SRC Technical Note
1997 - 021

September 30, 1997

Certificates and Fast Algorithms for Biconnectivity in Fully-Dynamic Graphs

Monika Rauch Henzinger and Han La Poutré

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Han La Poutré is at the Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. His electronic mail addresses is: Han.La.Poutre@cwi.nl

A preliminary version of this work appeared in the Proceedings of the Third Annual European Symp. on Algorithms – ESA'95.

Abstract

In this paper, we present sparse certificates for biconnectivity together with algorithms for updating these certificates. We thus obtain fully-dynamic algorithms for biconnectivity in graphs that run in $O(\sqrt{n}\log n\log\lceil\frac{m}{n}\rceil)$ amortized time per operation, where m is the number of edges and n is the number of nodes in the graph. This improves upon the results in [12], in which algorithms were presented running in $O(\sqrt{m}\log n)$ amortized time, and solves the open problem to find certificates to speed up biconnectivity, as stated in [2].

1 Introduction

The field of dynamic graph algorithms has become an important field in algorithmic research in recent years. Currently, several results exist for incremental and fully-dynamic graph problems, like for maintaining spanning trees, the 2-edge- or the 2-vertex-connected components of a graph, or the planarity of a graph under the insertions and/or deletions of edges and vertices [3, 4, 5, 7, 8, 9, 10, 11, 12, 14].

In [4, 5, 12], algorithms for maintaining minimum spanning trees and the connectivity, 2-edge-connectivity and the 2-vertex-connectivity relations in fully-dynamic graphs were presented that run in O($\sqrt{m}$) or $O(\sqrt{m}\log n)$ time per operation (amortized time for 2-vertex-connectivity). (In this paper, n is the number of nodes and m is the number of edges.) In the meantime, in [2], the concept of certificates and sparsification trees (for definitions, see Section 2) was introduced to speed up several fully-dynamic graph algorithms. In particular, sparse certificates could be used for speeding up fully-dynamic algorithms for maintaining minimum spanning trees and the connectivity and 2-edge-connectivity relations in graphs [4, 5], to O($\sqrt{n}$) time per operation. Basically, these sparse certificates were defined in terms of (successive, minimum) spanning trees, and were maintained by applying Fredericksons minimum spanning trees data structure [4]. The algorithm to be speeded up was thus used on the resulting certificate for the whole graph (viz., in the root of the sparsification tree).

However, this approach seems not to work for certificates for biconnectivity. Like for many other problems involving k-connectivity, such as designing static, parallel, incremental, or fully-dynamic algorithms, 2-vertex-connectivity appears to be substantially harder to deal with than 2-edge-connectivity. An example of this can also be observed in [1], considering parallel algorithms for k-connectivity, where sparse *static* certificates for biconnectivity are defined in terms of breadth-first trees, whereas those for 2-edge-connectivity can consist of any kind of spanning trees. Up to now, no efficient fully-dynamic algorithms for maintaining breadth-first trees are known, and it is commonly felt that these trees are hard to maintain indeed. So, designing certificates for fully-dynamic biconnectivity and thus speeding up the fully-dynamic algorithms for it is an appealing open problem [1, 2, 12].

In this paper, we present new, sparse certificates for biconnectivity that can efficiently be maintained under insertion and deletion of edges. We thus obtain $O(\sqrt{n}\log n\log\lceil\frac{m}{n}\rceil)$ amortized time algorithms for maintaining the biconnectivity relation in fully-dynamic graphs, and therefore show that fully-dynamic biconnectivity falls in the same (current) time complexity class as fully-dynamic 2-edge-connectivity.

We introduce sparse certificates that are determined in a relaxed, history-dependent way, and that thus are not defined in a more mathematical, "static" way, like in [2]. In particular, the certificates we use are not "stable", as defined and used throughout in [2] (see also Definition 2.1). Also, we develop extensions of the algorithms that must be speeded up themselves [12] as well, to be able to maintain the certificates. Thus, our approach also appears to be the first combination of certificates and maintenance algorithms that also uses the algorithm that must be "speeded up" itself to maintain the certificates, and not just Fredericksons minimum spanning tree algorithms.

In our solutions, the (on-line) sequence of update operations is split into two parts, which are treated separately: one concerning deletions and one concerning insertions. Viz., we typically start with subgraphs of which we only want to maintain the certificates in a *decremental* way: after any (delete) operation, roughly either the number of biconnected components increases, or we find some edges to "repair" the biconnectivity relation in the certificate of the subgraph (with some additional constraints). Insertions of edges are then temporarily processed outside the sparsification paradigm, and from time to time handled further in a batch-like way.

Thus, our relaxation of (non-stable) certificates, usage of sparsification trees, and the separation of decremental subproblems are new. We conjecture, that our approach may lead to fast algorithms for various other fully-dynamic graph problems as well, like k-vertex connectivity for $k \geq 3$.

The paper is organized as follows. Section 2 contains the preliminaries, including a description of sparsification trees. In Section 3, we define certificates for biconnectivity, and give algorithms for maintaining them. In Section 4, the replacement data structure, used in Section 3, is described. In this extended abstract, we omit many details and special cases.

