SIAM J. COMPUT.
Vol. 29, No. 6, pp. 1761–1815

IMPROVED DATA STRUCTURES FOR FULLY DYNAMIC BICONNECTIVITY*

MONIKA R. HENZINGER†

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 deterministic algorithms is $O(\sqrt{n})$ 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 $O(\sqrt{m \log n})$ 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 $O(n)$.

Key words. dynamic graph algorithms, biconnectivity, data structures

AMS subject classifications. 68P05, 68Q25

PII. S0097539794263907

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 data structures that, given an input graph G, maintain the solution of a graph problem in G while G is modified by insertions and deletions of edges.[1] In this paper we study the problem of maintaining the biconnected components (see below) of a graph.

We say that a vertex x is an *articulation point separating* vertex u and vertex v (or that x separates u and v) if the removal of x 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 a maximal set of vertices that are biconnected (resp., 2-edge connected). Note that biconnectivity implies 2-edge connectivity but not vice versa.

Given a graph $G = (V, E)$, a *dynamic biconnectivity algorithm* is a data structure that executes an arbitrary sequence of the following operations:

insert(u, v): Insert an edge between node u and node v.

delete(u, v): Delete the edge between node u and node v if it exists.

query(u, v): Returns *yes* if u and v are biconnected, and *no* otherwise.

complete-block-query: Return for all nodes all the blocks they belong to.

Operations *insert* and *delete* are called *updates*. To compare the asymptotic performance of dynamic graph algorithms, the time per update, called *update time*, and the time per query, called *query time*, are compared. Let m be the number of edges

*Received by the editors March 2, 1994; accepted for publication (in revised form) July 21, 1999; published electronically April 4, 2000.

http://www.siam.org/journals/sicomp/29-6/26390.html

†Google, Inc., 2400 Bayshore Parkway, Mountain View, CA 94043 (monika@google.com). This research was done while at the Department of Computer Science, Cornell University, Ithaca, NY 14853.

[1] Insertions or deletions of isolated vertices are usually trivial.

and n be the number of vertices in the graph. Prior to this work, the best update time was $O(\min(m^{2/3}, n))$ [11, 2] with a constant query time. This paper presents an algorithm with $O(\sqrt{m \log n})$ update time and constant query time. Subsequently, the sparsification technique was applied to the algorithm in this paper and its running time was improved to $O(\sqrt{n \log n} \log(m/n))$ [13]. Recently, a data structure that requires only polylogarithmic amortized time per operation was presented [15]. The data structure presented in this paper can answer complete-block-queries in time $O(n)$. This was not mentioned explicitly but can also be done with the data structure in [11, 2].

Additionally, we give an algorithm with $O(\log^2 n)$ update time and $O(\log n)$ query time for planar embedded graphs, under the condition that each insertion maintains the planarity of the embedding. The best previous algorithm took time $O(\sqrt{n})$ per update and $O(\log n)$ per query.

Related work. Frederickson [5] 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. 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 [6]. The sparsification technique of Eppstein et al. [3] and Eppstein, Galil, and Nissenzweig [2] improves the running time of an update operation to $O(\sqrt{n})$. Subsequently, a dynamic connectivity algorithm was given with $O(n^{1/3} \log n)$ update time and $O(1)$ query time [12]. It can also output all nodes connected to a given node in time linear in their number. Very recently, an algorithm with polylogarithmic amortized time per operation was presented [15]. Note that there is a lower bound on the amortized time per operation of $\Omega(\log n / \log \log n)$ for all these problems [7, 16].

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, Rauch, and Suri [14] and Eppstein et al. [4], and $O(\sqrt{n})$ for maintaining biconnected components by Eppstein et al. [4].

Outline of the paper. First (section 2), we study the dynamic biconnectivity problem for *general* graphs. Our basic approach is to partition the graph G into small connected subgraphs called *clusters* (see [5] for a first use of this technique in dynamic graph algorithms). Each biconnectivity query between a vertex u and a vertex v can be decomposed into a query in the cluster of u, a query in the cluster v, and a query between clusters. To test biconnectivity between clusters we use the 2-dimensional topology tree data structure [5] in a novel way and extend the ambivalent data structure [6]. These data structures were used before to test connectivity and 2-edge connectivity.

