Improved Data Structures for Fully Dynamic Biconnectivity

Monika Rauch*

TR 94-1412
March 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: mhr@cs.cornell.edu.
Improved Data Structures for Fully Dynamic Biconnectivity

Monika Rauch *

Abstract

We present fully dynamic algorithms for maintaining the biconnected components in general and plane graphs.

A fully dynamic algorithm maintains a graph during a sequence of insertions and deletions of edges or isolated vertices. Let \( m \) be the number of edges and \( n \) be the number of vertices in a graph. The time per operation of the best known algorithms are \( O(\sqrt{n}) \) in general graphs and \( O(\log n) \) in plane graphs for fully dynamic connectivity and \( O(\min\{m^{2/3}, n\}) \) in general graphs and \( O(\sqrt{n}) \) in plane graphs for fully dynamic biconnectivity. We improve the later running times to \( O(\min\{\sqrt{n}\log n, n\}) \) in general graphs and \( O(\log^2 n) \) in plane graphs. Our algorithm for general graphs can also find the biconnected components of all vertices in time \( O(n) \). The update times in general graphs are amortized. This shows that the biconnected components of a graph can be dynamically maintained almost as efficiently as the connected components.

1 Introduction

Many computing activities require the recomputation of a solution after a small modification of the input data. Thus algorithms are needed that update an old solution in response to a change in the problem instance. Dynamic graph algorithms are algorithms that allow the change of an input graph by the insertion or deletion of either an edge or an isolated vertex. These operations are called updates. A query operation tests whether two vertices of the graph fulfill a specific condition (i.e., are connected).

We say that a vertex \( z \) is an articulation point separating vertex \( u \) and vertex \( v \) if the removal of \( z \) disconnects \( u \) and \( v \). Two vertices are biconnected if there is no articulation point separating them. In the same way, an edge \( e \) is a bridge separating vertex \( u \) and vertex \( v \) if the removal of \( e \) disconnects \( u \) and \( v \). Two vertices are 2-edge connected if there is no bridge separating them. A biconnected component or block (resp. 2-edge connected component) of a graph is the set of all vertices that are biconnected (resp. 2-edge connected). Note that biconnectivity implies 2-edge connectivity but not vice versa.

Dynamic biconnectivity algorithms maintain the biconnected components of a graph under a sequence of edge insertions and deletions. They are useful to incrementally update the control flow graph in a compiler [11] and to dynamically update the biconnectivity properties of a network. The main contribution of this paper is to show that the biconnected components in a graph can be dynamically maintained nearly as efficiently as the connected components and the 2-edge connected components. In addition, we prove the first lower bounds for all these problems.

*Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: mhr@cs.cornell.edu.
First (Section 2), we study the dynamic biconnectivity problem for general graphs. Frederickson [4] gave the first dynamic graph algorithm for maintaining a minimum spanning tree and the connected components. His algorithm takes time $O(\sqrt{m})$ per update and $O(1)$ per query operation, where $m$ is the number of edges in the graph. The first dynamic 2-edge connectivity algorithm by Galil and Italiano [8] took time $O(m^{2/3})$ per update and query operation. It was consequently improved to $O(\sqrt{m})$ per update and $O(\log n)$ per query operation [5], where $n$ is the number of vertices in the graph. The sparsification technique of Eppstein et al. [2] improves the running time of an update operation to $O(\sqrt{n})$. The previously best known algorithm for maintaining the biconnected components takes amortized time $O(\min\{m^{2/3},n\})$ per update and $O(1)$ per query operation [12, 2]. As supposed to the case of dynamic connectivity and 2-edge connectivity the sparsification technique does not "automatically" speed up fully dynamic biconnectivity algorithms.

Our new approach is instead of applying sparsification "on top of" a dynamic algorithm, we use it as part of the data structure of a dynamic algorithm. We split the graph into subgraphs for which we build "lazy" data structures. Only an amortized constant number of these data structures have to be updated after an update in the graph, but when a data structure has to be updated, many of its edges change. These updates can be executed fast using a special case of sparsification. Note that to maintain connectivity information and to allow the split of a vertex into $O(1)$ vertices requires time $O(\sqrt{m})$ (using [4]) and a split into many vertices requires time $O(m+n)$. We extend sparsification to allow the split of a vertex into $O(1)$ vertices in time $O(\sqrt{n}\log n)$ and the split into many vertices in time $O(n\log n)$ if the graph $G = (V_1 \cup V_2, E)$ is bipartite and only vertices of $V_1$ are split. We also extend the amortization lemma of [12] to show that only an amortized constant number of data structures has to be rebuilt.

The amortized running time per update operation of the new algorithm is $O(\sqrt{m}\log n)$ and the worst case query time is $O(1)$. Note that this algorithm combined with sparsification [2] provides also a fully dynamic 2-edge connectivity algorithm with $O(1)$ query and $O(\sqrt{n}\log n)$ update time. The best known algorithm uses $O(\sqrt{n})$ update time, but $O(\log n)$ query time.

We also provide an additional operation, called a complete block query. It finds the biconnected components for all vertices in time $O(n)$. Our dynamic biconnectivity algorithm needs linear space and preprocessing time.

Second (Section 3), we study the dynamic biconnectivity problem for plane graphs. The best known dynamic algorithms in plane graphs take time $O(\log n)$ per operation for maintaining connected components by Eppstein et al. [1], $O(\log^2 n)$ for maintaining 2-edge connected components by Hershberger et al. [10, 3], and $O(\sqrt{n})$ for maintaining biconnected components by Eppstein et al. [3]. We present an algorithm that maintains biconnected components in time $O(\log^2 n)$ per operation in plane graphs. We use a topology tree approach based on [4].

An earlier version of this paper appeared in [13].

2 General graphs

Let $G$ be an undirected graph with $n$ vertices and $m$ edges. We assume in the paper that $G$ is connected. If $G$ is not connected, we build the data structure described below for each connected
component. This invariant can be maintained in time $O(\sqrt{n} \log n)$ per update operation. We map $G$ to a degree-3 graph $G'$ by replacing a vertex $z$ of degree $d \geq 4$ by $d$ new vertices $x_1, \ldots, x_d$ and connecting $x_i$ and $x_{i+1}$ by a dashed edge. Every edge $(x, y)$ is replaced by a solid edge $(x_i, y_j)$, where $i$ and $j$ are the appropriate indices of the edge in the adjacency lists for $x$ and $y$. We say that the edge $(x_i, x_{i+1})$ belongs to $z$ and that every $x_i$ is a representative of $z$. To contract an edge $(x, y)$, we remove $(x, y)$ and identify $x$ with $y$. The number of vertices and edges of $G'$ is $O(m + n)$.

Let $T$ be a spanning tree of $G'$. We maintain $T$ dynamically in a fully dynamic connectivity data structure [4, 2] in time $O(\sqrt{n})$ per operation.

Next we decompose $G'$ similar to [5]. A cluster is a set of vertices that induces a subgraph of $T$ that is connected. 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 vertex cluster which is incident to $\leq 3$ tree edges and has cardinality at most $k$.
2. Each cluster which is incident to 3 tree edges is of cardinality $1$.
3. All dashed edges incident to a cluster belong 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 at most $k$ and can be found in time $O(m + n)$ [5]. The cluster containing a representative of a vertex $z$ is denoted by $C(z)$ and we say that $C(z)$ contains $z$. If the representatives of $x$ are contained in more than one cluster, then $x$ is called a shared vertex and a cluster that contains a representative is called a cluster sharing $x$ or $x$-cluster.

By condition 3 every cluster shares $\leq 1$ vertex. Since each dashed edge between two clusters is a tree edge, there are $O(m/k)$ shared vertices. Let $\pi_{G'}(u, v)$ be the spanning tree path from $u$ to $v$ in $G'$. The biconnectivity properties of $G$ and $G'$ are related in the following way [12].

Lemma 2.1 Let $G''$ be the graph that results from $G'$ by contracting all dashed edges of every vertex on $\pi_{G'}(u, v)$. Two vertices $u$ and $v$ are biconnected in $G$ if and only if they are biconnected in $G''$.

Thus to answer a query $(u, v)$ operation we have to contract all dashed edges of every vertex on $\pi_{G'}(u, v)$. For this purpose we maintain three types of data structures. For each cluster we keep an internal graph (see Section 2.3) and for each shared vertex we keep shared graphs (see Section 2.4). Finally, we store two graphs of clusters, called high-level graphs (see Section 2.1).

After the insertion or deletion of an edge $(u, v)$ in $G$ (also called an update in $G$) we rebuild the internal graphs of the (at most two) clusters that contain $u$ and $v$ and we update the shared graphs of the at most four shared vertices whose shared graphs contain $u$ and $v$. This requires time $O((k + m/k) \log n)$. Additionally, we maintain the restricted partition of order $k$. If a tree edge between two clusters is deleted and another edge becomes a tree edge, then three tree edges can be incident to a cluster of cardinality larger than 1. Thus these clusters have to be split, each into up to five clusters. When the restricted partition is restored, we update the high-level graphs by applying the algorithm to it recursively.

Since an insertion can increase the number of vertices in a cluster and a deletion can increase the number of clusters, we completely rebuild all data structures every $\min(k, m/k)$ updates to
guarantee that there are $O(m/k)$ cluster, each with $O(k)$ edges. This is called complete rebuild and takes time $O(m)$ and adds an amortized cost of $O(k + m/k)$ to every update operation.

Let $B(n)$ be the time of a complete block query in a graph with $n$ vertices, let $Q(m)$ be the query time and $T(m)$ be the update time in a graph with $m$ edges. We will show that

$$B(n) = O(n), \ Q(m) = O(1), \ \text{and} \ T(m) = 26T((m/k)^2) + 2B(m/k) + O((k + m/k)\log n),$$

since we apply the algorithm recursively to the two high-level graphs with $O(m/k)$ vertices and $O((m/k)^2)$ edges. Choosing $k = 6\sqrt{m}$ gives the solution $T(m) = O(\sqrt{m}\log n)$.

### 2.1 The high-level graphs

There are two high-level graphs, $H_1$ and $H_2$. The graph $H_1$ contains a vertex for each cluster of $G'$, Two vertices $C$ and $C'$ of $H_1$ are connected by an edge (resp. tree or dashed edge) if and only if there is an edge (resp. tree or dashed edge) between a vertex of $C$ and a vertex of $C'$. The graph $H_2$ is created from $H_1$ by contracting all dashed edges of $H_1$. Thus a vertex of $H_1$ corresponds to one cluster and a vertex of $H_2$ corresponds to at least one cluster. We call the vertex corresponding to a cluster $C$ in both graphs $C$. Whenever an edge deletion splits a vertex of $H_2$, it also splits a vertex of $H_1$, but not vice versa. Each edge of $H_1$ is called a $H_1$-bundle, each edge of $H_2$ is called a $H_2$-bundle. Note that each $H_2$-bundle represents at least one $H_1$-bundle. We say a bundle represents all the edges between the two clusters that are its endpoints.