2 Preliminaries

For a graph G, two nodes x and y are called *biconnected* (or 2-vertex-connected) if the deletion of a node from G does not separate x and y in G (i.e., x and y are still connected).

For a tree T, we denote by $\pi_T(x, y)$ the (unique) path between nodes x and y in T.

We give some definitions concerning certificates, as occurring in [1, 2]. For any graph property P, and graph G, a *certificate* for G is a graph G' such that G has property P if and only if G' has property P. A *strong* certificate for G is a graph G' such that, for any graph H, $G \cup H$ has property P if and only if $G' \cup H$ has property P.

A property is said to have *sparse* certificates if there is some constant c such that for every graph G on an n-vertex set, we can find a strong certificate for G with at most cn edges.

In [2], some sparse certificates are given for k-edge-connectivity, minimum spanning trees, and bipartiteness, all depending on the data structures for minimum spanning trees presented by Frederickson [4, 5]. Also, the concept of stable certificate is defined, which is important for the use of certificates in sparsification trees, as follows.

Definition 2.1 *Let A be a function mapping graphs to strong certificates. Then A is* stable *if it has the following properties:*

1. For any graphs G and H, $A(G \cup H) = A(A(G) \cup H)$.

2. For any graph G and edge e in G, $A(G - e)$ differs from $A(G)$ by $O(1)$ edges.

However, for fully-dynamic biconnectivity, this definition seems to be too strict. Amongst others, it supposes that for each graph, a (unique) certificate can be chosen, which then has to be maintained by the dynamic algorithms with only $O(1)$ changes (see [2]). We seem to need a more liberal concept of certification.

2.1 Sparsification tree

We sketch how certificates can be used in sparsification trees [2] to maintain a property P. Our final strategies will be somewhat different, though.

As in [2], maintain a partition of the graph edges in $\lceil \frac{m}{n} \rceil$ groups, all but one containing exactly n edges. The remaining group is called the small group.

Insertion of an edge in the graph is always done in the small group. Deletion of an edge is performed in the proper group, after which an edge from the small group is transferred to this group (via a deletion and insertion in these groups, respectively). If the small group becomes empty, we delete it; if it contains n edges and we want to insert a new edge, we start a new group.

We form and maintain a *sparsification tree*, which is a binary tree of height $O(\log\lceil\frac{m}{n}\rceil)$, with $\lceil\frac{m}{n}\rceil$ leaves corresponding to the $\lceil\frac{m}{n}\rceil$ groups. Each node x in the sparsification tree corresponds to a subgraph $G(x)$ of G formed by the edges in the groups at the leaves that are the descendants of that node x. Also, to node x, a subgraph $S(x)$ of $G(x)$ is related, such that $S(x)$ has property P iff $G(x)$ has property P ($S(x)$ is actually a sparse certificate for $G(x)$). If x is a leaf, then $S(x) = G(x)$. We say that the edges in $S(x)$ are the *edges related to* x. Furthermore, a sparse certificate $C(x)$ (for $G(x)$ or $S(x)$) is maintained. The subgraph $S(x)$ related to node x is found by forming the union of the certificates $C(y)$ and $C(z)$ of the two child nodes y and z. The certificate $C(x)$ is found and maintained in $S(x)$. Thus, the certificate at the root is the certificate of the whole graph. For further details and elaboration, we refer to [2].

3 Certificates for biconnectivity

In this section, we present sparse certificates for biconnectivity together with algorithms for updating these certificates, and handling sparsification trees.

3.1 Monotone sparse certificates and their maintenance

Consider a graph G of n nodes. For a spanning tree T of graph G and a sequence of (ordered) edges e_i ($1 \leq i \leq l$), the extension graph TT_j ($0 \leq j \leq l$) consists of all the edges e_i with $1 \leq i \leq j$. We call the sequence of edges an *add-on sequence* if for every j, $0 \leq j < l$, the graphs $T \cup TT_j$ and $T \cup TT_{j+1}$ have a different number of biconnected components. Hence, the number of components in TT_{j+1} is at least one smaller than the number of components in $T \cup TT_j$.

Lemma 3.1 *Let G be a graph, T a spanning tree of G, and S an add-on sequence for biconnectivity. Then S contains at most $n-1$ edges.*

Proof: $TT_0 = T$ contains exactly n different biconnected components. Since any graph contains at least one biconnected component, it follows by the definition of add-on sequence that S contains at most $n-1$ edges. ▮

We first describe a certificate and the maintenance of it for some appropriate graph G.

A *dynamic color partition* of G colors the edges of G from three colors viz. blue, red, and green.

For convenience in this abstract, we assume in this subsection that G always is a connected graph (this is not essential, though). Blue and red edges have a *cost* related to them, which is 0 for blue edges and which is a natural number otherwise. No two red edges have the same cost.

