq i$. Thus, $c^*(s^v(u)u \cdots qr) \geq i$, and hence (q, r) should have been found and had its level increased in step 1. □

The remaining proof of correctness of Recover is quite similar to that for 2-edge connectivity.

Lemma 21 *Recover(v, w, i) correctly resolves all sets $c^*_{u,i}(s^v(u)|s^w(u))$ for $u = s^w(v) \cdots s^v(w)$.*

Proof: By Lemma 20 we are done for all values of u for which we have passed step 2. In particular, we are done if the phases are not stopped.

Now, suppose phase 1 is stopped. Let u_1 be the last value of u considered, and (q_1, r_1) be the last edge considered. Then increasing the level of (q_1, r_1) is illegal. Hence phase 2 will also stop, for otherwise, it would end up illegally increasing the level of (q_1, r_1). Let u_2 be the last value of u considered, and let (q_2, r_2) be the last edge considered in phase 2.

For $k := 1, 2$, from the illegality of increasing the level of (q_k, r_k), it follows that the biconnected component C_k of q_k in $G_{i+1} \cup \{(q_k, r_k)\}$ has $\geq \lceil n/2^{i+1} \rceil + 1$ vertices. However, we know that before the deletion of (v, w), C_1 and C_2 where both part of a biconnected component C of G_i, and this component had at most $\lceil n/2^i \rceil$ vertices. Hence C_1 and C_2 overlap in at least two vertices, implying that they are contained in the same biconnected component D of $G_{i+1} \cup \{(q_1, r_1), (q_2, r_2)\}$. Since covering is assumed complete for level $i+1$ edges in D, it follows that after our calls Cover(q_1, r_1, i) and Cover(q_2, r_2, i), all covering of D is updated.

To complete the proof, we need to show for each $u \in u_1 \cdots u_2$, either (1) that $s^v(u)us^w(u) \subseteq D$ in which case correct covering is inherited from D, or (2) that we have passed step 2 for u in which case we apply Lemma 20.

First consider u between u_1 and u_2. Both u_1 and u_2 are in the biconnected component D. Hence, by Lemma 18 (d), the triple $s^v(u)us^w$ is in D, so (1) applies.

Next, consider u_1. If step 2 has been passed for u_1, (2) applies. Otherwise, we were stopped in step 1, so $s^q(u_1) \in c^*_{u_1,i}(s^v(u_1))$, but then, $s^v(u_1)$ is in the biconnected component D.

Now, if $u_1 \neq u_2$, $s^w(u_1) \in v \cdots w \subseteq D$, so $s^v(u_1)u_1s^w(u_1) \subseteq D$ and (1) applies. If $u_1 = u_2$ and Phase 2 was stopped after step 2, (2) applies to $u_1 = u_2$. Otherwise, as above we get that $s^w(u_2) \in D$, hence that $s^v(u_1)u_1s^w(u_1) \subseteq D$, so (1) applies. □

Finally, we need to argue that Recover also restores the correct covering of edges as in 2-edge connectivity.

Lemma 22 *Recover(v, w, i) correctly covers the edges on $v \cdots w$ as in 2-edge connectivity.*

Proof: Inductively, when we start Recover(v, w, i), as in 2-edge connectivity, we assume that all edges covered on level $> i$ are correctly covered, and clearly this is maintained if we increment the level of a level i-edge (q, r) and call Cover$(q, r, i+1)$.

Our potential problem is a tree edge $e \in v \cdots w$ covered by a level i non-tree edge (q, r). Let $q' = meet(q, v, w)$ and $r' = meet(r, v, w)$. By symmetry we may assume that q' is closer to v than r'. Then (q, r) covers $s^w(q')q' \cdots qr$, so if Phase 1 got to q' and finished step 3, we should have found (q, r) and incremented its level. Thus $q' \cdots r' \subseteq u_1 \cdots u_2$ where u_1 are as defined in the proof of Lemma 21. In particular, $e \in u_1 \cdots u_2$. Thus e is in the biconnected component D of $G_{i+1} \cup \{(q_1, r_1), (q_2, r_2)\}$ from Lemma 21.