We apply the algorithm recursively to maintain the blocks of $H_1$ and $H_2$. An update in $G$ can create an edge insertion or deletion or a vertex split in $H_1$. A split of a vertex of degree $\leq 3$ requires $\leq 2$ edge updates. To split a vertex $C$ of degree $> 3$, we “cut” the chain of dashed edges belonging to $C$ into pieces and reconnect them appropriately. If the solid edges incident to the dashed chain are ordered in the order in which they are encountered by an Eulerian tour traversal of the (bidirected) spanning tree of $C$, a split of a cluster into $i$ clusters corresponds to at most $i$ deletions and at most $1$ insertion of dashed edges in $H_1$. This order can be maintained with at most four edge updates in $H_1$ whenever a tree edge inside a cluster is deleted, but the cluster is not split. After an update in $G$ up to two clusters can be split, each into up to five clusters. Together with inserting and deleting the updated edge, an update in $G$ can cause up to 13 edge insertions and deletions in $H_1$. Updating $H_2$ can be done symmetrically. This shows the following lemma.

**Lemma 2.2** A biconnectivity query in a high-level graph takes time $Q((m/k)^2)$. A complete block query takes time $B(m/k)$. Updating a high-level graph after an update in $G$ takes time $13T((m/k)^2)$.

The spanning tree of $H_1$ determines the clusters involved in a query as follows [12].

**Lemma 2.3** Let $u'$ and $v'$ be the representatives of $u$ and $v$ in $G'$ such that $C_0 = C(u')$, ..., $C_g = C(v')$ is the sequence of clusters on $\pi_{H_1}(C(u'), C(v'))$ and no representative of $u$ is in $C_1$ and no representative of $v$ is in $C_2$. Then $u$ and $v$ are biconnected in $G$ if
1. no vertex of $C_i$ separates the vertices of $C_{i-1}$ and of $C_{i+1}$ in $G$ for $1 \leq i < g$,
2. $u'$ is not separated from the vertices in $C_1$ by an articulation point in $C_0$,
3. $v'$ is not separated from the vertices in $C_{g-1}$ by an articulation point in $C_g$. and
4. no shared vertex on $\pi_G(u, v)$ separates $u$ and $v$.

Internal graphs (see Section 2.3) are used to test conditions 1 to 3, shared graphs (see Section 2.4) are used to test condition 4.

2.2 The testing data structure

In this section we describe a data structure that is maintained for $H_1$ and $H_2$ to decide which internal graphs and which shared graphs have to be rebuilt after the deletion of the edge $(u, v)$. An internal (resp. shared) graph is a data structure for a vertex $C$ of $H_1$ (resp. $H_2$) containing a vertex for each neighboring cluster of $C$. If one of the neighboring clusters $D$ of $C$ is split into two clusters $D'$ and $D''$ and both, $D'$ and $D''$ are adjacent to $C$, the data structure of $C$ has to be updated except if $D'$ and $D''$ are biconnected in $H_1$ (resp. $H_2$). Updating the data structure of $C$ to reflect all splits of its neighbor $D$ is called rebuilding $C$. Thus we rebuild $C$ if $C$ and $D$ fulfill some special condition or if $D'$ and $D''$ are not biconnected right after the split. Otherwise we delay the rebuild until some later deletion causes $D'$ and $D''$ no longer to be biconnected. We call this a delayed rebuild. Note that after a deletion more than a constant number of delayed rebuilds can occur. In this section we present a data structure to detect delayed rebuilds and we prove that during $l$ update operations only $O(l)$ delayed rebuilds occur. Thus each update operation causes only an amortized constant number of delayed rebuilds.

2.2.1 Description

Given a high-level graph $H$ (where $H$ can be either $H_1$ or $H_2$) we present a data structure that determines in time $B(m/k) + O(m/k)$ all clusters $C$ for which a delayed rebuild has to be executed because of the deletion of the edge $(u, v)$. Note that no test is necessary after an insertion, since an insertion does not create articulation points in $H$.

We call a cluster $A$ an ancestor of a cluster $C$, if the vertices of $C$ belonged to $A$ after the last complete rebuild. It is easy to maintain at each cluster its ancestor. Whenever $D$ is split into $D'$ and $D''$, we rebuild the data structure of $C$ if $C$ and $D$ have the same ancestor or if $D'$ and $D''$ are not biconnected in $H$. Otherwise, we delay rebuilding $C$ until the first deletion that articulates $D'$ and $D''$ in $H$. (Note that is is possible that $D'$ and $D''$ are represented by the same vertex of $H_2$ in which case they will never be articulated in $H_2$.)

We mark $H$-bundles at each endpoint as follows: Whenever a cluster $C$ is rebuilt, all $H$-bundles incident to $C$ are unmarked at $C$. Every newly inserted $H$-bundle in unmarked. When an unmarked $H$-bundle $(C, C')$ is split because $C'$ is split, we store this original bundle and mark the two new $H$-bundles at $C$ with pointers to the original bundle. When a marked $H$-bundle $(C, C')$ is split because $C'$ is split, we mark the two new $H$-bundles at $C$ with pointers to the original bundle to which $(C, C')$ pointed. Thus each marked $H$-bundle points to the original bundle that represented its edges (and additional edges). This data structure is only affected by splits and rebuilds of clusters. As we will, a constant number of cluster is split and an amortized constant number of clusters is rebuilt after each update operation in $G$. Each split or rebuild of a cluster requires updating the marks of all bundles incident to it and takes thus $O(m/k)$ time. This shows that the
testing data structure can be updated in $O(m/k)$ amortized time after each update operation in $G$.

The only candidates for delayed rebuilds are clusters that become articulation points by the deletion of $(u, v)$. Thus they have to lie on $\pi_H(C(u), C(v))$ in the updated high-level graph $H$. Let $P = \pi_H(C(u), C(v))$. We test for every cluster $C$ on $P$ whether the two $H$-bundles incident to $C$ on $P$ point to the same original bundle. If no, $C$ does not have to be rebuilt. If yes, we test whether the other endpoints of the two bundles are biconnected in $H$. This can be done in constant time after executing one complete block query. If they are biconnected, $C$ does not have to be updated. If they are not biconnected, $C$ has to be updated and all $H$-bundles adjacent to $C$ are unmarked.

Since there are $O(m/k)$ clusters on $P$ and each test whether two clusters are biconnected takes constant time, the whole process takes time $O(m/k)$. This shows the following lemma.

Lemma 2.4 The testing data structure determines in $B(m/k) + O(m/k)$ time which delayed rebuilds have to be executed after each update operation. It can be updated in $O(m/k)$ amortized time after each update operation.

2.2.2 The amortization lemma

We show in this section that if $l$ update operations have been executed since the last complete rebuild the total number of delayed rebuilds is $O(l)$. This lemma applies to any high-level graph and is thus an extension of the lemma shown in [12].

Lemma 2.5 Let $H$ be a high-level graph and let $l$ be the number of update operations since the last complete rebuild. Then at most $4l$ delayed rebuilds have occurred since the last complete rebuild.

Proof: Consider the following bipartite graph $K$. It contains a blue node for every cluster that was created by a cluster split after the last complete rebuild and has not been split since. It contains a red node $(C, b)$ for every original $H$-bundle $b$ at every cluster $C$ with whom a bundle incident to $C$ is marked (i.e. that has been split since the last complete rebuild). Let $D_1, \ldots, D_j$ be the other endpoints for every bundle that is marked at $C$ with $b$. Then there is an edge between $(C, b)$ and every blue node $D_i$. (Note that each $D_i$ was created by a split and thus is represented by a blue node.) Thus every edge in $K$ corresponds to an $H$-bundle. Whenever a cluster is split or rebuilt, $K$ is adapted accordingly.

We first show the following claim.

Claim: If two blue nodes $C$ and $C'$ are connected in $K$, then $C$ and $C'$ have the same ancestor.

Proof: If there is a path in $K$ between the two blue nodes $C$ and $C'$, then let $B_1, \ldots, B_j$ be the blue nodes on this path with $C = B_1$ and $C' = B_j$. It follows from the construction of $K$ that $B_i$ and $B_{i+1}$ have the same ancestor and by the transitivity of the ancestor relation that $C$ and $C'$ have the same ancestor. ■
Let $C_1, \ldots, C_l$ be the clusters represented by a vertex $C$ in $H$. Whenever a vertex $C$ of $H$ has to rebuild because the edge bundles $(C, D)$ and $(C, D')$ that are marked with the same bundle $b$ do not belong to same block of $H$, $C$ is an articulation point in $H$ separating $D$ and $D'$ and also $C$ and $D$ do not have the same ancestor. If follows from the above claim that the blue nodes of $C$ and $D$ and of $C$ and $D'$ are not connected in $K$. Thus there does not exist a path between $D$ and $D'$ that contains the blue node of $C$. Since $K$ is bipartite, every path in $K$ from $D$ to $D'$ consists of a path from $D$ to a blue node $A$, an edge from $A$ to a red node $(C_{j,,})$ for some $1 \leq j \leq l$, an edge from the red node $(C_{j,,})$ to a blue node $B$ and a path from $B$ to $D'$. The above claim shows that $A$, $B$, $D$, and $D'$ have the same ancestor. Updating $C$ will remove the edge between $A$ and any red node $(C_{j,,})$ and between $B$ and any red node $(C_{j,,})$. Thus updating $C$ disconnects $D$ and $D'$ and thus the number of connected components of $K$ is increased by one.

We assume first that there are no insertions of edges in $G'$. Then two connected components in $K$ that are created during a split cannot become connected again. For any $l' \leq l$, there are at most $l'$ splits of clusters during $l''$ deletions, each split increases the number of clusters by at most four. Therefore, there are at most $4l''$ blue vertices in $K$. Every red vertex is connected to a blue vertex. Thus there can be at most $4l'$ connected components in $K$. As we showed above, each delayed rebuild increases the number of connected components by one. Thus there are at most $4l''$ delayed rebuilds during $l'$ deletions.

Now consider also insertions of edges. Because of insertions two connected components of $K$ that were disconnected by a split can become connected again during a later split. Note, however, that every insertion can connect at most two connected components of $K$ and thus decrease the number of connected components by at most 1. There are $l - l'$ insertions and thus at most $4l'' + (l - l') \leq 4l$ delayed rebuilds.

### 2.3 The internal graphs

Let $V(C)$ be the set of vertices in $G$ whose representative is contained in $C$ or connected to $C$ by a tree edge. We build an internal graph $I(C)$ for each cluster $C$ to answer a biconnectivity query between two vertices of $V(C)$ in time $O(1)$. If $C$ has tree degree 3, $V(C)$ consists of at most four vertices, all in different clusters. Thus a biconnectivity query in $V(C)$ can be answered with one biconnectivity query in $H_1$. Hence we assume in the rest of this section that the tree degree of $C$ is 1 or 2.

#### 2.3.1 Description

After each complete rebuild, $I(C)$ consists of a vertex for every cluster that is adjacent to $C$ in $H_1$ (called c-vertex) and for every vertex contained in $C$. The graph contains all the edges between vertices of $C$. An edge between a vertex $u \in C$ and a vertex $v \notin C$ is represented by the edge $(u, C(v))$. All dashed edges are contracted except for edges incident to c-vertices. After some updates in $G$, a c-vertex represents $\geq 1$ clusters, but only if these clusters are biconnected in $H_1$. The c-vertices are connected by artificial edges such that
1. if there are \( l \) c-vertices, then they are connected by \( \leq l - 1 \) artificial edges and

2. so that two c-vertices are connected by artificial edges if and only if the clusters represented by them are biconnected in \( H_1 \).