At any time, the blue edges form a spanning tree of G, and the red edges are $O(n)$ non-tree edges that form an add-on sequence if ordered according to their cost. In addition, this add-on sequence is *maximal*, i.e., such that $T \cup TT_l$ contains the same (number of) biconnected components as G. The blue and red edges together form the certificate for G. Hence, this is a *sparse certificate* for G.

We give algorithms for maintaining the sparse certificate of G under deletions and certain insertions of edges. The edge insertions are only allowed if they do not change the biconnectivity relation. Therefore, the changes in the biconnectivity relation are monotone for a sequence of these operations, i.e., biconnected components are only split up and are never joined. Thus, we call this sparse certificate a *monotone sparse certificate*. (We want to point out that the restriction on insertions only applies to maintaining the certificates themselves, and not to the overall algorithms presented in Subsection 3.2.)

Here and in the sequel, we use (implicitly) an operation that turns a given green edge into a red edge, while simultaneously labelling this edge with a cost which is the number of preceding operations.

3.1.1 Initialisation.

We can construct a sparse certificate for G as follows. Initially, all edges are of G are green. First, a breadth-first search tree of G is constructed, and all the edges in it are made blue. This blue tree is denoted by T_b. The graph G_r consisting of red edges is then incrementally constructed as follows. First, a breadth-first search forest B of $G \setminus T_b$ is made and the edges are enumerated in some way. A (green) edge of B is converted into a red edge only if it changes the number of components of $T_b \cup G_r$. This can be done by maintaining the biconnected components dynamically [8, 14], thus yielding an add-on sequence indeed. As was shown in [1], the graph $T_b \cup B$ is a certificate for biconnectivity, hence, so is $T_b \cup G_r$. (We do not really need the breadth-first forest, but can do this with all the existing edges once they are ordered as well.)

The edges in G_r are given a cost which corresponds to their position in the add-on sequence.

3.1.2 Insertions and Deletions.

The algorithms for maintaining the certificate processes deletions and insertions as follows.

Edge insertions into G are only allowed if they do not change the biconnectivity relation. The algorithm inserts a new edge as a green edge.

When an edge is deleted, the following may happen to the color of edges. After the deletion of one red edge, some green edges may become red. After the deletion of a blue edge, one red edge may become blue and some green edges may become red. And finally, the deletion of a green edge is done without consequences for other edges and their colors.

For the deletion of an edge, we have the following more detailed strategy.

Whenever a blue edge e is deleted, it is replaced by the minimum cost red edge e' which restores a blue spanning tree of G. This is processed by first interchanging the colors of e and e', and subsequently deleting the (now) red edge e, as in the case below.

Whenever a red edge (x, y) is deleted, a node a that now is a new articulation point for x and y in the *current* $T_b \cup G_r$ is generated. Such a node is called a *potential* articulation point. Then, for potential articulation point a, it is tested whether it is also an articulation point of G: If it is, we call it a *definite* articulation point. Otherwise, (i.e., if not,) a green edge (x, y) is turned red such that a is not an articulation point in $T_b \cup G_r$ any more, and G_r is updated accordingly. Successively, the next potential articulation point is generated for the current (possibly updated) $T_b \cup G_r$ (if any), and the above process is repeated until no unprocessed potential articulation points for x and y exist.

Lemma 3.2 *At any moment, the current red edges (in the order of their costs) form an add-on sequence for biconnectivity.*

Proof: We distinguish the three basic changes with respect to red and blue edges.

1. Obviously, if a red edge is deleted from the add-on sequence, the remaining sequence still is an add-one sequence.

2. When a red edge $e = (x, y)$ and a blue edge $d = (u, v)$ interchange colors, then $d = (u, v) \in \pi_{T_b}(x, y)$. We show that replacing e by d in the add-on sequence (while interchanging the numbers related to these edges as well) yields another add-on sequence.

Let $S = e_1, ..., e_k$ be the red add-on sequence before deletion of d, where $e = e_j$. Let T_1 and T_2 be the two subtrees remaining after deleting d from T_b, and let T_b' be $T_1 \cup T_2 \cup \{(x, y)\}$ (the new spanning tree).

Let the sequence S' equal S where e is replaced by d. Then S' is an add-on sequence for T_b', which is seen as follows. Note that $T_b \cup TT_j$ equals $T_b' \cup TT_j'$. Hence, $T_b \cup TT_i = T_b' \cup TT_i'$ for $i \geq j$, and hence the graphs $T_b' \cup TT_i'$ and $T_b' \cup TT_{i+1}'$ have a different number of biconnected components, for $j \leq i < k$. Furthermore, for $i < j$, an edge e_i must have both its end points in either T_1 or T_2, because $e = e_j$ was the minimum cost edge connecting T_1 and T_2. Therefore, since $T_b \cup TT_i$ and $T_b \cup TT_{i+1}$ have a different number of biconnected components for $0 \leq i < j$, and since T_b and T_b' equal on their subtrees T_1 and T_2 and differ only in the edges e and d, it follows that the graphs $T_b' \cup TT_i'$ and $T_b' \cup TT_{i+1}'$ have a different number of biconnected components, for $0 \leq i < j$.

Hence, the (new) sequence of red edges is an add-on sequence for T_b'.