Generally, we have that a tree edge e is covered if and only if it is in a biconnected component with some other edge. If this applies to G_{i+1}, e is covered on level $i+1$, hence correctly covered by induction. Otherwise, e is not covered on level $i+1$, but e is in a biconnected component with other edges in D, so there is an edge (x, y) in D covering e. Since (x, y) is not in G_{i+1}, (x, y) is either (q_1, r_1) or (q_1, r_1). In either case, we know that Cover(x, y, i) has been called. □

7.2 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 levels on the links between succeeding elements such that $c^*(xyz)$ is the minimum level of a link between x and z in $c^*_{y,\cdot}(\cdot)$. Then $c^*_{y,i}(x)$ is a connected segment of $c^*_{y,\cdot}(\cdot)$. 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 weight link to its neighbors. Since both of these links are of weight 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 weight 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.

We will represent the neighbor list $c^*_{y,\cdot}(\cdot)$ using a standard balanced binary tree representation of lists admitting split and join [28, §4]. We can then easily determine $c^*(xyz)$, identify $c^*_{y,i}(x)$, or move a segment $c^*_{y,j}(z)$, in $O(\log n)$ time. Also, in $O(\log n)$ time, we can mark a segment $c^*_{u,j}(u'|u'')$ as fuzzy, implying for $x \in c^*_{u,j}(u'|u'')$ that $c^*_{u,j}(x) = c^*_{u,j+1}(x)$.

InitEdge(v, w) Link w to $c^*_{v,\cdot}(\cdot)$ on level -1 and v to $c^*_{w,\cdot}(\cdot)$ on level -1.

FreeEdge(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 $j = 0, \ldots, i$, we do as follows. If there is no fuzzy covered triple $x'yz'$ with $c^*_{y,j}(x'|z') \neq \emptyset$, we just move $c^*_{y,j}(x)$ to $c^*_{y,j}(z)$ on level j. If there is a fuzzy covered triple $x'yz'$ and $c^*_{y,j}(x'|z') \neq \emptyset$, we divide into cases. By symmetry we may assume that if $c^*_{y,j}(x)$ contains x' or z', it contains x'. If $z' \in c^*_{y,j}(z)$, we move $c^*_{y,j}(x'|z')$ and $c^*_{y,j}(z')$ to $c^*_{y,j}(x')$ on level j, and set $c^*_{y,j}(x'|z') := \emptyset$. Otherwise, if $x' \in c^*_{y,j}(x)$, we move $c^*_{y,j}(z)$ to $c^*_{y,j}(x')$ on level j. If $c^*_{y,j}(z)$ was in $c^*_{y,j}(x'|z')$, it is now deleted from $c^*_{y,j}(x'|z')$. Finally, if $x' \notin c^*_{y,j}(x)$ and $z' \notin c^*_{y,j}(z)$, we just 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 $j := i, \ldots, 0$, set $c^*_{y,j}(x|z) := c^*_{y,j}(x)$. Then move $c^*_{y,j+1}(x)$ and $c^*_{y,j+1}(z)$ to the end of neighbor list $c^*_{y,\cdot}(\cdot)$ on level -1. A note is made that xyz is the fuzzy covered triple around y.

7.3 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 $\text{cover}_C = c^*(\pi(C))$.

Biconnected(v, w) Set $C := \text{Expose}(v, w)$. Return $(\text{cover}_C \geq 0)$.

Also, cover-edge_C, cover^+_C, cover^-_C, and cover-edge^+_C are defined analogously to in 2-edge connectivity. The cover edges cover-edge_C and cover-edge^+_C are exactly the same, while cover^+_C and cover^-_C, like cover_C, now refer to covering of triples instead of edges. As for 2-edge connectivity, the information in non-path clusters will never be missing any lazy information.

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. Let v and w be tree neighbors. We say that w is an *offspring* of v if there is a non-path cluster containing v and w with v the boundary node. We then let $C(v, w)$ denote the minimal such cluster. Note that the offspring relation is anti-symmetric. Also, note that v can have at most two tree neighbors that are not its offspring. We call the cluster $C(v,w)$ above an *offspring cluster*. Then $C(v,w)$ is either the edge (v,w) if w is a leaf, or the merge of a path cluster and a non-path cluster as in Figure 1(3). We are going to use the neighbor lists to propagate counters directly from the offspring clusters to the minimal path clusters containing them, bypassing 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 offspring clusters.