To test biconnectivity within a cluster we need to know how the vertices outside the cluster are connected with each other. Thus, we build two graphs, called *internal* and *shared* graphs. Each graph contains all vertices and edges inside the cluster C and a *compressed certificate* of $G \setminus C$. A compressed certificate is a graph that has the same connectivity properties as $G \setminus C$, but is not necessarily a minor of $G \setminus C$. This approach is similar to the concept of strong certificates in the sparsification technique: a strong certificate is not necessarily a subgraph of the given graph. The crux in the analysis of the algorithm is that we can show that only an amortized constant number of compressed certificates need "major" updates after an update in G (see Lemma 2.44).

Second (section 3), we study the dynamic biconnectivity problem for *plane* graphs. We use a topology tree approach based on [5].

An earlier version of this paper appeared in [17].

2. General graphs. Let G be an undirected graph with n vertices and m edges. The *size* $|G|$ of a graph is the total number of its nodes and edges. We assume in the paper that G is connected, which implies $m \geq n-1$. If G is not connected, we build the data structure described below for each connected component and during an update combine two data structures or split a data structure in time $O(\sqrt{m \log n})$.

Let $m/\log^2 n \geq k \geq \sqrt{m}$ be a parameter to be determined later. Note that $m/k \leq k$. We build a data structure for G that we rebuild from scratch every m/k update operations. The operations between two rebuilds form a *phase*. This allows us to limit the necessary maintenance of the data structure within a phase, i.e., the data structure slowly "deteriorates" within a phase.

The data structure consists of the following parts: We map G to a graph G' of degree at most 3 and partition G' into $O(m/k)$ many subgraphs of size $O(k)$, called *clusters*. We keep data structures for (1) the graph of clusters, (2) each cluster (called *cluster graph*), and (3) for special *shared* nodes (called *shared graphs*). We will show how to rebuild these data structures in time $O(m \log n)$ and update them in time $O(\sqrt{m \log n})$ after an edge insertion or deletion in G.

2.1. The graph G' and the relaxed partition of order k. We want to partition G into about equally sized subgraphs (see the relaxed partition below). For this purpose, we map G to a graph G' of degree at most 3 as in [5]. At each rebuild, G' is created by *expanding* a vertex u of degree $d \geq 4$ by $d-2$ new degree-3 vertices $u'_1, \ldots, u'_{d-2}$ and connecting u'_i and u'_{i+1} by a *dashed* edge, for $1 \leq i \leq d-3$. Every node u of degree at most 3 is represented by one node u'_1 in G'. Every edge (u, v) is replaced by a *solid* edge (u'_i, v'_j), where i and j are the appropriate indices of the edge in the adjacency lists for u and v. We say that the edge (u'_i, u'_{i+1}) *belongs* to u and that every u'_i is a *representative* of u. The vertex u of G is called the *origin* of the vertex u'_i in G', for $1 \leq i \leq d-2$. We denote vertices of G' by variables with prime, like u' or u'_i, and their origin by variables without prime, like u. Thus, at the beginning of a phase the graph G' contains at most $2m$ vertices and at most $3m$ edges. We use $deg(u)$ to denote the degree of a vertex u in G and $num(u)$ to denote the number of representatives of u in G'.

The graph G' is maintained during insertions and deletions of edges as follows: Consider how the representatives of u are updated when an edge (u, v) is inserted. If $num(u)$ is 1 and $deg(u)$ was 3 before the insertion, then add a new vertex u'_2, connect u'_1 and u'_2 by a dashed edge, and make u'_1 and u'_2 each incident to 2 edges incident to u. If $num(u)$ is greater than 1, then one new vertex $u'_{num(u)+1}$ is created and connected by a dashed edge to $u'_{num(u)}$. Thus an edge insertion increases the number of vertices in G' by up to 2 and the number of edges by up to 3. If an edge is deleted, the corresponding edge is removed from G', *but no vertices are deleted from* G'. Thus, within a phase there are at most $2m + 2m/k$ vertices and $3m + 3m/k$ edges in G'.

Data structure: The algorithm keeps the following mapping from G to G' and vice versa:

(G1) Each vertex of G stores a list of its representatives, ordered by index, and pointers to the beginning and the end of this list. Each vertex of G' keeps a pointer to its position in this list, to its origin, and a list of incident edges.

(G2) Each dashed edge of G' stores a pointer to the vertex of G that it belongs

to; each solid edge of G' stores a pointer to the edge of G that it represents.