In [12] we describe a data structure that maintains the graph \( I(C) \) so that

- building the data structure for \( I(C) \) takes time \( O(m/k + k) \),
- changing the artificial edges takes time \( O(l) \), where \( l \) is the number of c-vertices,
- determining the block of \( I(C) \) for all tree edges of \( G \) in \( C \) or adjacent to \( C \) takes time proportional to the number of those tree edges, and
- testing whether two vertices of \( V(C) \) are biconnected in \( I(C) \) takes constant time.

Given these properties of the internal graph we show in the following subsections that all internal graphs can be updated in amortized time \( O(m/k + k) \) after an update in \( G \).

### 2.3.2 Insertion

After the insertion of the edge \((u, v)\) we rebuild \( I(C(u)) \) and \( I(C(v)) \). The only other internal graphs that change are the ones whose clusters lie on \( \pi_{H_1}(C(u), C(v)) \) and were articulation points separating \( C(u) \) and \( C(v) \) before the insertion. Since these clusters are no longer articulation points, the artificial edges have to be updated.

To find all these clusters we find all articulation points separating \( C(u) \) and \( C(v) \) using a complete block query in \( H_1 \) before the insertion. Then we determine all clusters on \( \pi_{H_1}(C(u), C(v)) \) such that the blocks of the two adjacent tree edges on \( \pi_{H_1}(C(u), C(v)) \) are not identical. These are exactly the clusters whose artificial edges have to be updated. We first remove all the artificial edges of these internal graphs and then we add new ones. Since these clusters have tree degree 2 in \( H_1 \) and are no longer articulation points, the new artificial edges consist of one chain connecting all c-vertices. This guarantees that properties (1) and (2) are fulfilled in all internal graphs.

Note that the degree of these clusters in \( H_1 \) sums to \( O(m/k) \) since all such clusters are articulation points lying on a path in \( H_1 \). Since the number of artificial edges in each internal graph is bounded by the degree of the cluster in \( H_1 \) and updating the artificial edges of an internal graph takes time linear in their number, all artificial edges can be updated in time \( O(m/k) \) plus the time for one complete block query in \( H_1 \).

### 2.3.3 Deletion of a non-tree edge

The deletion of a non-tree edge \((u, v)\) is basically the inversion of an insertion. We rebuild \( I(C(u)) \) and \( I(C(v)) \). No other internal graphs change except for the internal graphs of clusters that separate \( C(u) \) and \( C(v) \) in the new high-level graph. In these internal graphs only the artificial edges have to be updated. Determining these clusters, removing the artificial edges, and adding the new artificial edges can be done in time \( O(m/k) \) as described for insertions.

However, we also have to compute the new artificial edges. Thus we number the vertices of \( H_1 \) with a prefix and postfix number during a depth-first traversal of the spanning tree of \( H_1 \) starting at a leaf of \( H_1 \). Let \( l \) be the number of c-vertices in the internal graph that we are currently
updating and let $C$ be the cluster represented by the internal graph. We connect every $c$-vertex whose cluster $C'$ is not a tree neighbor of $C$ by an edge to 'its' tree neighbor of $C$, i.e. to the tree neighbor of $C$ that lies on the tree path from $C'$ to $C$. (Since $C'$ and $C$ are connected by an edge, $C'$ and 'its' tree neighbor are biconnected in $H_1$.) In this way we guarantee that there are at most $l - 1$ artificial edges and that two $c$-vertices are connected by artificial edges if and only if the two clusters that are represented by them are biconnected in $H_1$. Using the prefix and postfix number at each cluster we can determine in constant time for each $c$-vertex to which cluster it has to be connected. Thus one artificial edge can be computed in time $O(1)$ and computing all artificial edges takes time $O(m/k)$. This guarantees that property (1) and (2) of artificial edges are fulfilled for all internal graphs.

Additionally, we update the testing data structure of $H_1$ and we execute all required delayed rebuilds.

### 2.3.4 Deletion of a tree edge

If a tree edge $(u, v)$ is deleted, we rebuild $I(C(u))$ and $I(C(v))$. If the deletion does not disconnect $G$, a non-tree edge $(x, y)$ becomes a tree edge. If $C(x) \neq C(y)$, this might require splitting $C(x)$ and $C(y)$ to maintain a restricted partition (since a cluster that is incident to three tree edges has to consist of one vertex). In this case we also have to rebuild $I(C(x))$ and $I(C(y))$. Neither property (1) nor (2) of artificial edges are violated in clusters that do not lie on $\pi_{H_1}(C(u), C(v))$ in the new high-level graph $H_1$ or in clusters that lie on $\pi_{H_1}(C(u), C(v))$ in the new high-level graph and that are not separating $C(u)$ and $C(v)$ in the new high-level graph. But for clusters on $\pi_{H_1}(C(u), C(v))$ that are separating $C(u)$ and $C(v)$ in the new high-level graph the artificial edges have to be updated. This can be done in time $O(m/k)$ for all such clusters as described above.

Additionally, a constant number of clusters might possibly have to be split. If $C(u) = C(v)$, then it is possible that $C(u)$ is now disconnected and has to be split. If $C(u) \neq C(v)$, then possibly $C(x)$ and $C(y)$ have to be split. Whenever a cluster is split into two clusters $C'$ and $C''$, we rebuild $I(C')$ and $I(C'')$ and we also rebuild the internal data structure of all clusters that have the same ancestor as $C$. Since they altogether contain $O(k + m/k)$ edges and vertices, this takes time $O(k + m/k)$. If $C'$ and $C''$ are not biconnected in $H_1$, we rebuild the (one) cluster that is adjacent to both if it exists. If $C'$ and $C''$ are biconnected, we use the testing data structure of $H_1$ (see Section 2.2) to determine in time $B(m/k) + O(m/k)$ after each update operation which clusters require a delayed rebuild. Lemma 2.5 shows that only an amortized constant number of internal data structures have to be rebuilt after each update operation. This adds an amortized cost of $O(k + m/k)$ to each update operation since rebuilding an internal graph takes time $O(k + m/k)$.

Note that only at most one complete block query in $H_1$ has to be executed to update the internal graphs. If $H_1$ is not changed by the update, no such query has to be executed. Otherwise, if an edge is inserted, we execute a complete block query before we add the edge to $H_1$, if an edge is deleted, we execute a complete block query after the edge is deleted from $H_1$. This shows the following lemma.
Lemma 2.6  Let \( C \) be a cluster. A biconnectivity query in \( I(C) \) can be answered in constant time and the blocks in \( I(C) \) of the tree edges of \( G \) can be determined in time linear to the number of such tree edges. The amortized time to update all internal graphs after an edge insertion or deletion in \( G \) is \( B(m/k) + O(k + m/k) \).

2.4 The shared graphs

2.4.1 Description

We call a shared vertex old if it is created during a complete rebuild of the data structure and we call it new otherwise (i.e. if it is created by the split of a cluster). For each shared vertex \( s \) we build shared graphs to test in constant time whether \( s \) separates its two tree neighbors \( x \) and \( y \) of \( G \), i.e. whether \( x \) and \( y \) are disconnected in \( G \setminus s \). Let

- \( V(s) \) denote the set of vertices of \( G \) that are currently contained in a \( s \)-cluster without \( s \) if \( s \) is new and let \( V(s) \) denote this set of vertices after the last complete rebuild if \( s \) is old, let
- \( m(s) \) denote the number of edges incident to vertices in \( V(s) \), let
- \( N(s) \) denote the set of clusters that do not share \( s \), are adjacent to a \( s \)-cluster, and whose vertices are not contained in \( V(s) \), and let
- \( n(s) \) denote the number of clusters in \( N(s) \).

Note that, due to splits of clusters, \( V(s) \) of an old shared vertex \( s \) consists of the vertices of all \( s \)-clusters and of the neighboring clusters that are created by splits of \( s \)-clusters. It follows that every vertex is contained in the shared graphs of at most two shared vertices, namely at most one old and at most one new shared vertex. If \( s \) is new, \( N(s) \) consists of all clusters that do not share \( s \), but are adjacent to a \( s \)-cluster. If \( s \) is old, the clusters of \( N(s) \) have additionally to be vertex-disjoint with \( V(s) \), i.e. a neighbor of a \( s \)-cluster that is created by a split of an \( s \)-cluster is not contained in \( N(s) \).

To maintain \( G \setminus s \) directly is too expensive. Thus we maintain instead one graph \( G_s \) for all shared vertices and 4 “personal” graphs \( G_i(s), 0 \leq i \leq 3 \), for \( s \). The graph \( G_s \) consists of \( G \) with all shared vertices removed. Intuitively, the “personal” graphs \( G_i(s) \) are used to “add” all shared vertices except \( s \) to \( G_s \) and thus to create a compressed representation of \( G \setminus s \). Edges in \( G_s \) between 2 vertices in the same cluster \( C \) or in the same set \( V(s) \) for any shared vertex \( s \) have cost 0, all other edges have cost 1. Thus two vertices of a cluster \( C \) (resp. \( V(s) \)) are connected in the minimum-spanning forest \( F_s \) by a path contained in \( C \) (resp. \( V(s) \)) if such a path exists. We use the data structure in [2] to maintain \( F_s \) dynamically.

The graph \( G_0(s) \)

The graph \( G_0(s) \) is the subgraph of \( G_s \) induced by \( V(S) \). We maintain a restricted partition of order \( k \) with respect to \( F_s \) that splits \( G_0(s) \) into \( O(m(s)/k + cc) \) sets, called subclusters, of \( \leq k \) vertices, where \( cc \) is the number of connected components in \( G_0(s) \). We call a subcluster small if it is not connected to any other subcluster of \( G_0(s) \) and large otherwise. Thus there are \( O(m(s)/k) \) large and \( O(m(s)) \) small subclusters. To assign small subclusters to clusters of \( N(s) \) we call one of the tree edges connecting a small subcluster to a cluster \( C \) of \( N(s) \) in \( F_s \) a designated edge of the subcluster (if it exists) and say the subcluster is designated to \( C \).
Since small subclusters of $G_0(s)$ are not adjacent to large subclusters of $G_0(s)$ in $G_s$, tree edges of $F_s$ connect large subclusters of $G_0(s)$ either with other large subclusters of $G_0(s)$ or with clusters of $N(s)$. Thus there are $O(n(s) + m(s)/k)$ tree edges incident to large subclusters of $G_0(s)$ in $F_s$, since every subcluster of $G_0(s)$ contains a subtree (and not a subforest) of $F_s$.

**The graph $G_1(s)$**

To consider paths from $x$ to $y$ that contain other shared vertices, we build a graph $G_1(s)$ that consists of all vertices in $V(s)$, one vertex for each shared vertex except for $s$, and all edges (with cost 1) between a shared vertex and a vertex of $V(s)$. Additionally, we add the edges of $F_s$ between vertices of $V(s)$ with cost 0. We maintain the minimum spanning forest $F_1(s)$ of $G_1(s)$ in the dynamic data structure of [4]. For every small subcluster of $G_0(s)$ we call one of its tree edges to a shared vertex a designated edge (if it exists). Every shared vertex and all its designated small subclusters form a subtree of $F_1(s)$, called S-tree. Since there are $O(n(s))$ S-trees and every subcluster contains a subtree (and not a subforest) of $F_1(s)$, there are $O(n(s) + m(s)/k)$ tree edges incident to large subclusters or S-trees in $F_1(s)$.