- Let j and k be levels, and let C be a path cluster with $\partial C = \{v,w\}$. Let $\text{size}_{C,v,j,k}$ denote the number of internal vertices q of C such that either $q \in \pi(C)$ and $c^*(v \cdots q) \geq j$ 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 $\text{incident}_{C,v,j,k}$ 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 i be a level and let $C = C(v,w)$ be an offspring cluster. Let $\text{size}_{C,v,i}$ be the number of internal vertices q of C with $w \in v \cdots q$ such that $c^*(vw \cdots q) \geq i$, and let $\text{incident}_{C,v,i}$ be the number of (directed) non-tree edges (q,r) where q is an internal vertex of C and $c^*(vw \cdots qr) \geq i$.

Consider an edge (v,w). If (v,w) is a path cluster, it has no internal vertices, so all of the above size and incidence counters are zero. However, if v is the only boundary vertex of (v,w), (v,w) is an offspring cluster with $\text{size}_{(v,w),v,i} = 1$ for all i. Then $\text{incident}_{(v,w),v,i}$ is the number of non-tree neighbors in $c^*_{w,i}(v)$. Note that since w is a leaf in the underlying forest, any edge (w,q) covers vwq, so $c^*_{w,i}(v)$ contains (w,q) if $\ell(w,q) \geq i$.

As for 2-edge connectivity we note that when a non-tree edge gets inserted or deleted, or its level changes, we always expose its end-points so that they are not internal to any clusters. This has the convenient effect that we do not affect any of the incident-counters at the clusters until we start covering or uncovering the path between the end-points.

To get information from offspring 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 $\sum_{w \in W}(\text{Size}_{v,C(v,w),i}$ if w offspring 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, $\text{Incident}_{v,C(v,w),i}$ if w offspring of v, and 0 otherwise).

Neighbor$X(u,u',i)$ where $X \in \{$**Size, Incident**$\}$, returns $X(u, c^*_{u,i}(u'), i)$

Neighbor$X(u,u'|u'',i)$ where $X \in \{$**Size, Incident**$\}$, returns $X(u, c^*_{u,i}(u') \cup c^*_{u,i}(u'') \cup c^*_{u,i}(u'|u''), i)$.

NeighborFind(u,u',i) Finds $z \in c^*_{u,i}(u')$ such that z is either a non-tree neighbor of u or an offspring with $\text{incident}_{C(u,z),u,i} > 0$.

For each offspring w of v, the $O(\log n)$ counters of $C(v,w)$ are stored with w in the neighbor list $c^*_{v,\cdot}(\cdot)$ of v. Accumulating these counters in the binary tree representation of $c^*_{v,\cdot}(\cdot)$, we can easily support each of the above functions in $O(\log n)$ time. However, storing $O(\log n)$ counters with each binary tree node implies that list operations such as moving a segment $c^*_{v,j}(z)$ of $c^*_{v,\cdot}(\cdot)$ now takes $O(\log^2 n)$ instead of just $O(\log n)$ time. 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$(u'uu'', i)$.

C:=Merge(A, B) Suppose $\partial C = \{a\}$ and $a \in \partial A$ (Figure 1(3-4)). Then C is a non-path cluster. If A is a non-path cluster (Figure 1(4)), C is not an offspring cluster, so we are done. Otherwise (Figure 1(3)), let (u', u) the the tree edge such that $u' \in \pi(A)$, and $\{u\} = \partial A \cap \partial B$. Then for $X \in \{\text{size,incident}\}$ and $k := -1, \ldots, \ell_{\max}$, $X_{C,a,k} := X_{A,a,k,k}$ if $\text{cover}_A < k$, $X_{C,a,k} := X_{A,a,k,k}$+Neighbor$X(u, u', k)$ if $\text{cover}_A \geq k$. Let a' be the successor of a in $\pi(A)$. Then $C = C(a, a')$, so we have to update the $2\ell_{\max}$ counters associated with a' in a's neighbor list $c^*_{a,\cdot}(\cdot)$.