Note that there exists a spanning tree of G' that contains every dashed edge. The algorithm maintains such a spanning tree, denoted by T'. Let T be the corresponding spanning tree in G. We denote by $\pi_{T'}(u', v')$ the path from u' to v' in T'. If the spanning tree is understood, we use $\pi(u', v')$. Let u'_v denote the representative of u with the shortest tree path to a representative of v. Note that every articulation point separating u and v must have a representative that lies on $\pi_{T'}(u'_v, v'_u)$.

Data structure:

(G3) Both T and T' are stored in a degree-k ET-tree data structure [12] and in a dynamic tree data structure [18]. The ET-tree has constant depth.

We build a "balanced" decomposition of G' into subgraphs of size $O(k)$ and maintain data structures based on this decomposition. At the beginning of each phase, the decomposition and data structures are rebuilt from scratch in time $O(m \log n)$, adding an amortized cost of $O(k \log n)$ to each update. The rebuilds significantly simplify the "rebalancing" operations needed to maintain the decomposition balanced during updates.

We next describe the balanced decomposition of G' into *clusters*. A *cluster* is a set of vertices of G' that induces a connected subgraph of T'. If the representatives of a vertex u of G belong to different clusters, u is called a *shared vertex*. An edge is *incident* to a cluster if exactly one of its endpoints is in the cluster. An edge is *internal* if both endpoints are in the cluster. Let (x, y) be a tree edge incident to a cluster C and let $x \in C$. Then x is called a *boundary node* of C. The *tree degree* of a cluster is the number of tree edges incident to the cluster. Let $|C|$ denote the number of nodes in a cluster.

A *relaxed partition of order k* with respect to T' is a partition of the vertices into clusters so that

(C1) each cluster contains at most $k + 2m/k$ vertices of G';
(C2) each cluster has tree degree at most 3;
(C3) each cluster with tree degree 3 has cardinality 1;
(C4) if a cluster contains a shared vertex, then all boundary nodes are representatives of this shared vertex;
(C5) there are $O(m/k)$ many clusters; and
(C6) at the beginning of the phase if a cluster contains a shared vertex s, then every non-tree edge incident to the cluster either is adjacent to a representative of the shared vertex or is incident to another cluster sharing s.

This definition is an extension of [6]. We denote the cluster containing a node u' of G' by $C_{u'}$.

If all representatives of a vertex u of G belong to the same cluster C, we denote C by C_u and say that C *contains* u and u *belongs to* C. For a cluster C, let $V(C)$ denote the set of origins in G of the nodes of G' that (1) either belong to C or (2) are connected to a node in C by a solid tree edge.[2]

Each cluster containing a representative of a shared vertex u is called a cluster *sharing* u or *u-cluster*. Condition (C4) implies that each cluster shares at most one vertex. Together with condition (C5), it follows that there are $O(m/k)$ shared vertices.

[2]The origin of a node s' that is connected by a dashed edge to a node t' in C belongs to $V(C)$ since the origin of s' equals the origin of t', which belongs to $V(C)$ according to (1). Thus set C_T of [11] equals $V(C)$.

Data structure:

(G4) Each cluster keeps (a) a doubly linked list of all its vertices, (b) a doubly linked list of all its incident tree edges, and (c) a pointer to its shared vertex (if it exists).

(G5) Each nonshared vertex of G' keeps a pointer to the cluster it belongs to. Each shared vertex keeps a doubly linked list of all the clusters that share the vertex.

We repeatedly make use of the following fact.

FACT 2.1. *Let G be an n-node graph and let $u_1, \ldots, u_a$ be a set of articulation points that lie on a (simple) path in G. Then $\sum_i deg(u_i) \leq 2n$.*

To guarantee that conditions (C1)–(C4) are maintained within a phase any cluster violating the conditions is split into two clusters (see section 2.2). We show below that all these splits create only $O(m/k)$ new clusters, i.e., condition (C5) is always fulfilled.

2.2. Maintaining a relaxed partition of order k. To maintain a relaxed partition during updates, we create a more restricted partition at each rebuild, i.e., at the beginning of each phase, and let it gradually "deteriorate" during updates. Specifically, a cluster might be split but two clusters will never be merged. This implies that if a vertex becomes shared at some point in a phase it stays shared until the end of the phase.