**Lemma 2.7** Let $s$ be a shared vertex. If a small subcluster of $G_0(s)$ is designated to a cluster $C$ of $N(s)$ (resp. to a shared vertex $s'$), then there exists a path without $s$ in $G$ from every vertex in the small subcluster to every vertex in $C$ (resp. to $s'$).

**Proof:** If a small subcluster is designated to $C$ (resp. to $s'$), then there exists an edge from the subcluster to $C$ (resp. to $s'$). Thus every vertex of the small subcluster is connected to every vertex in $C$ (resp. to $s'$) by a path in $G \setminus s$. []

**The graph $G_2(s)$**

The spanning forest $F_s$ (resp. $F_1$) contains the paths between two vertices of $G_s$ (resp. two shared vertices) that "pass through" a small subcluster. To consider paths between a vertex of $G_s$ and a shared vertex that "pass through" a small subcluster we build the bipartite graph $G_2(s)$. After each rebuild $G_2(s)$ contains a vertex for every cluster in $N(s)$ (called c-vertex) and a vertex for every shared vertex to whom a small cluster is designated. After some updates in $G$, a c-vertex of $G_2(s)$ can represent $\geq 1$ cluster and we maintain the following invariants.

**Invariant of $G_2(s)$:**

1. A shared vertex and a c-vertex are connected in $G_2(s)$ by an edge if and only if there is a small subcluster that is designated to the shared vertex and to one of the clusters represented by the c-vertex.

2. All clusters represented by the same c-vertex are biconnected in $H_2$ and they have the same ancestor.

Since there are $O(n(s))$ vertices in $G_2(s)$, the spanning forest $F_2(s)$ consists of $O(n(s))$ edges. Let $c_1, \ldots, c_d$ for $d \geq 1$ be the c-vertices of $G_2(s)$. Wlog assume that $d$ is a power of 2. To maintain $F_2(s)$ we build the following complete binary sparsification tree with $d$ leaves. Leaf $i$ of
the sparsification tree is labeled with the graph consisting of a vertex for every shared vertex in \( G_2(s) \) and one vertex for \( c_i \). There is an edge between a shared vertex and \( c_i \) if and only if there exists a small subcluster designated to the shared vertex and to a cluster represented by \( c_i \). An internal node in the sparsification tree is labeled with the graph that is the union of the spanning forests of the graphs that are stored at the two children of the node. The sparsification tree has height \( O(\log n(s)) \) and the root of the sparsification tree is labeled with a graph whose spanning forest is \( F_2(s) \).

**Lemma 2.8** If the representatives of two vertices \( x \) and \( y \) of \( G \) are connected in \( F_2(s) \), then \( x \) and \( y \) are connected in \( G \setminus s \).

**Proof:** Every edge in \( F_2(s) \) corresponds to a path in \( G \setminus s \) and every vertex in \( F_2(s) \) represents a connected component of \( G \setminus s \). The lemma follows from this observation.

The graph \( G_3(s) \)

To combine \( F_3 \), \( F_1(s) \), and \( F_2(s) \) we build \( G_3(s) \). After each complete rebuild, \( G_3(s) \) contains a vertex for each cluster in \( N(s) \) (called c-vertex) and a vertex for every large subcluster. After some updates in \( G \), a vertex of \( G_3(s) \) can represent more than one cluster of \( N(s) \) and we maintain the following invariant.

**Invariant of \( G_3(s) \):** All clusters represented by a vertex of \( G_3(s) \) are biconnected in \( H_2 \) and have the same ancestor.

Note that c-vertices in \( G_2(s) \) always represent the same clusters as c-vertices in \( G_3(s) \). The graph \( G_3(s) \) contains the edges of \( F_s \) incident to large subcluster, the edges of \( F_1(s) \) incident to S-trees, and the edges of \( F_2(s) \) (a shared vertex is identified with the cluster in \( N(s) \) containing it). In addition, all clusters of \( N(s) \) that are in the same block of \( H_2 \) are connected by a chain of artificial edges. Thus there are \( O(n(s) + m(s)/k) \) vertices and \( O(n(s) + m(s)/k) \) edges in \( G_3(s) \).

Each vertex not in \( V(s) \) is represented in \( G_3(s) \) by its cluster (if it is in \( N(s) \)) or by the cluster in \( N(s) \) to which its cluster is connected by a tree path in \( H_2 \). Each vertex in a large subcluster is represented by the vertex for its subcluster. Each vertex of a small subcluster is represented by its designated cluster of \( N(s) \) if it exists. If the small subcluster is not designated to a cluster of \( N(s) \), but to a shared vertex, the vertex is represented by the vertex representing the shared vertex. If the small subcluster does not have a designated edge, its vertices are not represented in \( G_3(s) \). Let \( F_3(s) \) be the spanning forest of \( G_3(s) \).

**Lemma 2.9** If two vertices \( x \) and \( y \) of \( G_s \) are connected in \( G_s \), then their representatives are connected in \( F_3(s) \) (if they exist).

**Proof:** If all vertices on \( \pi_{F_s}(x,y) \) are in \( V(s) \), then all edges on \( \pi_{F_s}(x,y) \) have cost 0 and thus \( x \) and \( y \) either belong to the same subcluster or to two subclusters that are connected in \( F_1(s) \). In either case their representatives are connected in \( F_3(s) \).

If there exist vertices not in \( V(s) \) on \( P = \pi_{F_s}(x,y) \), then let \( C_1, \ldots, C_t \) be the ordered sequence of clusters in \( N(s) \) on \( P \) (with possibly multiple copies) and \( z_{2i} \) (resp. \( z_{2i+1} \))
be the first (resp. last) vertex on $P$ in $C_i$ for $0 \leq i \leq l$. The vertices $z_{2i}$ and $z_{2i+1}$ are represented by the same vertex in $F_3(s)$. Note that $C(z_j)$ is the representative of $z_j$ in $G_3(s)$ if $z_j$ is adjacent to a vertex of $V(s)$. If $\pi_{F_3}(z_{2i-1}, z_{2i})$ does not contain a vertex of $V(s)$, then $\pi_{F_3}(z_{2i-1}, z_{2i})$ consists of one edge between two clusters not sharing $s$. Thus the representative of $z_{2i-1}$ and of $z_{2i}$ are either identical or connected by artificial edges in $F_3(s)$.

If $\pi_{F_3}(z_{2i-1}, z_{2i})$ contains vertices of a small subcluster, then all vertices on $\pi_{F_3}(z_{2i-1}, z_{2i})$ (excluding $z_{2i-1}$ and $z_{2i}$) belong to the same small subcluster. Thus the S-trees of $C(z_{2i-1})$ and of $C(z_{2i})$ are connected in $F_1(s)$ and thus $C(z_{2i-1})$ and $C(z_{2i})$ are connected in $F_3(s)$.

If $\pi_{F_3}(z_{2i-1}, z_{2i})$ does not contain vertices of small subclusters, all vertices belong to large subclusters. Since these subclusters are connected by a path that only contains vertices of $V(s)$, these subclusters are connected in $F_1(s)$. Thus $C(z_{2i-1})$ and $C(z_{2i})$ are connected in $F_3(s)$.

If the subcluster of $x$ is large, it is connected to $C(z_1)$ in $F_s$ and thus in $F_3(s)$, since it is connected by an edge in $F_1(s)$. If the subcluster of $x$ is small, it must be represented by a cluster $C$ of $N(s)$. The clusters $C$ and $C(z_1)$ are connected in $F_s$ (since they are both connected to the small subcluster of $x$) and thus they are connected in $F_3(s)$. The same argument holds for $y$ and $F(z_{2i+1})$. Thus the representative of $x$ and of $y$ are connected in $F_3(s)$.

Lemma 2.10 Let $s$ be a shared vertex and let $x$ and $y$ be two tree neighbors of $s$ in $G$. Then $s$ does not separate $x$ and $y$ in $G$ iff $x$ and $y$ are connected in $G_s$ or the vertices that represent $x$ and $y$ in $G_3(s)$ are connected in $G_3(s)$.

Proof: If $s$ separates $x$ and $y$, then $x$ and $y$ are not connected in $G_s$. Assume that there exists a path in $F_3(s)$ between the representatives of $x$ and of $y$. If an edge used by this path belongs to $F_s$ or $F_1(s)$, it is an edge of $G \setminus s$. If it belongs to $F_3(s)$, it corresponds to a path in $G \setminus s$ according to Lemma 2.8. Every vertex in $G_3(s)$ corresponds to a path of $G \setminus s$ and, according to Lemma 2.7, each vertex in a small subcluster is connected to its representative by a path of $G \setminus s$. Thus the path in $F_3(s)$ from the representative of $x$ to the representative of $y$ corresponds to a path in $G \setminus s$ connecting $x$ and $y$. This contradicts the assumption that $s$ separates $x$ and $y$ and hence, the representative of $x$ and of $y$ are not connected in $G_3(s)$.

If $s$ does not separate $x$ and $y$, then there exists a path $P$ in $G$ from $x$ to $y$ that does not use $s$. If $P$ does not contain any shared vertex, then $x$ and $y$ are connected in $G_s$. Otherwise, let $s_1, \ldots, s_l$ be the shared vertices in the order that they occur on $P$ and let $x_{2i-1}$ (resp. $x_{2i}$) be the vertex adjacent to $s_i$ and before (resp. after) $s_i$ on the path for $0 \leq i \leq l$. Define $x_0 = x$ and $x_{2i+1} = y$ if $x$ and $y$ are no shared vertices. It follows that $x_{2i}$ and $x_{2i+1}$ are connected in $F_s$ and, by Lemma 2.9, that the representatives of $x_{2i}$ and of $x_{2i+1}$ are connected in $F_3(s)$. In the rest of the proof we shor that the representatives of $x_{2i-1}$ and of
Let $S_i$ be the representative of $s_i$ in $G_3(s)$ and let $j$ be either $2i - 1$ or $2i$. We consider three cases, namely that $x_j$ lies in a large subcluster, that $x_j$ lies in a small subcluster, and that $x_j$ does not lie in $V(s)$. If $x_j$ lies in a large subcluster of $G_3(s)$, then this subcluster and $S_i$ are connected in $F_1(s)$ and thus in $F_3(s)$.

If $x_j$ lies in a small subcluster, then $x_j$ has been designated either to only a shared vertex or to a shared vertex and to a cluster of $N(s)$. Let us first consider the case that $x_j$ has been designated only to a shared vertex. If this vertex is $s_i$, then the representative of $x_j$ is $S_i$. If the designated shared vertex is a vertex $s' \neq s_i$, then $s_i$ and $s'$ are connected in $F_1(s)$ (since they are both adjacent to $x_j$) and thus their representatives are connected in $F_3(s)$. In either case, the representative of $x_j$ and $S_i$ are connected in $F_3(s)$. Now let us consider the case that $x_j$ is designated to a cluster $C$ of $N(s)$ and to a shared vertex. If this shared vertex is $s_i$, then $C$ and $s_i$ are connected in $F_2(s)$ and thus in $F_3(s)$. If the designated shared vertex is $s' \neq s_i$, then $C$ and $s'$ are connected in $F_2(s)$ and thus in $F_3(s)$. Additionally $s_i$ and $s'$ are connected in $F_1(s)$ (since they are both connected to $x_j$). Thus their representatives are connected in $F_3(s)$ and it follows that $C$ and $S_i$ are connected in $F_3(s)$.