3. When a green edge $e = (x, y)$ is turned red, the thus extended new sequence of red edges obviously is an add-on sequence again, since otherwise this edge would not have been added. ▮

Corollary 3.3 *During a sequence of d deletions of edges, at most $d + 2n - 2$ edges have been blue or red for some time.*

Proof: The total number of blue or red edges that have existed (as blue or red edge) is at most $2(n-1)$ (the final number of blue and red edges) plus d (the number of deleted edges). ▮

We refer to the above algorithms for maintaining the certificates as *certificate algorithms*.

3.1.3 Data structures for monotone sparse certificates.

To determine which edges change color, as described above, we use the following dynamic data structures, which allow deletions, restricted insertions, and color changes of edges as described above.

1. We keep the graph $T_b \cup G_r$ in a dynamic minimum spanning tree data structure of Frederickson [4] with all blue edges having cost 0. Whenever a blue edge is deleted, this data structure provides us with the minimum cost red edge. This edge becomes blue.

2. We keep $T_b \cup G_r$ in a dynamic biconnectivity data structure of Rauch [12]. Whenever a blue or red edge is deleted, this data structure provides us with the new potential articulation points generated one at a time, in time proportional to the number of actually generated potential articulation points.

3. We keep G in a dynamic biconnectivity data structure of Rauch [12]. This allows to test for every new potential articulation point a of $T_b \cup G_r$ whether it is also an articulation point of G. If not, some green edge (x, y) such that $a \in \pi_{T_b}(x, y)$ should become red, as above.

4. We keep a *replacement data structure* that provides such a green edge. It is described in the next section.

By [12], the operations on the data structures 1 and 2 can be performed in $O(\sqrt{n} \log n)$ time per returned edge or node, since $T_b \cup G_r$ has $O(n)$ edges, and the operations on data structure 3 can be performed in $O(\sqrt{m} \log n)$ time, while the operations on data structure 4 can be performed in $O(\sqrt{m} \log n)$ time, by Theorem 4.2.

In the following, a certificate operation denotes the insertion, deletion or color change of an edge, the generation of a new potential articulation point or the other queries on the data structures as described above.

Lemma 3.4 *A certificate operation can be performed in $O(\sqrt{m} \log n)$ time.*

3.2 The overall algorithms and data structures

We can now combine the above certificate and certificate algorithms with sparsification as follows. (For terminology, we refer to Subsection 2.1.)

The edges in the graph are partitioned into yellow and non-yellow. The sparsification tree "contains" only non-yellow edges. (A non-yellow edge has (local) color(s) defined at relevant sparsification nodes.) Each leaf corresponds to a group of at most n non-yellow edges. At *each* node x of the sparsification tree, the above certificate algorithms are executed locally on the graph $S(x)$, to maintain the certificate $C(x)$ in $S(x)$. Note that, thus, $S(x)$ has a color partition, with just the colors blue, red, and green, where the colors are defined "locally" in $S(x)$ (independent of the actual colors red and blue in e.g. $C(y)$ and $C(z)$ in the children y and z, if any), and where thus $C(x)$ consists of the blue and red edges of $S(x)$.

Whenever a new edge e is inserted in the graph, keep it as a yellow edge in the "yellow delay set" Y. If the total number of yellow edges exceeds $O(n)$, we create a new leaf x with $S(x)$ consisting of these yellow edges, then color these edges green and run the certificate initialization algorithm on $S(x)$. Subsequently, we "build/rebuild" every node on the root path of x.

Whenever an edge e is deleted, it is deleted from the proper leaf x, i.e., from $S(x)$. The deletion from x might cause that at most $n - 1$ green edges turn red at x. Let y be the parent of x in the sparsification tree. First add all the new red edges of x as green edges to y, called "forced insertions". (Note that forced insertions do not change the biconnectivity relation of $S(y)$.) Subsequently, if y contains e, delete e from y (where again, green edges may turn red) and (recursively) repeat this procedure for the parent of node y, until e is not deleted from a tree node any more or until we reach the root.

Whenever there are more than $2\lceil \frac{m}{n} \rceil$ leaves, rebuild the entire sparsification tree.

Thus, the certificate $C(G)$ for G is given by $C(root) \cup Y$, i.e., the certificate related to the root, joined with the existing yellow edges in the yellow delay set. This obviously is a sparse certificate again. We run the biconnectivity algorithms of [12] on this for maintaining $C(G)$ and for computing the queries asked.

Lemma 3.5 *Let x be a node of the sparsification tree. At any time during a sequence S of operations, the number of edges related to x is at most* $4(n-1)$*, and the number of red and blue edges is at most* $2(n-1)$*. During sequence S, at most $d + 2n - 2$ edges have been blue or red for some time, and at most $d + 4n - 4$ edges have been related to x, where d is the number of edge deletions in the node x.*

Proof: Follows from the fact that the edges related to X consist of two spanning trees and two add-on sequences and from Corollary 3.3. ▮

Note that (new) green edges are added to a sparsification node only if they do not change the biconnectivity relation indeed (at the very moment of their insertion), as mentioned before.