A *restricted partition of order k* with respect to T' is a partition of the vertices into clusters so that

(C1′) each cluster has cardinality at most k;

(C2′) each cluster has tree degree ≤ 3;

(C3′) each cluster with tree degree 3 has cardinality 1;

(C4′) if a cluster contains a shared vertex, then all boundary nodes are representatives of this shared vertex;

(C5′) there are $O(m/k)$ clusters; and

(C6′) if a cluster contains a shared vertex s, then every non-tree edge incident to the cluster is either adjacent to the representative of the shared vertex or is incident to another cluster sharing s.

LEMMA 2.2. *A partition fulfilling* (C1′)–(C5′) *can be found in linear time.*

Proof. The algorithm in [6] shows how to find a partition fulfilling (C1′)–(C3′) and (C5′) in time $O(m + n)$.

Enforcing (C4′) *for a cluster in time linear in its size:* Each cluster that does not fulfill (C4′) has tree degree 2. Let x' and y' be the two boundary nodes in the cluster. Since x' and y' represent different shared vertices, the tree path between x' and y' contains at least one solid edge. Splitting the cluster at the solid edge on $\pi(x', y')$ closest to x' and at the solid edge on $\pi(x', y')$ closest to y' creates at most three clusters that fulfill conditions (C1′)–(C4′). Each split takes time linear in the size of the cluster. The lemma follows. □

LEMMA 2.3. *By modifying the spanning tree T' a partition fulfilling* (C1′)–(C5′) *can be modified in time $O(m \log n)$ to additionally fulfill* (C6′).

Proof. We need to enforce that if a cluster contains a shared vertex s, then every non-tree edge incident to the cluster is either adjacent to a representative of the shared vertex or is incident to another cluster sharing s. The main idea of our approach is to make every edge that violates this condition a tree edge, i.e., to remove a subtree of T' connected to the violating edge from the cluster sharing s.

To be precise, mark all vertices in clusters sharing s and test every edge incident

to such a vertex whether it is adjacent either to a shared vertex or to a marked vertex. If x' is in an s-cluster and the edge (x', y') does not fulfill the above condition, then determine the representative s' of s that is closest to x' in T' and the tree edge e incident to s' on $\pi(s', x')$. Make e a non-tree edge and make (x', y') a tree edge. Remove the subtree $\tilde{T}$ of $T' \setminus e$ containing x' from the s-cluster and add it to the cluster of y' if the resulting cluster $C_{y'}$ does not violate (C1′). Otherwise, create a cluster containing $\tilde{T}$. This increases the tree degree of $C_{y'}$ by 1. Since $C_{y'}$ contains y', a vertex which had an adjacent non-tree edge, it follows from (C3′) that $C_{y'}$ has now tree degree at most 3. If it has degree 3 and consists of more than one vertex, $C_{y'}$ is split as follows.

Let v', y', and z' be the three (not necessarily distinct) boundary nodes of $C_{y'}$. There exists a tree degree-3 node w' that belongs to $\pi(v', y')$, $\pi(v', z')$, and $\pi(y', z')$. The algorithm splits $C_{y'}$ at w' by creating a tree degree-3 cluster for w' alone and up to three additional tree degree-2 clusters, namely, if $v' \neq w'$, a cluster containing v', if $y' \neq w'$, a cluster containing y', and if $z' \neq w'$, a cluster containing z'. The new clusters fulfill (C1′)–(C3′). It is possible that (a) w' was not a shared vertex before the split, but is a shared vertex after the split, and that (b) $C_{y'}$ shared a vertex u' different from w' before the split. If (a) and (b) hold, then one of the new tree degree-2 clusters might violate condition (C4′). Split these clusters in the same way as described in the proof of Lemma 2.2, namely, at the solid edge on $\pi(w', u')$ closest to w' and at the solid edge on $\pi(w', u')$ closest to u'. This creates at most three clusters that fulfill conditions (C1′)–(C4′).

Note that a constant number of clusters was formed from the vertices in $C_{y'}$ and the vertices in $\tilde{T}$ and that there are more than k vertices in $C_{y'}$ and $\tilde{T}$ combined. Thus the total number of clusters when (C6′) holds for all clusters is still $O(m/k)$, i.e., (C5′) continues to hold.

To implement the above algorithm in time $O(m \log n)$ we use the following data structure to represent T', G', and the current cluster partitioning.