If $x_j$ is not in $V(s)$, then either $x_j$ is represented by $S_i$ or by another cluster $C$. Since $S_i$ and $C$ are connected by an edge in $H_2$ and both are adjacent to $C(s)$ in $H_2$, $S_i$ and $C$ are biconnected in $H_2$. Thus they are connected by artificial edges in $G_3(s)$. biconnected in $H_2$, the representative of $x_j$ in $F_3(s)$ is connected to $S_i$ in $F_3(s)$.

This shows that the representatives of $x_j$ and $S_i$ are connected in $F_3(s)$.

The vertex of $G_3(s)$ representing a vertex $x$ can be determined in $O(1)$ time. Thus we can answer the query whether a shared vertex $s$ separates $x$ and $y$ in $O(1)$ time.

### 2.4.2 Updates

**Updating $G_s$, $G_0$, and $G_1$**

An update in $G$ causes at most one insertion or deletion in $G_s$. Thus $G_s$ can be updated in time $O(\sqrt{n}) = O(k + m/k)$ and $F_s$ changes by at most two edges. Since each edge is contained in at most one graph $G_0$ and in at most one graph $G_1$, each update in $G$ requires at most one insertion or deletion in one graph $G_0$ and on graph $G_1$. This takes time $O(\sqrt{n}) = O(k + m/k)$.

If a cluster is split, a new shared vertex $s$ can be created or the set $V(s)$ of a new shared vertex can change. Note that $V(s)$ of old shared vertices is not affected. In the first case, we build the shared graphs for $s$. Since $m(s) + n(s) = O(k)$, all graphs $G_i(s)$ have size $O(k)$. Thus $G_0(s)$ and $G_1(s)$ can be built in time $O(k)$. In the second case we rebuild all shared graphs of the new shared vertex $s$. This takes time $O(k)$, since $m(s) + n(s) = O(k)$.

**Updating $G_2$**
The graph $G_2$ contains $c$-vertices, each representing $\geq 1$ biconnected clusters of $N(s)$, and a vertex for each shared vertex to whom a small subcluster is designated. A $c$-vertex is connected by an edge to a shared vertex if there exists a small subcluster that is designated to a cluster represented by the $c$-vertex and the shared vertex. An update in $G$ can affect the graphs $G_2$ in two ways: The designated edges of small subclusters can change and the clusters represented by a $c$-vertex are no longer biconnected.

Let us first discuss the changes in designated edges. Since each vertex is contained in the shared graphs of at most two shared vertices, each edge can be the designated edge of at most two small subclusters and can be contained in at most two small subclusters. If it is a designated edge of a small subcluster of a shared vertex $s$, then its insertion or deletion requires the insertion or deletion of at most one edge in $G_2(s)$. If the endpoints of the edge are both contained in a small subcluster of a shared vertex $s$, then its insertion or deletion can join two small subclusters or split one small subcluster. In both cases at most two insertions and at most two deletions of edges in $G_2(s)$ are necessary. Thus changes in designated edges require only a constant number of edge insertions and deletions in at most two graphs $G_2$. To update a graph $G_2(s)$ we rebuild every graph on the path in the sparsification tree from the leaf that corresponds to the affected $c$-vertex to the root. This takes time $O(n(s) \log(n(s)))$.

Now we discuss how to update the graphs $G_2$ if the clusters represented by a $c$-vertex are no longer biconnected. This situation can occur if either an edge connecting two clusters is deleted or if the deletion of an edge in a cluster $C$ splits $C$ into two clusters that are not biconnected in $H_2$. Whenever a cluster $C$ of $H_2$ is split, we rebuild the shared graphs of clusters that have the same ancestor as $C$ in $H_2$. Thus two clusters are represented by the same $c$-vertex of $G_2(s)$ only if the clusters and $C(s)$ in $H_2$ do not have the same ancestor in $H_2$. Using the testing data structure of $H_2$ we can determine after each update in $G$, the vertices of $H_2$ and thus the shared vertices whose data structure has to be updated. Lemma 2.5 shows that only an amortized constant number of graphs $G_2$ have to be updated after an update in $G$.

To update $G_2$, we remove the leaf labeled with $c$-vertex and its path to the root from the sparsification tree. Then we build a second sparsification tree for the clusters represented by the $c$-vertex and we merge the two sparsification tree. The details are as follows: We construct a graph for each cluster represented by the $c$-vertex. The graph contains the cluster and one vertex for each shared vertex that is designated to the same small subcluster as the cluster. These graphs become the labels of the leaves of the second sparsification tree. Note that all the vertices of clusters represented by a $c$-vertex have the same ancestor. Thus there are $O(k)$ edge adjacent to all such clusters and, hence, the number of designated edges adjacent to these clusters is $O(k)$. This shows that the graphs labeling the leaves can be built in time $O(k)$. Since the sparsification tree has depth $O(\log(n(s)))$, it takes time $O(k \log(n(s)))$ to build the second sparsification tree. Joining two sparsification trees takes time $O(n(s) \log(n(s)))$. This shows that the amortized time of updating all graphs $G_2$ is $O((k + n(s)) \log(n(s)))$.

**Updating $G_3$**

Each graph $G_3(s)$ contains one vertex for each large subcluster of $G_0(s)$ and special vertices, called $c$-vertices, each representing $\geq 1$ cluster of $N(s)$. The vertices are connected by the edges of
$F_s$ incident to large subclusters of $G_0(s)$, the edges of $F_1(s)$ incident to S-trees, the edges of $F_2(s)$, and by artificial edges between c-vertices. Thus $G_3$ graphs have to be updated if there is a change in either $F_s$, $F_1$, $F_2$, or $H_2$. After an update in $G$ only an amortized constant number of $G_3$ graphs have to be updated because of changes in the first three forests and only articulation points of $H_2$ have to be updated because of changes in $H_2$. Additionally, a vertex of $G_3(s)$ might have to be split if a large subcluster or a cluster of $N(s)$ is split. Updating $G_3(s)$ means reconstructing the whole graph and computing its connected components. This takes time $O(n(s) + m(s)/k)$.

Every update in $G$ changes at most a constant number of tree edges in $F_s$. Since every tree edge of $F_s$ is incident to large subclusters in at most four shared data structures (at most two for each endpoint), this requires an insertion and/or deletion in at most four $G_3$ graphs. Each edge is contained in at most two graphs $G_1$, requiring at most two $G_3$ graphs to be updated because of changes in a $F_1$ forest. As we showed above an amortized constant number of $F_2$ forests changes. Since each $F_2$ forest contributes to only one $G_3$ graph, only a amortized constant number of $G_3$ graphs have to be rebuilt because of changes in $F_s$, $F_1$, or $F_2$.

If an edge of $H_2$ is inserted, the $G_3$ graphs that have to be updated are the graphs of shared vertices whose cluster in $H_2$ is an articulation point in $H_2$ (before the insertion). Symmetrically, if an edge of $H_2$ is deleted, the $G_3$ graphs that have to be updated are the graphs of shared vertices whose cluster in $H_2$ is an articulation point in $H_2$ (after the deletion). Updating $G_3(s)$ takes time $O(n(s) + m(s)/k)$. Since all updated graphs correspond to articulation points in $H_2$, it follows that $O(\sum_{G_3(s) is updated} n(s)) = O(m/k)$. Each vertex is contained in sets $V(s)$ of at most two shared vertices $s$. Thus $O(\sum_{G_3(s) is updated} m(s)/k) = O(m/k)$. Hence updating the $G_3$ graphs because of changes in $H_2$ takes time $O(m/k)$.

Finally one of the vertices of $G_3(s)$, either a large subcluster or a cluster of $N(s)$, can be split. The split of a large subcluster of $G_0(s)$ requires to recompute only the graph $G_3(s)$. Note that the c-vertices of $G_3(s)$ and the c-vertices of $G_2(s)$ always represent the same clusters of $N(s)$. Thus the same argument as in the previous subsection shows that only an amortized constant number of graphs $G_3$ have to be rebuilt because clusters represented by the same c-vertex are no longer biconnected. Thus the amortized time to update all graphs $G_3$ is $O(m/k)$.

This shows the following lemma.

**Lemma 2.11** All shared graphs can be updated in time $O(B(m/k) + (k + m/k) \log n)$. The data structure tests in $O(1)$ time whether a shared vertex separates two of its tree neighbors.

### 2.5 Complete block queries

A complete block query determines all the blocks to which a vertex belongs by computing for each tree edge the block to which it belongs. A vertex belongs to exactly these blocks to which the tree edges adjacent to the vertex belong. We can find the blocks in $I(C)$ for every tree edge in or incident to the cluster $C$ in time $O(k)$ whenever we recompute $I(C)$. To compute the blocks of $G$, we have to determine which blocks of different clusters form a block of $G$.

Let $C$ and $C'$ be two clusters that are connected by a tree edge $e = (x, y)$ with $x \in C$ and $y \in C'$ and let $b$ and $b'$ be the blocks of $C$ and $C'$ to which $e$ belongs. If $e$ is a solid edge, then by
Lemma 2.3 $x$ and $y$, and thus $b$ and $b'$ belong to the same block of $G$.

If $e$ is a dashed edge belonging to a vertex $s$, then both $x$ and $y$ represent $s$. If either $b$ or $b'$ consists of only $e$, then no vertex of $C$ is biconnected with a vertex of $C'$. Thus no blocks of $C$ and $C'$ belong to the same block of $G$. If $b$ and $b'$ consist of more than one edges, let $u$ resp. $u'$ be a vertex of $G$ adjacent to $s$ belonging to $b$ resp. $b'$ (any such vertex can be used). According to Lemma 2.3, the blocks $b$ and $b'$ belong to the same block of $G$ if and only if $s$ does not separate $u$ and $u'$, i.e. if and only if $u$ and $u'$ are connected in the shared graph of $s$. Using $I(C)$ and the shared graphs, we can test in $O(1)$ time for any pair of adjacent clusters whether two of their blocks should be joined.

During a depth-first traversal of the spanning tree of $H_1$ we compute lists of the blocks of different clusters that have to be joined. Then we mark all the edges of blocks in the same list as belonging to the same block of $G$. Thus the total cost is proportional to the number of tree edges in $T$, which is $n - 1$.

**Theorem 2.12** A complete block query in a graph of $n$ vertices can be answered in time $O(n)$.

### 2.6 Biconnectivity queries

After each update operation we root the spanning tree of $H_1$ at a leaf cluster $R$ and compute for every cluster $C$ the lowest ancestor $A$ of $C$ that is an articulation point separating $C$ from $R$ and also the tree edge on $\pi_{H_1}(C, A)$ that is incident to $A$. This can be done in time $O(m/k + B(m/k))$ by first executing a complete block query in $H_1$ and then traversing the spanning tree of $H_1$ in depth first order, thereby maintaining a stack of all the articulation points separating the current cluster from $C$. The topmost articulation point on the stack is the lowest ancestor $A$ of the current cluster.