Lemma 3.6 *For a sparsification node x, since its last (re)building, the total number of new, definite articulation points in $S(x)$ is $O(n)$, and the total number of certificate operations in $S(x)$ is at most $O(n + d)$, where d is the number of edge deletions in node x.*

Proof: The first statement follows since definite articulation points do not vanish any more. For the second statement, we charge the cost of obtaining a potential articulation point that vanishes again by making a green edge red, to this color change; We observe that color changes only occur from green to red to blue; and we use Lemma 3.5. ▮

Theorem 3.7 *There exists a fully-dynamic algorithm for biconnectivity in graphs such that a sequence of s operations starting from the empty graph takes $O(s.\sqrt{n} \log n \log\lceil \frac{m}{n} \rceil)$ time, while each query takes $O(1)$ time, and where n is the current number of nodes. Thus, each operation takes $O(\sqrt{n} \log n \log\lceil \frac{m}{n} \rceil)$ amortized time, and each query takes $O(1)$ worst-case time.*

Proof: The number of certificate operations in a sparsification node x since its last (re)building is $O(n+d)$, where d is the number of (local) deletions in that node. Charge the $O(n)$ operations to the rebuilding (initialisation) at x, and charge each occurring local deletion for one operation. Thus, each "global" deletion is charged for $O(\log \frac{m}{n})$ certificate operations in total (viz., for the sparsification nodes that all lie on a root path). Furthermore, maintaining the graph $S(root) \cup Y$ (of $O(n)$ edges) takes time proportional to maintaining $S(root)$ and $O(1)$ amortized certificate operations per yellow edge. We charge the cost of the former to maintaining $S(root)$ and the cost of the latter to the insertion of a yellow edge, viz., $O(1)$ certificate operations per operation. Finally, each insertion of an edge is now charged for $O(\log \frac{m}{n})$ amortized operations in total, by the delay-build/rebuild strategy. Each such certificate operation takes $O(\sqrt{n} \log n)$, since the occurring graphs have $O(n)$ edges, and by [12] and Theorem 4.2.

Adding up the number of certificate operations and using [12], yields the theorem. ▮

4 The replacement data structure

In this section, we describe the replacement data structure, as mentioned in Subsection 3.1.3.

We are given a graph G of blue, red, and green edges such that the blue edges form a spanning tree T_b of G (for convenience, we assume that G is connected). We refer to an edge of T_b just as *tree edge* and denote the tree path $\pi_{T_b}(x, y)$ just by $\pi(x, y)$.

Let P be a path in T_b and let b_1, a, and b_2 be three consecutive vertices on P such that a is an articulation point in $T_b \cup G_r$ that separates b_1 and b_2. We say a green edge *e covers a* on P iff a does not separate b_1 and b_2 in $T_b \cup G_r \cup e$.

We describe the functionality of the replacement data structure. On the one hand, it is able to perform the deletions, restricted insertions, and color changes of edges as described in Subsection 3.1.3. On the other hand, it can return the appropriate green edges as described in Subsection 3.1.3. We describe this operation in more detail.

Let the red edge (u, v) have to be deleted. The replacement data structure is given a (newly generated) potential articulation point a of $\pi(u, v)$ on the current $T_b \cup G_r$ that is *not* an articulation point on $\pi(u, v)$ in G (called *candidate*). Then for candidate a, it outputs a green edge covering a on $\pi(u, v)$.

In this section, we assume that each blue or red edge has cost 0 and each green edge has cost 1.

Now, first assume we would maintain the following (too costly) data structure, namely, for each node x the minimum spanning tree $F(x)$ of $G \setminus x$. Note that two neighbors of x are biconnected iff they are connected in $F(x)$. Since the blue and red edges form a sparse certificate of G before an edge deletion, all edges in F are blue or red. The deletion of a blue or red edge removes at most one edge from F and adds at most one, potentially green, edge. This edge is a green edge covering a, i.e. it is an edge that the replacement data structure is looking for.

However, it is too expensive to maintain for each node x the minimum spanning tree $F(x)$ of $G \setminus x$. Thus, we decompose the graph into subgraphs, called *clusters*, which are connected by blue edges, and maintain for each cluster a data structure similar to the one described above. Since the nodes in a cluster are connected by a spanning tree containing only blue edges, we have to focus on edges between clusters.

4.1 Graph decomposition

We expand every node of G with degree $d > 3$ into d nodes that are connected by a chain of $d-1$ *dashed* edges. We naturally expand T_b to be a spanning tree of the expanded graph G' (where all dashed edges thus are in T_b).

We decompose G' as in [12]. A *cluster* is a set of vertices that induces a connected subgraph of T. An edge is *incident* to a cluster if exactly one of its endpoints is in the cluster. A *restricted partition of order k* with respect to T is a partition of the vertices so that

1. Each set in the partition is a cluster that is incident to ≤ 3 tree edges and contains $\leq k$ vertices.

2. A cluster that is incident to 3 tree edges contains exactly one vertex.

3. If a cluster is incident to a dashed edge, then all tree edges incident to the cluster are incident to the same vertex of G.