(1) We store a list of clusters and for each cluster we store its boundary vertices, its incident tree edges, and its shared vertex if it exists. For each shared vertex s we keep a list of all s-clusters.

(2) Each vertex of G' stores a doubly linked list of all incident edges, separated by tree and non-tree edges. Each edge points to its two positions in these lists.

(3) We also assume data structure (G1) exists already. This data structure will not be modified by this algorithm.

(4) Each vertex of G stores a bit indicating whether it is shared or not.

(5) We build in linear time a degree-k ET-tree data structure [12]. By removing the tree edges incident to a cluster, determining the size of the resulting spanning tree inside the cluster, and adding the tree edges again this data structure allows us to determine the number of vertices of a cluster in time $O(\log n)$.

(6) We build two dynamic tree data structures [18]. We mark in linear time each edge of T' in one of the dynamic tree data structures (G3) by a weight which is 1 for solid edges and 0 for dashed edges. These data structures are updated in time $O(\log n)$ after each edge insertion or deletion in T'. We use the second dynamic tree data structure to mark or unmark in time $O(\log n)$ all the vertices on an arbitrary path and to determine in time $O(\log n)$ the marked vertex on a (second) path closest to one of the endpoints of the path.

This data structure can be built in time $O(m)$.

The algorithm checks each cluster on the list of clusters to determine whether it

has a shared vertex s and, if so, it traverses and marks all vertices in the s-clusters starting at a boundary vertex. This takes time linear in the number of these vertices. Afterward it traverses them a second time and tests in constant time each adjacent non-tree edge whether it is incident to a marked vertex or a representative of a shared vertex. To test whether a vertex u' is the representative of a shared vertex u of G we determine the vertex u of G which u' represents and test the bit at u. This takes constant time.

We next show how to deal efficiently with an edge that violates the condition. Making a tree edge a non-tree edge or vice versa takes time $O(\log n)$. Determining the size of a cluster takes time $O(\log n)$ as discussed above. We still need to show how to determine s', e, w' and the solid tree edges closest to w' and closest to u' on $\pi(w', u')$ in time $O(\log n)$.

Determining s' and e: Mark in the second dynamic tree data structure all the representatives of s using (G1). Root the dynamic tree data structure at a vertex not in an s-cluster, for example, at y', and determine the marked vertex on $\pi(x', y')$ closest to x'. This vertex is s'. Then root the dynamic tree at x' and determine the edge from s' to its parent. This is the edge e. Finally unmark the marked vertices.

Determining w': Mark in the second dynamic tree data structure all the vertices on $\pi(v', z')$. Root the (updated) dynamic tree data structure at v' and determine the marked vertex on $\pi(y', v')$ closest to y'. This vertex is w'. Unmark the marked vertices.

Determining the solid tree edges closest to w' (resp., u') on $\pi(w', u')$: Root the first dynamic tree data structure at u' (resp., w') and determine the edge with weight 1 closest to w' (resp., u'). This is the desired edge.

Thus, for each non-tree edge that violates the condition we spend time $O(\log n)$ to make it a tree edge and to restore conditions (C1)–(C4). Note that if a tree edge becomes a non-tree edge, it stays a non-tree edge since it is adjacent to a shared vertex. Thus the cost of $O(\log n)$ is incurred $O(m)$ times, for a total time of $O(m \log n)$. ∎

We discuss next how to maintain a relaxed partition during updates. We show that an update does not violate condition (C1), (C4), or (C5). Obviously condition (C6) can never be violated since it only has to hold at the beginning of a phase. Condition (C2) or (C3) might be violated but can be restored by splitting a constant number of clusters. Restoring (C3) might lead to a violation of (C4), which can also be restored with an additional constant number of cluster splits.

We use the following *update algorithm for the relaxed partition*: If an insert(u, v) operation replaces u by two nodes, add both to C_u. If only one new representative $u_{num(u)+1}$ is created, add it to the cluster of $u_{num(u)}$ and increment $num(u)$ afterward. A deletion does not remove any vertices.

LEMMA 2.4. *The update algorithm for the relaxed partition does not violate condition* (C1), (C4), *or* (C5). *Conditions* (C2) *and* (C3) *might be violated for the clusters containing the endpoints of the newly inserted edge (in the case of an insertion) or the endpoints of the new tree edge (in the case of a deletion).*

Proof. Condition (C1)*:* An update increases the number of nodes in a cluster by at most two, implying that at the end of the phase each cluster contains at most $k + 2m/k$ nodes. It follows that condition (C1) is never violated.