To test whether vertex $u$ and $v$ are biconnected, we check whether $C(u)$ and $C(v)$ store the same articulation point. If no, then one of the articulation points stored at $C(u)$ and $C(v)$ separates $C(u)$ and $C(v)$. By Lemma 2.3 it follows that $u$ and $v$ are not biconnected. If $C(u)$ and $C(v)$ store the same articulation point $A$ and the same tree edge, the path between $u$ and $v$ does not contain $A$ and thus no articulation point separates $C(u)$ and $C(v)$ in $H_1$. Lemma 2.3 shows that $u$ and $v$ are biconnected if and only if $u$ and the representative of $v$ are biconnected in $I(C(u))$, $v$ and the representative of $u$ are biconnected in $I(C(v))$, the shared vertex of $C(u)$ (if it exists) does not separate $u$ and $v$, and the shared vertex of $C(v)$ (if it exists) does not separate $v$ and $u$.

If $C(u)$ and $C(v)$ store the same articulation point $A$, but different tree edges, then $A$ is the only articulation point on $\pi_{H_1}(C(u), C(v))$. Thus $u$ and $v$ are biconnected if and only if the endpoints of the two tree edges stored at $C(u)$ and $C(v)$ belong to the same block of $I(A)$, the shared vertex of $A$ (if it exists) does not separate $u$ and $v$, $u$ (resp. $v$) and the representative of $v$ (resp. $u$) are biconnected in $I(C(u))$ (resp. $I(C(v))$), and the shared vertex of $C(u)$ (resp. $C(v)$) (if it exists) does not separate $u$ and $v$.

We store at each vertex its cluster and the (at most two) shared vertices $s$ to whose set $V(s)$ it belongs. At each cluster we store its representative for each internal and each shared graph. Since there are $O(m/k)$ clusters and $O(m/k)$ such graphs, this requires $O((m/k^2) = O(m)$ space. Using
this additional information, all the above test can be executed in constant time in the appropriate internal and shared graphs.

**Theorem 2.13** The given data structure can answer a biconnectivity query in constant time and can be updated in amortized time $O(\sqrt{m} \log n)$ after an edge insertion or deletion in $G$.

**Proof:** As was shown in [4, 2] maintaining a restricted partition of order $k$ and a spanning tree of $G'$ takes time $O(\sqrt{m}) = O(k + m/k)$ per update. Let $T(m)$ be the update time in a graph with $m$ edge and let $B(n)$ be the time for a complete block query in a graph with $n$ vertices. In Theorem 2.12 we showed that $B(n) = O(n)$ and in Lemma 2.2, 2.4, 2.6, and 2.11 we proved that the high-level graphs, the testing data structures, the internal graphs, and the shared graphs can be updated in time $26T((m/k)^2) + 2B(m/k) + O((k + m/k)\log n)$. Since a complete block query is executed in $H_1$ after each update operation when using the testing data structure, we only have to root the spanning tree of $H_1$ and store some information at each cluster to answer biconnectivity queries in constant time (see Section 2.6). This takes time $O(m/k)$. Thus $T(m) = 26T((m/k)^2) + 2B(m/k) + O((k + m/k)\log n)$. Choosing $k = 6\sqrt{m}$ gives the desired result.

3 Plane graphs

In this section we present an algorithm for fully dynamic biconnectivity in plane graphs with $O(\log n)$ query time and $O(\log^2 n)$ update time. We modify the extended topology tree data structure of [10] and prove that this data structure dynamically maintains biconnectivity information.

3.1 Definitions

As in general graphs (see Section 2) we transform a given graph $G$ into a degree-3 graph $G'$ by replacing every vertex $x$ of degree $d > 3$ with a chain of $d - 1$ dashed edges $(x_1, x_2), \ldots, (x_{d-1}, x_d)$. We say each $x_i$ is a representative of $x$ and $x$ is the original node of every $x_i$. Then we find an embedding of $G'$ and a spanning tree $T$ of $G'$. A topology tree of $G'$ based on $T$ is a hierarchical representation of $G'$ introduced by Frederickson [4]. On each level of the hierarchy it partitions the vertices of $G'$ into connected subsets called clusters. An edge is incident to a cluster if exactly one endpoint of the edge is contained in the cluster. The external degree of a cluster is the number of tree edges that are incident to the cluster. Each vertex of $G'$ is a level-0 cluster. Two clusters at level $i > 0$ are formed by either

1. the union of two clusters of level $i - 1$ that are joined by an edge in the spanning tree and either both of external degree 2 or one of them has external degree 1, or
2. one cluster of level $i - 1$, if the previous rule does not apply.

Each cluster at level $i$ is a node of height $i$ in the topology tree. If a cluster $C$ at level $i$ is formed by two clusters $A$ and $B$ of level $i - 1$, then $A$ and $B$ are the children of $C$ in the topology tree. If $C$ is formed by one cluster $A$ of level $i - 1$, then $A$ is the only child of $C$ in the topology tree. The
topology tree has depth $D = O(\log n)$ [4]. In the following node denotes a vertex of the topology tree.

In [10] the topology tree data structure is extended to maintain non-tree edges of $G'$ and additional connectivity information at each node, called recipe. We use the same technique to maintain dynamic 2-vertex connectivity.

Every insert$(u,v)$, delete $(u,v)$, or query$(u,v)$ operation requires that the topology tree is expanded at an (arbitrary) representative of $u$ and of $v$: We mark all clusters containing the two representatives in the topology tree. Note that all these clusters lie on a constant number of paths to the root. Then we build the graph which consists of the two representatives and a compressed representation of all the clusters that are unmarked children of a marked node in the topology tree. This creates a compressed version of $G$, called $G(u,v)$, of size $O(\log n)$. This graph is used to answer queries. In the case of update operations, the edge is added to or deleted from $G(u,v)$. Afterwards the topology tree is merged together again, i.e. a topology tree representation is created for the (possibly modified) graph $G(u,v)$.

To add non-tree edges to the topology tree data structure we define a bundle between two clusters $C$ and $C'$ as follows: If neither $C$ is an ancestor of $C'$ nor vice verse, let $e(C,C')$ be the set of all edges between $C$ and $C'$. Otherwise, assume wlog that $C'$ is the ancestor of $C$. We define $e(C,C')$ to be the set of all edges incident to $C$ whose least common ancestor in the topology tree is $C'$. Since we are considering an embedded graph, the edges incident to a cluster $C$ are embedded at $C$ in a fixed circular order. A bundle between a cluster $C$ and $C'$ is a subset of $e(C,C')$ that forms a maximal continuous subsequence in the circular order at $C$ and $C'$. Note that this definition is independent of the level of the clusters and planarity guarantees that there are at most three bundles between two clusters [10]. The first and last edge of a bundle in this order are called the extreme edges of the bundle. In the topology tree a bundle between $C$ and $C'$ is represented by two bundles, one from $C$ to the least common ancestor of $C$ and $C'$ (called the LCA-bundle of $C$) and one from $C'$ to the least common ancestor. Whenever the topology tree is expanded and the graph $G(u,v)$ is created, we convert these two bundles back into one.

An edge $(u,v)$ with $u,v \in C$ is called an internal edge of the cluster $C$. Assume all dashed internal edges of $C$ are contracted. The projection of an edge $(x,y)$ onto a tree path $P$ is the path $\pi(x,y) \cap P$. Note that, by definition, the vertices of each cluster are connected by a subtree of $T$. In the following we define the projection edge of an edge, the projection path $p(C)$, the coverage graph of $C$ which consists of small and big supernodes of $C$. All these definitions are independent of the level of the cluster.

- If $C$ has external degree 1, the projection path $p(C)$ of $C$ consists of the endpoint $z$ of the (unique) tree edge incident to $C$. This endpoint is a small supernode. The coverage graph of $C$ consists of this supernode and of all LCA-bundles of $C$. For each edge $e$ incident to $C$ where $y$ is the endpoint in $C$, the projection edge of $e$ is $e$ if $y = z$ and otherwise the tree edge incident to $z$ that lies on $\pi(y,z)$.

- If $C$ has external degree 3, it consists of only one vertex $z$. Both, the projection path $p(C)$ and the coverage graph consist of only this one vertex which is a small supernode.
• If the external degree of a cluster $C$ is 2, there is a unique simple tree path between the tree edges that are incident to $C$. This path is the projection path $p(C)$ of $C$. The projection $p(x)$ of a vertex $x$ in $C$ is the vertex closest to $x$ on the projection path. The projection edge of a vertex $x$ is the edge on $\pi(x, p(x))$ incident to $p(x)$. If $x = p(x)$, the projection edge of $x$ is undefined. The projection edge of an edge $(x, y)$ with one endpoint $x$ in $C$ is the projection edge of $x$, if it is defined and it is $(x, y)$ otherwise. The projection edges of an edge $(x, y)$ with $x, y \in C$ are the projection edge of $x$ if it is defined and $(x, y)$ otherwise and also the projection edge of $y$ if it is defined and $(x, y)$ otherwise.

If $(x, y)$ is an internal edge of $C$, then the subpath $\pi(x, y) \cap p(C)$ is the projection of $(x, y)$ on $p(C)$, $p(x)$ and $p(y)$ are the extreme vertices of the projection, and all vertices on the subpath except for $p(x)$ and $p(y)$ are the internal vertices of the projection.

Let $(w, z)$ and $(x, y)$ be the extreme edges of a LCA-bundle between a cluster $C$ and a cluster $C'$ with $w, x \in C$ and $z, y \in C'$. The path $\pi(w, x) \cap p(C)$ is called the projection of the edge bundle on $p(C)$, $p(w)$ and $p(x)$ are called the extreme vertices of the projection and all vertices on the subpath except $p(w)$ and $p(x)$ are internal vertices of the projection. The projection edges of a bundle are the projection edges of the extreme edges of the bundle.

The coverage graph of $C$ is built by compressing $p(C)$ as follows:

1. Let $u_1, u_2, \ldots, u_p$ be a maximal subpath of $p(C)$ such that
   - $\pi(u_1, u_p)$ intersects the projection of a LCA-bundle on $p(C)$,
   - $u_1$ is the extreme vertex of the projection of a LCA-bundle or an internal edge,
   - $u_p$ is the extreme vertex of the projection of a LCA-bundle or an internal edge, and
   - every vertex $u_i$ for $1 < i < p$ is an internal vertex of the projection of a bundle or an internal edge or
     there exist two projections with projection node $u_i$ and the same projection edge at
     $u_i$ such that $u_i$ and $u_j$ with $j < i$ are the extreme vertices of one projection and $u_i$ and $u_k$ with $k > i$ are the extreme vertices of the other projection.

   If $p > 2$, we contract the path $u_2, \ldots, u_{p-1}$ to one vertex $u$, called big supernode, and we say $u_2, \ldots, u_{p-1}$ are replaced by the big supernode. The vertices $u_1$ and $u_p$ are called small supernodes and the edges $(u_1, u)$ and $(u, u_p)$ are called superedges. All edges incident to $u_2, \ldots, u_{p-1}$ are now incident to $u$. This splits a bundle that is incident to $u_1$ and/or $u_p$ and also $u_i$ with $1 < i < p$ into up to three subbundles, one incident to $u$ and the other(s) incident to $u_1$ and/or $u_p$. If the edge $(u_1, u_2)$ (resp. $(u_{p-1}, u_p)$) is dashed, then the edge $(u_1, u)$ (resp. $(u, u_p)$) is dashed.

   If $p \leq 2$ then no nodes are compressed.