Suppose $\partial C = \{a, b\}$, $a \in \partial A$, and $b \in \partial B$ (Figure 1(1-2)). cover_C, cover-edge$_C$, cover^+_C, cover^-_C and cover-edge^+_C are maintained as in 2-edge connectivity. For $X \in \{\text{size,incident}\}$ and $j, k := -1, \ldots, \ell_{\max}$ compute $X_{C,a,j,k}$ as follows ($X_{C,b,j,k}$ is symmetric): If A is a non-path cluster (Figure 1(2)), set $X_{C,a,j,k} := X_{B,a,j,k}$. Otherwise if B is a non-path cluster (Figure 1(2)), set $X_{C,a,j,k} := X_{A,a,j,k}$. Finally if both A and B are path clusters (Figure 1(1)), let $u'uu''$ be the triple such that $u' \in \pi(A)$, $\{u\} = \partial A \cap \partial B$, and $u'' \in \pi(B)$. Then $X_{C,a,j,k} := X_{A,a,j,k}$ if $\text{cover}_A < j$, $X_{C,a,j,k} := X_{A,a,j,k}$+Neighbor$X(u, u', k)$ if $\text{cover}_A \geq j \wedge c^*(u'uu'') < j$, and finally $X_{C,a,j,k} := X_{A,a,j,k}$+Neighbor$X(u, u'|u'', k) + X_{B,u,j,k}$ if $\text{cover}_A \geq j \wedge c^*(u'uu'') \geq j$.

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, i, u').
- $D :=$ Expose(q, r).
- Let (q, q') and (r', r) be the end edges on $q \cdots r$
- If $\text{size}_{D,q,-1,i+1} + 2$+NeighborSize$(q, q', i+1)$+NeighborSize$(r, r', i+1) > n/2^{i+1}$,
  - 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'.

$u := \text{FindBranch}(v, C, i)$.

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 $\text{incident}_{C,a,-1,i} = 0$ 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 $u \in \partial A \cap \partial B$. If $\text{incident}_{A,a,-1,i} > 0$ then return FindBranch(a, A, i). Otherwise if NeighborIncident$(u, u'|u'', i) > 0$ then return u else return FindBranch(u, B, i).

VertexFind(u, i, u') Let $z :=$NeighborFind(u, u', i). If z is a non-tree neighbor, return (u, z). Otherwise, if $C(u, z)$ is the edge (u, z), return (z, r) where r is a non-tree neighbor of z in $c^*_{z,i}(u)$. Otherwise $C(u, z)$ has two children A and B with $u \in A$, $A \cap B = \{b\}$. If $\text{incident}_{A,u,i,i} > 0$, return PathFind$(u, A, i)$. Otherwise, return VertexFind(b, i, b') where b' is the predecessor of b in $u \cdots b$.

PathFind(a, C, i) Call Clean(C). If C has only one path child A, return PathFind(a, A, i). Otherwise, let A be the path child nearest to a and B be the other path child. If $\text{incident}_{A,a,-1,i} > 0$ then return PathFind(a, A, i). Else let b be the boundary vertex nearest to a in B and b' be the predecessor of b on $a \cdots b$. If NeighborIncident$(b, b', i) > 0$, return VertexFind(b, i, b'), else return PathFind(b, B, i).

Theorem 23 *There exists a deterministic fully dynamic algorithm for maintaining biconnectivity in a graph, using $O(\log^5 n)$ amortized time per operation.*

Proof: Relative to 2-edge connectivity, the essential new cost is when we cover a triple. This moves $O(\log n)$ segments in the neighbor list of the center of the triple. Each move affects $O(\log n)$ binary tree nodes in the representation of the neighbor list, and each of these tree nodes has $O(\log n)$ counters associated with it, so the cost of covering a triple is $O(\log^3 n)$. This in turns means that Clean(C) and Split(C) now takes $O(\log^3 n)$ time, as opposed to the $O(\log^2 n)$ time in 2-edge connectivity. As a result, our total operation cost is increased by a factor $O(\log n)$ to $O(\log^5 n)$. □

Our fully dynamic biconnectivity algorithm uses $O(m + n\log^2 n)$ space. As for 2-edge connectivity, we can improve the space to $O(m + n \log n)$ and the query time to $O(\log n)$. It requires, however, that we use biased search trees [26] for the neighbor lists $c^*_{v,\cdot}(\cdot)$, giving the at most two non-offspring tree neighbors maximal bias, and giving offspring tree neighbors bias proportional to the size of their offspring clusters, that is, an offspring cluster $C(v, w)$ gets bias $\text{size}_{C(v,w),v,0}$. Thorup's [30] improvement by a factor $O(\log n/ \log\log n)$ also works here. In fact it seems that the techniques from [30] can save a further factor $O(\log n)$ by providing a kind of biased deletion for the covering and uncovering of triples. This would then lead to an amortized update time of $O(\log^3 \log\log n)$ with a query time of $O(\log n)$ and a space bound of $O(m + n \log n \log\log n)$, as for 2-edge connectivity. Details of this will be presented in the journal version of [30].