4. No two adjacent clusters can be combined and still satisfy 1 to 3.

The partition splits G into $O(m/k)$ clusters of size $\leq k$ and is found in time $O(m+n)$ [5]. We denote by $C(x)$ a cluster containing a representative of a vertex x and say that $C(x)$ *contains* x. If the representatives of x are contained in > 1 clusters, then x is a *shared vertex* and all clusters $C(x)$ are called x*-clusters* and *share* x. By condition 3 every cluster shares ≤ 1 vertex. Since each dashed edge between two clusters is a tree edge, there are $O(m/k)$ shared vertices.

This decomposition induces the following *graph* H_1 *of clusters*. Two vertices C and C' of H_1 are connected by an edge (respectively blue edge) if and only if there is an edge (respectively blue edge) between a vertex of C and a vertex of C'. If there is no blue edge between a vertex of C and C', but a red edge, then C and C' are connected by a red edge. If there is neither a blue nor a red, but a green edge between them, they are connected by a green edge.

We define H_2 to consist of H_1 with all dashed edges contracted. For a non-shared vertex x, the unique cluster $C(x)$ is represented by a unique node of H_1. For a shared vertex x, all clusters $C(x)$ are represented by a unique vertex in H_2.

For $i = 1, 2$ we denote by H_i^{br} the subgraph of H_i induced by blue and red edges. The blue edges form a spanning tree of H_i and of H_i^{br}. Two clusters that are adjacent in this spanning tree are called *tree neighbors.*

4.2 Outline of the data structure

For $i = 1, 2$ we maintain H_i and H_i^{br} dynamically in the high-level data structure of [12] [1]. For a graph H' (with H' either H_i or H_i^{br}), the high-level data structure of [12] maintains for each node C in H' the following graph $H'(C)$: $H'(C)$ contains a node for each tree neighbor of C. There is an edge between two tree neighbors L_1 and L_2 of C iff there is an edge in $H' \setminus C$ between the subtree containing L_1 and the subtree containing L_2. The data structure in [12] can be extended to label an edge (L_1, L_2) in $H'(C)$ with an edge in $H' \setminus C$ connecting the subtree of L_1 with the subtree of L_2.

As shown in [12], all graphs $H'(C)$ can be maintained in time $O(k+(m/k)/logn)$ per edge insertion in G or edge deletion in G. Augmenting the data structure to label edges of $H'(C)$ with edges of H' does not increase the running time. A connectivity query in $H'(C)$ can be answered in constant time. Note that two tree neighbors of C are connected in $H'(C)$ iff they are biconnected in H.

We use the high-level data structures of H' to

1. test in constant time if two tree neighbors of a node C are biconnected in H',
2. output all biconnected tree neighbors in H' of a node C in time linear in their number and a spanning forest connecting them in $H'(C)$,

If there is an edge between L_1 and L_2 in $H_i(C)$, but not in $H_i^{br}(C)$, then all edges in $H_i \setminus C$ between the subtree containing L_1 and the subtree containing L_2 are green. Thus, using the high-level data structure of H_i and of H_i^{br}, we can determine the minimum cost edge in H_i connecting the subtree containing L_1 and the subtree containing L_2. Given the spanning forests connecting the tree neighbors of C in

[1]The full version of this paper is available at http:\\www.research.digital.com\SRC\personal\monika\papers.html

$H_i(C)$ and $H_i^{br}(C)$, we can compute a minimum spanning forest connecting the tree neighbors of C in $H_i(C)$ in time linear in the number of tree neighbors. A query to determine this minimum spanning forest is called a *minimum spanning query for C*.

As mentioned above, new shared vertices can be created during the sequence of operations. To distinguish them from the others, we call all shared vertices that are not new, *old*. There are $O(k)$ vertices in all clusters sharing a new vertex.

In the next subsections, we will describe how we find a green covering edge for different types of candidates. Again, we have the high-level data structures of [12] as the base, but we modify and extend it. We will consider a candidate a, where b_1 and b_2 are the neighbors of a on $\pi(u, v)$. We distinguish between a being a "non-shared vertex", a "new shared vertex", or an "old shared vertex."

4.3 Non-shared candidates or new shared candidates

To cover a non-shared or new shared candidate a with a green edge e, we first build a graph $G(a)$ of size $O(k)$ and then we determine e using $G(a)$. Both steps take time $O(k \log n + m/k)$. We do this as follows. For a non-shared candidate let H' denote H_1, for a new shared candidate let H' denote H_2. Let C_a denote representing (all) C_a in H'.

Let C_a be incident to j tree edges. The graph $G(a)$ contains as nodes all the nodes of C_a and one additional node for each tree neighbor L_j of C_a in H', called *cluster-node*. The graph $G(a)$ contains as edges (1) all the edges between two nodes of C_a, (2) an edge (x, L_j) for each edge (x, y) incident to a node $x \in C_a$ if $\pi(x, y)$ contains the tree edge between C_a and L_j, and (3) the edges of a minimum spanning tree in $H'(C_a)$ (between the tree neighbors of C_a).