2. After replacing all subpaths that fulfill condition 1, let $v_1, v_2, \ldots, v_q$ be a subpath of $p(C)$ such that $v_1$ and $v_q$ are two small supernodes and no vertex $u_i$ with $1 < i < q$ is a supernode. We contract the path $v_2, \ldots, v_{q-1}$ to one superedge $(v_1, v_q)$ and we say $v_2, \ldots, v_{q-1}$ are replaced by the superedge. If all edges $(v_1, v_2), \ldots, (v_{q-1}, v_q)$ are dashed, then the superedge is dashed, otherwise it is solid.
The coverage graph of $C$ consists of this compressed representation of $p(C)$ and all LCA-bundles grouped into sets according to their projection edges.

Note that our definition of a supernode replaces a supernode of [10] by two small and one big supernode and each bundle is split into at most three subbundles, one incident to each small supernode and one incident to the big supernode.

When expanding the topology tree, we build the coverage graph for each node that was marked and each child of a marked node. For each subbundle that is incident to a supernode in a coverage graph we maintain its projection edges implicitly as described below.

The coverage graph of a cluster $C$ is maintained as a doubly linked path of supernodes. Each supernode stores up to two doubly linked lists of projection edges incident to it (called projection list), one list for each side of the tree path $p(C)$. Each projection edge $e$ stores a double linked list of the subbundles such that $e$ is the projection edge of the subbundle. If $C$ has external degree 1, there is only one supernode, and only one list of projection edges. If $C$ has external degree 3, it consists of only one supernode without any projection edges or subbundles. The projection edges and the subbundles are listed in the counterclockwise order of their embedding. Only the first and last subbundle in a list have direct access to the projection edge and only the first and last projection edge in a list have direct access to the supernode to which they are incident. The data structure lets us coalesce two adjacent supernodes or two projection lists into one in constant time; we can also split a supernode or a projection edge list into two in constant time if we are given pointers that tell where to split the lists. Note that each subbundle can be contained in at most two lists and if it is contained in two lists, it is the first element of the one and the last element of the other list.

3.2 Recipes

Each node in the topology tree is enhanced by a recipe that describes how the coverage graph of the children of the node can be created from the coverage graph of the node. The only difference in the algorithm of [10] and this biconnectivity algorithm is in the contents of the recipes. We describe our recipes in the following. A recipe contains four kinds of instructions:

1. Split a subbundle. Replace a subbundle of $m$ edges that have the same target by up to four adjacent subbundles that have that target and whose (specified) sizes sum to $m$.

2. Split a projection edge. Split the subbundle list at specified locations and replace the old subbundle list at the supernode by the new subbundle lists.

3. Split a supernode. Split the two projection lists on either side of the supernode into two pieces at specified locations. Replace the old supernode by two new ones linked by a superedge, and give the appropriate piece of each projection list to each of the new supernodes.

4. Create a new subbundle. Create a subbundle with a specified target and number of edges, and insert it at a specified place in a subbundle list of at most two projection edges.
Using these instructions the coverage graphs of the children of a cluster $C$ can be transformed into a coverage graph of $C$. The sequence of instructions together with the appropriate parameters (e.g. which subbundle list has to be split at which location) is called a recipe and is stored at the node in the topology tree that represents $C$. These parameters are either a record of a subbundle (consisting of the number of edges in the subbundle and its target), a record of a projection edge (consisting of the edge), or a pointer, called location descriptor. A location descriptor consists of a pointer to a subbundle and an offset into the subbundle (in terms of number of edges) or a pointer into a projection list. It takes constant time to follow a location descriptor.

Whenever we expand the topology tree, we use the recipes to create the coverage graphs along the expanded path. Whenever we merge the topology tree, we first determine how to combine the coverage graphs of two clusters to create the coverage graph of their parent, and then we remember how to undo this operation in a recipe. We now describe the instructions in the recipe of $C$, depending on the number of children of $C$ and their external degrees. In the following subbundle stands for LCA-subbundle.

**Case 1**: $C$ has only one child.

In this case the coverage graph of $C$ is identical to the coverage graph of its child. The recipe is therefore empty.

**Case 2**: $C$ has two children with external degrees 3 and 1.

Let $Y$ be the child with external degree 3 and let $Z$ be the child with external degree 1. The coverage graph of $Y$ and of $Z$ consists of one supernode. We build the coverage graph of $C$ by as follows:

If the tree edge between $Y$ and $Z$ is dashed, we simply contract it by making the projection list of $Z$ the projection list of one side of the path of $C$. The projection list of the other side is empty. The projection edges of the bundles do not change and, thus, the subbundle lists do not change.

If the edge $(Y, Z)$ is not dashed, then the supernode of $C$ has only one projection edge, namely the tree edge between $Y$ and $Z$. Thus, the supernode of $C$ has one projection list (the projection list of the other side is empty) containing one projection edge. The subbundle list of this projection edge consists of the concatenation of all subbundle lists of $Z$. In the recipe we use location descriptors to point to the locations of the concatenation. The number of location descriptors is proportional to the number of removed projection edges.

**Case 3**: $C$ has two children, both with external degree 1.

In this case $C$ is the root of the topology tree. Its coverage graph is empty. The coverage graphs of the children contain one supernode and at most one subbundle apiece, corresponding to the set of non-tree edges linking the children. Since each subbundle is contained in at most 2 projection lists, there are at most 4 projection lists. The recipe stores these projection lists (i.e. whether a bundle is contained in 1 or 2 lists) and subbundles (i.e. the number of non-tree edges linking the children).
Case 4: $C$ has two children with external degrees 2 and 1.

Let $Y$ be the child of degree 2 and $Z$ be the child of degree 1. We collapse all supernodes of $Y$ to one supernode $s$ to build the coverage graph of $C$ from the coverage graph of $Y$ as follows: On each side of the tree edge between $Y$ and $Z$ there may be a subbundle that connects $Y$ and $Z$. We remove these subbundles and make all remaining subbundles incident to $s$.

If the edge $(Y, Z)$ is dashed, then the projection edge of the subbundles incident to $Y$ does not change. Thus we concatenate the two projection lists of $Y$ and the projection list of $Z$ (in the order of the embedding). This creates a single supernode with a single projection list.

If the edge $(Y, Z)$ is solid, then this edge becomes the projection edge for all subbundles incident to $Y$. Thus we concatenate all bundle lists of all projection edges of $Y$ to create the bundle list for $(Y, Z)$. Then we concatenate the two projection lists of $Y$ and the projection list of $Z$ (in the order of the embedding). This creates a single supernode with a single projection list.

In both cases, if two newly adjacent subbundles have the same target, we merge them into one subbundle and update the subbundle and projection lists appropriately.

In the recipe we need a location descriptor to point to each subbundle where we concatenated projection lists or subbundle lists or merged subbundles. We also have to store any subbundles that connect $Y$ and $Z$ and all projection edges that we removed. The number of location descriptors we store is proportional to the number of supernodes of $Y$ plus the number of removed projection edges.

Case 5: $C$ has two children, both with external degree 2.

Let $Y$ and $Z$ be the children of $C$. To join the coverage graphs of $Y$ and $Z$ we consider two cases: If the tree edge between $Y$ and $Z$ is dashed, we join the 2 coverage graphs by identifying the appropriate small supernodes (that are terminating the coverage graphs) and concatenating their projection lists. If the tree edge between $Y$ and $Z$ is not dashed, we connect the 2 coverage graphs by an edge.

In both cases we remove then all subbundles between $Y$ and $Z$. If one of the supernode that was incident to a removed subbundle is no longer incident to a bundle, we replace it by a superedge. Afterwards we coalesce all the supernodes between the $(Y, Z)$-subbundle endpoints into three supernodes as follows: If the path $P$ between their endpoints contains only one supernode other than the endpoints, nothing has to be done. Otherwise, we replace these (at least 2) supernodes by one supernode by concatenating their projection lists. We also merge newly adjacent subbundles into a single subbundle if they have the same target.

The recipe contains a location descriptor pointing to each subbundle where we coalesced supernodes and concatenated projection lists (and possibly merged adjacent subbundles). We also store the subbundles that were merged together or deleted. If there is a subbundle that loops around the tree, we need two more location descriptors to mark its endpoints. The number of location descriptors is proportional to the number of coalesced supernodes in $Y$ and $Z$.  

23
New subbundles may be created during recipe evaluation. For each new subbundle, the recipe stores a bundle record, preloaded with the count of bundle edges, and a location descriptor pointing to the place in the old subbundle list where the new subbundle is to be inserted. The target field of the subbundle is easy to set: the least common ancestor of the bundled edges is exactly the node at which the recipe is being evaluated. In a way similar to [10] we can show the following lemma.

**Lemma 3.1** If the topology tree is expanded at a constant number of vertices and recipes are evaluated at the expanded clusters, the total number of edge bundles, supernodes, and superedges created is $O(\log n)$. The expansion takes $O(\log n)$ time.

**Proof:** Since the topology tree has depth $O(\log n)$, there are $O(\log n)$ marked nodes and $O(\log n)$ children of marked nodes. Thus, the cluster graph consists of the coverage graph of $O(\log n)$ clusters. Planarity guarantees that these clusters are connected by $O(\log n)$ bundles, each bundle is split into up to three subbundles. Thus there are $O(\log n)$ subbundles. Since each supernode in a cluster with more than one supernode is incident to a subbundle, there are $O(\log n)$ supernodes. Because the supernodes and superedges form a tree, the number of superedges is also $O(\log n)$. Each subbundle has two projection edges. Thus the total number of projection edges is $O(\log n)$.

Evaluating a recipe takes time proportional to the number of supernodes or projection edges created by the recipe plus constant ‘overhead’ time. Thus the total expansion time is $O(\log n)$. ⊣

### 3.2.1 Queries

To answer a query $(u, v)$, we mark all the clusters containing $u$ and $v$ in the topology tree. Then we create the graph $G(u, v)$ in the following steps:

1. We build the cluster graph by expanding the topology tree at a representative of $u$ and of $v$.
2. Let $e_1, e_2, \ldots, e_p$ with $p > 1$ be all the subbundles whose extreme edges have the same projection edge $(x, y)$ in a cluster $C$ with $x \in p(C)$. We add a small supernode $y$ and connect all these extreme edges to $y$.
3. We contract all dashed edges. When contracting a dashed edge between two supernodes, the resulting supernode is a small supernode.

Since the cluster graph consists of $O(\log n)$ supernodes, subbundles, and superedges and can be computed in time $O(\log n)$, the graph $G(u, v)$ resulting from these 3 steps contains $O(\log n)$ supernodes, subbundles, and superedges and can be computed in time $O(\log n)$.

The following lemmata show that two vertices $u$ and $v$ are not biconnected in $G$ if and only if there is an articulation point in $G(u, v)$ separating $u$ and $v$ that is not a big supernode. Since the cluster graph has size $O(\log n)$ this can be tested in time $O(\log n)$.

**Lemma 3.2** Let $u$ and $v$ be two vertices of $G_2$ and of $G_1$ and let $G_2$ be a graph created from $G_1$ by
• contracting connected subgraphs into one vertex,
• replacing the only two edges \((a, b)\) and \((b, c)\) incident to a vertex \(b\) by the edge \((a, c)\),
• replacing parallel edges, and
• removing self-loops.