Finally we note that it is not difficult to augment our biconnectivity algorithm to provide articulation points using the same principles as was used to augment the 2-edge connectivity to provide bridges.

8 Concluding remarks

Deterministic fully dynamic algorithms with polylogarithmic *amortized* operation costs have been presented for connectivity, minimum spanning forest, 2-edge, and biconnectivity. It remains a major open problem such feasible bounds can be achieved in the *worst-case*, where currently, the best known is $O(\sqrt{n})$ per update [5, 7]. Another, major challenge is to find good algorithms for directed graphs. Recently, it has been settled that one can maintain the transitive closure of a digraph in $O(n^2)$ time per operation [3, 23]. This is optimal in the sense that one update can make $\Omega(n^2)$ changes to the transitive closure. However, if the problem is just to maintain reachability between to fixed vertices s and t, no solution better than the static is known.

Acknowledgment

We would like to thank Haim Kaplan for careful comments to an earlier draft of the document. Also, we would like to thank Majerech Vladan for a very careful reading of the paper, and in particular for discovering the high cost of covering triples, leading to an amortized operation cost of $O(\log^5 n)$ rather than the $O(\log^4 n)$ mistakenly claimed in [21].

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th Int. Coll. Automata Languages and Programming (ICALP), LNCS 1256*, pages 270–280, 1997.

[2] T.C. Biedl, P. Bose, E.D. Demaine, and A. Lubiw. Efficient algorithms for Petersen's matching theorem. *J. Algorithms*, 38:110–134, 2001.

[3] C. Demetrescu and G. Italiano. Fully dynamic transitive closure: Breaking through the $o(n^2)$ barrier. In *Proc. 41st IEEE Symp. on Foundations of Computer Science*, pages 381–389, 2000.

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

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

[6] T. Feder and M. Mihail. Balanced matriods. In *Proc. 24th ACM Symp. on Theory of Computing (STOC)*, pages 26–38, 1992.

[7] 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.

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

[9] G. N. Frederickson and M. A Srinivas. Algorithms and data structures for an expanded family of matroid intersection problems. *SIAM J. Computing*, 18(1):113–139, 1989.

[10] M. Fredman and M.R. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[11] H.N. Gabow, H. Kaplan, and R.E. Tarjan. Unique maximum matching algorithms. In *Proc. 31st ACM Symp. on Theory of Computing*, pages 70–78, 1999.

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

[13] M. R. Henzinger. Improved data structures for fully dynamic biconnectivity. *SIAM J. Computing*, 29(6):1761–1815, 2000. See also STOC'94.

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

[15] M. R. Henzinger and V. King. Fully dynamic 2-edge connectivity algorithm in polygarithmic time per operation. Technical Report SRC 1997-004a, Digital, 1997.

[16] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th Int. Coll. on Automata, Languages, and Programming (ICALP), LNCS 1256*, pages 594–604, 1997.

[17] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(502–536), 1999. See also STOC'95.

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

[19] 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.

[20] M.R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.

[21] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th ACM Symp. on Theory of Computing*, pages 79–89, 1998.

[22] R.D. Iyer, D. Karger, H.S. Rahul, and M. Thorup. An experimental study of poly-logarithmic fully-dynamic connectivity algorithms. In *Proc. 2nd Workshop on Algorithms Engineering and Experiments (ALENEX)*, 2000.

[23] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, pages 81–89, 1999.

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

[25] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *J. Computer and System Sciences*, 26(3):362–391, 1983. See also STOC'81.

[26] W.W. Bent D.D. Sleator and R.E. Tarjan. Biased search trees. *SIAM J. Computing*, 14(3):545–568, 1985.

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

[28] R. E. Tarjan. *Data Stuctures and Network Algorithms.* SIAM, 1983.

[29] M. Thorup. Decremental dynamic connectivity. *J. Algorithms*, 33:229–243, 1999. See also SODA'97.

[30] M. Thorup. Near-optimal fully-dynamic connectivity. In *Proc. 32nd ACM Symp. on Theory of Computing*, pages 343–350, 2000.

[31] M. Thorup and D. Karger. Dynamic graph algorithms with applications (invited talk). In *Proc. 7th Scandinavian Workshop on Algorithms Theory (SWAT), LNCS 1851*, pages 1–9, 2000.

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