The graph $G(a)$ can be built as follows: The edges of (1) can be found by inspecting the edges incident to nodes of C_a in time $O(k)$. The edges of (3) can be found in time $O(1)$ per edge using a minimum spanning query for C. To determine the edges of (2), we first store the spanning tree of H' in a dynamic tree data structure. Then we scan all edges incident to nodes in C_a to find every edge (x, y) with $x \in C_a$ and $y \notin C_a$. For each such edge we determine the corresponding tree neighbor of C_a. This determines in time $O(\log n)$ the edge (x, L_j) that belongs to $G(a)$. Since $O(k)$ edges have at least one endpoint in C_a, the total time is $O(k \log n + m/k)$.

Note that b_1 and b_2 are either contained in $G(a)$ or represented by nodes L_j. Since a is a candidate, b_1 and b_2 or their representatives are disconnected in the graph induced by the blue and red edges of $G(a) \setminus a$, but connected in $G(a) \setminus a$. Thus, there exists a cut in $G(a) \setminus a$ separating b_1 and b_2 or their representatives that only contains green edges. Any one edge e of the cut covers a in $G(a)$. The cut can be found in time $O(k)$ by executing a BFS from b_1 or its representative. If e is an edge between two neighbor clusters L_1 and L_2, the corresponding "original" green edge is the label of e as given by the high-level data structure.

4.4 Old shared vertices

To cover a candidate that is an old shared vertex a, we would like to build the same graph $G(a)$ as for a non-shared vertex. However, an old shared vertex can be shared by all clusters and, thus, $G(a)$ can consist of m edges. Thus we cannot afford to construct $G(a) \setminus a$ whenever we have to cover a, but we have to maintain dynamically a data structure that represents $G(a) \setminus a$ instead.

Let us call a vertex not belonging to any cluster $C(a)$ an *outside* vertex of a. If an outside vertex is adjacent to a vertex in a cluster $C(a)$, it is called an *outside neighbor* of $C(a)$.

In [12] we maintain a graph $G(a)$ constructed as follows: (1) Remove from $G \setminus a$ edges between some outside vertices and (2) add *artificial* edges between some outside neighbors such that the following conditions hold:

(a) Every artificial edge connects two vertices that are connected in $G \setminus a$.

(b) All outside neighbors that belong to the same cluster are connected in $G(a)$.

By condition 3 of a restricted partition it follows that all outside neighbors belonging to the same node of H_2 are connected in $G(a)$. For each node of H_2 that does not contain a but contains an outside neighbor of $C(a)$, call one of its outside neighbors a *special* neighbor.

The *augmented graph* $A(a)$ is built by "combining" $G(a)$ and $H_2 \setminus C(a)$: $A(a)$ consists of $G(a)$ with the following additional edges. If two nodes C and C' of H_2 both contain a special neighbor of $C(a)$ and there is an edge between C and C' in the minimum spanning forest of $H_2 \setminus C(a)$ then there is an edge (with the same cost) between the special neighbor of C and the special neighbor of C'.

We use the following property of $A(a)$

Lemma 4.1 *Two neighbors of a are connected in $A(a)$ iff they are connected in $G \setminus a$.*

Proof: Every edge in $A(a)$ corresponds to an edge or a path in $G \setminus a$. Thus, if two neighbors are connected in $A(a)$ they are connected in $G \setminus a$.

Consider a path P in $G \setminus a$ between two neighbors b_1 and b_2 of a. Every edge of P that is incident to a vertex of a cluster $C(a)$ also belongs to $G(a)$. Removing these edges from P forms subpaths, each connecting two nodes u_1 and u_2 from the set $\{b_1, b_2,$ outside neighbors of $C(a)\}$. We show that these two nodes are also connected in $A(a)$. The lemma follows.

If u_1 and u_2 belong to the same node of H_2, they are connected in $A(a)$ by the above observation. If u_1 and u_2 belong to different nodes of H_2, these nodes are connected in $H_2 \setminus C(a)$ and therefore the special neighbors of these vertices are connected in $A(a)$. Since both u_1 and u_2 are connected to "their" special neighbors, u_1 and u_2 are connected in $A(a)$. ▮

Now we proceed similar to before. Since a is a candidate, b_1 and b_2 are disconnected in the subgraph of $A(a)$ induced by blue and red edges but are connected in $A(a)$. Thus, there exists a cut in $A(a)$ separating b_1 and b_2 that only contains green edges. We cannot afford to exhaustively search $A(a)$ for such a cut.

However every minimum spanning forest of $A(a)$ (with green edges having cost 1 and all others having cost 0) must cross the cut with a green edge. Thus, it suffices to determine the first green edge on the tree path between b_1 and b_2 in the minimum spanning forest of $A(a)$.

We show next how to implement this algorithm efficiently. The data structure in [12] maintains a spanning forest of $G(a)$. It can be extended to maintain a minimum spanning forest MSF without increase in the asymptotic running time. Additionally MSF can be stored in a dynamic tree data structure [13] without running time increase.