Let \(x\) be a vertex of \(G_1\) that is not contained in the contracted subgraphs and not a removed degree-2 vertex. Then \(x\) is an articulation point in \(G_1\) separating \(u\) and \(v\) if and only if \(x\) is an articulation point separating \(u\) and \(v\) in \(G_2\).

**Proof:** Consider first the case that \(x\) separates \(u\) and \(v\) in \(G_1\). To achieve that \(u\) and \(v\) are not separated by \(x\) in \(G_2\) a cycle has to be created that contains \(u, x,\) and \(v\). Contracting pieces of \(G_1\) that do not contain \(x\) or removing degree-2 vertices (other than \(x\)) cannot create new cycles. Thus \(x\) is also an articulation point separating \(u\) and \(v\) in \(G_2\).

If \(x\) separates \(u\) and \(v\) in \(G_2\), then expanding vertices (other than \(x\)) of \(G_2\) to connected subgraphs, replacing one edge by two edges and a degree-2 vertex or adding parallel edges to edges not on \(\pi(u, v)\) and self-loops does not create a cycle that contains \(x, u,\) and \(v\). Thus \(x\) separates \(u\) and \(v\) also in \(G_1\).  

**Lemma 3.3** Let \(u\) and \(v\) be two vertices of \(G\). The graph \(G(u, v)\) is created from \(G\) by

• contracting connected subgraphs into one vertex,
• replacing the only two edges \((a, b)\) and \((b, c)\) incident to a degree-2 vertex \(b\) by the edge \((a, c)\),
• collapsing parallel edges, and
• removing self-loops.

No small supernode on \(\pi(u, v)\) (except for \(u\) and \(v\) itself) in \(G(u, v)\) is contained in a contracted subgraph of any of these operations.

**Proof:** The graph \(G(u, v)\) can be created from \(G\) by the three operation given in the lemma using the following steps. Note that \(G(u, v)\) does not contain dashed edges and every small supernode of \(G(u, v)\) represents a unique vertex \(x\) of \(G\).

1. Mark all the nodes that are small supernodes of \(G(u, v)\) red.
2. Collapse all nodes on the tree path between two red nodes to one blue node.
3. Contract every blue node and all the subtrees whose root is uncolored and connected to the blue node by a tree edge to a green node.
   Now we are left with red, green, and uncolored nodes and every green node is connected by tree edges to two red nodes.
4. Replace all parallel edges by one edge and remove all self-loops.
5. Replace every degree-2 green node by a superedge. (All remaining green nodes correspond to big supernodes.)
6. If a red node $x$ lies on $\pi_G(u, v)$ and does not lie on $\pi(u, v)$, shrink all subtrees whose root is uncolored and connected by a tree edge to $x$ to a yellow node. Otherwise contract all subtrees whose root is uncolored and connected to $x$ by a tree edge to the node $x$.

7. Replace all parallel edges by one edge and remove all self-loops.

The resulting graph is $G(u, v)$. Note that $u$ and $v$ are small supernodes in $G(u, v)$ and then marked red. Hence, if a small supernode $x$ lies on $\pi(u, v)$, it is not replaced by step 6. No small supernodes are contained in a connected subgraph that is contracted in step 1-5. The lemma follows.

**Lemma 3.4** No vertex on a subpath that is replaced by a big supernode in $G(u, v)$ is an articulation point separating $u$ and $v$ in $G$.

**Proof:** Let $C$ be the cluster of $G(u, v)$ containing a vertex $x$ that is replaced by a big supernode. Since $x$ is replaced by a big supernode, it follows that $x$ is an internal vertex of the projection path $P$ creating this supernode. Let $z_1$ and $z_2$ be the extreme vertices of $P$. Then $z_1$ and $z_2$ are connected by a path in $G$ that does not use $x$. It follows that $x$ does not separate $u$ and $v$ in $G$.

**Lemma 3.5** If a vertex $x$ of $G$ is replaced by a superedge $(y, z)$ and if $x$ separates $u$ and $v$ in $G$, then $y$ and $z$ also separate $u$ and $v$ in $G$.

**Proof:** Let $C$ be the cluster of $G(u, v)$ that contains $x$, let $v_1, v_2, \ldots, v_q$ be the subpath $P$ that is replaced by $(y, z)$ with $y = v_1$ and $z = v_q$ and $x = v_i$ for some $1 < i < q$. Wlog let the tree path from $v_2$ to $u$ contain $v_1$. From the definition of a superedge it follows that no vertex $v_i$ with $1 < i < q$ is a supernode. Thus the projection of none of the subbundles incident of $C$ (i.e. edges with one endpoint in $C$) contains a vertex $v_i$. Since $x$ is an articulation point separating $u$ and $v$, no edge with both endpoints outside $C$ exists whose projection on $\pi(u, v)$ contains a node $v_i$ for $1 \leq i \leq q$. Since $v_1$ is a small supernode, it is the extreme vertex of a projection of an edge or subbundle whose projection onto $\pi(u, v)$ lies inside $\pi(v_1, u)$. Thus no edge or subbundle exists whose projection onto $\pi(u, v)$ contains a vertex on $\pi(v_1, u)$ other than $v_1$ and $v_2$. Additionally, if such a projection contains $v_1$ it does not have the same projection edge as any edge whose projection contains $v_2$. Thus every path from $u$ to $v_2$ contains $y$ and, hence, every path from $u$ to $v$ contains $y$. The symmetric argument shows that $z$ separates $u$ and $v$ in $G$.

**Lemma 3.6** Two vertices $u$ and $v$ are not biconnected in $G$ if and only if there is an articulation point separating $u$ and $v$ that is not a big supernode in the cluster graph $G(u, v)$.

**Proof:** Lemma 3.3 shows that the cluster graph $G(u, v)$ is created from $G$ by contracting subgraphs, removing degree-2 nodes, collapsing parallel edges, and removing self-loops. Thus Lemma 3.2 does apply with $G_1 = G$ and $G_2 = G(u, v)$.
Let \( x \) be an articulation point separating \( u \) and \( v \) in \( G \). Then \( x \) lies on \( \pi(u,v) \). From Lemma 3.4 it follows that \( x \) is cannot be represented by a big supernode in \( G(u,v) \). If \( x \) is represented by a small supernode, then according to Lemma 3.3, \( x \) was not affected by the contraction of \( G \) to \( G(u,v) \). Thus Lemma 3.2 shows that \( x \) is an articulation point separating \( u \) and \( v \) in \( G(u,v) \). If \( x \) is represented by a superedge \((y,z)\), then according to Lemma 3.5 \( y \) is an articulation point separating \( u \) and \( v \) in \( G \) as well. Since \( y \) is a small supernode, the same argument as above shows that \( y \) separates \( u \) and \( v \) in \( G(u,v) \).

If a small supernode \( x \) is an articulation point separating \( u \) and \( v \) in \( G(u,v) \), then by Lemma 3.3 \( x \) was not part of a contracted subgraph. It follows from Lemma 3.2 that \( x \) is an articulation point separating \( u \) and \( v \) in \( G \).

Theorem 3.7 The given data structure can answer biconnectivity queries in time \( O(\log n) \).

Proof: Lemma 3.6 shows that to test the biconnectivity of \( u \) and \( v \) in \( G \) it suffices to test whether \( u \) and \( v \) are separated by a small supernode in \( G(u,v) \). Since \( G(u,v) \) has size \( O(\log n) \), this can be done in time \( O(\log n) \).

3.3 Updates

An \( \text{insert}(u,v) \) or \( \text{query}(u,v) \) operation consists of 3 steps. First the topology tree is expanded at a representative of \( u \) and of \( v \) to create the cluster graph as discussed in Section 3.2. Second, we add or remove the edge \((u,v)\) from the cluster graph. Third we merge the topology tree back together.

By adding or deleting a constant number of vertices and edges we guarantee that the graph stays a degree-3 graph. Note that if a tree edge is deleted, we run along the faces adjacent to \((u,v)\) to find a subbundle that connects the two disconnected spanning trees. We can determine one of the edges of the subbundle by repeatedly expanding the clusters containing the endpoints. This edge becomes the new tree edge.

The details of merging the topology tree back together are given in [10]. There are three basic steps. First, the new topology tree for the updated cluster graph is computed. Second, the new subbundles and their LCA-targets are computed. Third, the recipes in all clusters that are affected by the modification of subbundles are recomputed. Step one and two are identical to [10] and take time \( O(\log n) \).

In step three of [10] the recipe of clusters is recomputed that contain the endpoint of an extreme edge of a modified subbundle. The following lemma shows that with the recipes described in Section 3.2 it suffices to update these clusters also for 2-vertex connectivity. Thus the same algorithm as in [10] can be used to update the data structure after each update operation.

Lemma 3.8 If a subbundle is split into a constant number of subbundles or if a constant number of subbundles are merged, the only recipes that have to be updated are the recipes of clusters containing the endpoints of the extreme edges of the modified subbundles.

Proof: A recipe at a cluster \( C \) contains location pointers into subbundles, subbundle lists, and projection lists. Additionally, it contains subbundle records and projection edges.
All subbundles in the subbundle lists of $C$ are incident to $C$. All projection edges for whom we keep a projection list at $C$ are projection edges of subbundles whose extreme edges have at least one endpoint in $C$. These lists only have to be updated if one of these subbundles are modified.

For each subbundle whose record is stored in the recipe or that is pointed to by a location descriptor at least one of the endpoints is contained in $C$. The record only has to be updated if the subbundle is modified.

Each projection edge that is stored in the recipe is the projection edge of a subbundle whose extreme edges have at least one endpoint in $C$. The projection edge information only changes if this subbundle is modified.

Theorem 3.9 The given data structure can be updated in time $O(\log^2 n)$ after an edge insertion or deletion.

4 Conclusions

We presented data structures that allow insertions and deletions of edges in a graph and queries whether two nodes are biconnected. In general graphs the amortized update time is $O(\sqrt{m} \log n)$ and the query time is $O(1)$, in plane graphs the update time is $O(\log^2 n)$ and the query time is $O(\log n)$. The fastest known algorithm for connectivity and 2-edge connectivity in general graphs is $O(\sqrt{n})$. For both problems a dynamic algorithm was developed with update time $O(\sqrt{m})$ [4, 5] that was consequently speeded up with sparsification to $O(\sqrt{n})$ [2]. Thus, an interesting question is whether there exists a sparse certificate for 2-vertex connectivity so that the sparsification technique can speed up the algorithm presented in this paper to $O(\sqrt{n} \log n)$. We are currently investigating this question.

Since the running time for dynamic connectivity and 2-edge connectivity problems in plane and planar graphs [1, 3, 10] and dynamic 2-vertex connectivity in plane graphs is only polylogarithmic in the size of the graph, an obvious open problem is whether any of these problems can be solved in time $o(\sqrt{n})$ in general graphs. The only known lower bounds are $\Omega(\log n / \log \log n)$ per operation [13]. Improving these lower bounds is another interesting problem.

Finally, the sparsification technique allows insertions and deletions of edges, but no splits of vertices. As we showed in Section 2, sparsification can be combined with vertex splits in the case that the graph $G = (V_1 \cup V_2, E)$ is bipartite and only vertices of $V_1$ are split. It would be interesting to explore whether this observation can be applied also to speed up other dynamic problems.

References