A minimum spanning forest of $H_2 \setminus C(a)$ consists of the minimum spanning forest of $H_2(C(a))$ combined with the (blue) spanning forest T_2 of H_2 where all edges incident to $C(a)$ are removed. This minimum spanning forest of $H_2 \setminus C(a)$ is known since the blue edges of H_2 and the minimum spanning forest of $H_2(C(a))$ are maintained.

The minimum spanning forest of $A(a)$ is a subgraph of the minimum spanning forest of $G(a)$ and the minimum spanning forest of $H_2 \setminus C(a)$.

Thus to cover a first "add" the edges of the minimum spanning forest of $H_2 \setminus C(a)$ to MSF and then determine the first green edge on the path in MSF between b_1 and b_2.

We describe how to add the edges of the minimum spanning forest $H_2 \setminus C(a)$ to MSF. Consider such an edge. If both of its endpoints contain special neighbors, replace it by an edge e between these special neighbors. Otherwise stop. If e's endpoints are not connected in the current MSF, add it to the MSF. Otherwise, determine the maximum cost edge e' on the path in MSF connecting its endpoints. If e's cost is smaller than e''s cost, replace e' with e in the MSF. If its cost is not smaller, do nothing.

Using the dynamic tree data structure of MSF adding an edge takes time $O(\log n)$, for a total of $O(m/k\ .\log n)$. The data structure in [12] requires time $O(k \log n +$

$\sqrt{m}\log n)$ to maintain MSF.

In summary, the additional data structure needed is

- the MSF of $G(a)$, stored in a dynamic tree data structure, and
- a special neighbor for each node of H_2 that contains a neighbor of $C(a)$.

4.5 Complexity and Correctness

We presented a data structure determines in time $O(k\log n + m/k\ .\log n)$ an edge that covers an old shared candidate. It can be updated in amortized time $O(k\log n + \sqrt{m}\log n)$ after each update operation in G and it can be built in time $O(m)$. By choosing $k = \sqrt{m}$, we obtain the following theorem.

Theorem 4.2 *The given data structure determines in time $O(\sqrt{m}\log n)$ an edge that covers a candidate. It can be updated in amortized time $O(\sqrt{m}\log n)$ after each update operation in G and it can be built in time $O(m)$.*

5 Conclusion

We have presented certificates for biconnectivity together with algorithms for updating these certificates. We thus obtained fully-dynamic algorithms for biconnectivity in graphs that run in $O(\sqrt{n}\log n\log\frac{m}{n})$ amortized time per operation. We used novel techniques and approaches to handle sparsification.

We conjecture that our liberalization of certificates and their maintenance, and the usage of monotone subproblems, are important for other dynamization problems, like k-vertex-connectivity for $k \geq 3$.

References

[1] J. Cheriyan and R. Thurimella, "Algorithms for parallel k-vertex connectivity and sparse certificates" *Proc. 23rd Annual Symp. on Theory of Computing*, 1991, 391-401.

[2] D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig, "Sparsification - A technique for speeding up dynamic graph algorithms" *Proc. 33nd Annual Symp. on Foundations of Computer Science*, 1992, 60–69.

[3] D. Eppstein, Z. Galil, G. F. Italiano, and T. Spencer. "Separator based sparsification for dynamic planar graph algorithms". *Proc. 25th Annual Symp. on Theory of Computing*, 1993, 208–217.

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

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

[6] Monika R. Henzinger and Han La Poutré. “Certificates and Fast Algorithms for Biconnectivity in Fully-Dynamic Graphs”. *Proc. Third Annual European Symp. on Algorithms ESA’95*, pages 171–184.

[7] M.H. Rauch Henzinger and V. King, “Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation” *Proc. 27th Annual Symp. on Theory of Computing*, 1995, 519-527.

[8] J.A. La Poutré, “Dynamic Graph Algorithms and Data Structures” *Ph.D. Thesis*, Utrecht University, 1991.

[9] J.A. La Poutré, “Alpha-Algorithms for Incremental Planarity Testing” *Proc. 26 Annual Symp. on Theory of Computing*, 1994, 706-715.

[10] J.A. La Poutré and J. Westbrook, “Dynamic Two-Connectivity with Backtracking” *Proc. 5th Annual ACM-SIAM Symp. on Discrete Algorithms*, 1994, 204-212.

[11] M. H. Rauch. “Fully Dynamic Biconnectivity in Graphs” *Algorithmica* 13, 1995, 503–538. Also appeared in *Proc. 33nd Annual Symp. on Foundations of Computer Science*, 1992, 50–59.

[12] M. H. Rauch. “Improved Data Structures for Fully Dynamic Biconnectivity” *Proc. 26 Annual Symp. on Theory of Computing*, 1994, 686–695. A full version of the paper is available at http:\\www.research.digital.com\SRC\personal\monika\papers.html.

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

[14] J. Westbrook, R. E. Tarjan, “Maintaining bridge-connected and biconnected components on-line” *Algorithmica* 7 (1992), 433–464.