Condition (C4)*:* Every newly added dashed edge has both endpoints in the same cluster, i.e., it is *not* incident to a cluster. Thus, condition (C4) is not violated.

Condition (C5)*:* A delete(u,v) operation might disconnect the connected component of C_u if $C_u = C_v$, leading to one additional cluster. Since there are m/k updates

in a phase, there exist $O(m/k)$ clusters during a phase, i.e., condition (C5) is not violated by the update algorithm.

Conditions (C2) *and* (C3)*: Deletions:* Note first that the deletion of a non-tree edge does not invalidate condition (C2) or (C3) and, thus, does not require any cluster splits. A deletion of a tree edge might make a (solid) non-tree edge into a tree edge, and, if this edge is an intercluster edge, add one new incident tree edge to its endpoint clusters. This might lead to a violation of condition (C2) or (C3) for the clusters incident to the new tree edge. *Insertions:* In the case that G is disconnected, a newly inserted edge might become a tree edge, adding one new (solid) tree edge to at most two clusters. As before, this might lead to a violation of conditions (C2) and (C3) for the clusters incident to the new tree edge. Additionally, an insertion might increase the number of nodes in a tree degree-3 cluster to two, violating condition (C3).

In conclusion, an update violates conditions (C2) and/or (C3) for at most two clusters, namely, the clusters containing the endpoints of the newly inserted edge or of the new tree edge. □

The algorithm first restores condition (C2) and then condition (C3). However, restoring (C3) might lead to the violation of condition (C4). If this happens, condition (C4) is restored after condition (C3).

Restoring condition (C2)*:* If a cluster violates (C2), it has tree degree 4 and consists of exactly two tree degree-3 nodes. Splitting it into two clusters creates two connected 1-node clusters of degree 3, fulfilling conditions (C1)–(C4).

Restoring condition (C3)*:* If a cluster C violates condition (C3), let x', y', and z' be the three (not necessarily distinct) boundary nodes of C. There exists a tree degree-3 node w' that belongs to $\pi(x', y')$, $\pi(x', z')$, and $\pi(y', z')$. The algorithm splits C at w' by creating a tree degree-3 cluster for w' alone and up to three additional tree degree-2 clusters, namely, if $x' \neq w'$, a cluster containing x', if $y' \neq w'$, a cluster containing y', and if $z' \neq w'$, a cluster containing z'. It is possible that (a) w' was not a shared vertex before the split, but is a shared vertex after the split, and that (b) C was incident to up to two dashed edges before the update, i.e., shared a vertex different from w'. If (a) and (b) hold, then one of the new tree degree-2 clusters might violate condition (C4). Splitting these clusters in the same way as in the proof of Lemma 2.2 creates at most three additional tree degree-2 clusters, each fulfilling conditions (C1)–(C4).

Thus, restoring conditions (C1)–(C4) requires creating a constant number of additional clusters after an update operation. Since there are m/k updates in a phase, condition (C5) is fulfilled at any point in a phase.

We summarize this discussion in the following lemma.

LEMMA 2.5.

(1) *An insertion does not split any cluster, but restoring the relaxed partition after an insertion might require a constant number of cluster splits, namely, of the clusters that contain the endpoints of the inserted edge.*

(2) *A deletion of a non-tree edge does not require any cluster splits.*

(3) *A deletion of a tree edge might split the cluster containing the endpoints of the deleted edge. Additionally, restoring the relaxed partition after a deletion might require a constant number of cluster splits, namely, of the clusters that contain the endpoints of the new tree edge.*

Testing whether a cluster violates (C2) or (C3) takes constant time; splitting a cluster takes time linear in its size. Thus, it takes time $O(k)$ to update the relaxed

partition of order k after each update.

2.3. Queries. Mapping G to G' causes correctness problems: If two nodes u and v are biconnected in G, they are also biconnected in G', but the reverse statement does *not* always hold (see [11] for an example).

The following lemma (an extension of Lemma 2.2 of [11]) relates the biconnectivity properties of G and of G'. To *contract* an edge (u, v) identify u and v and remove (u, v). To *contract* a vertex of G contract all dashed edges in G' belonging to the vertex.

LEMMA 2.6. *Let u and v be two vertices of G.*

(1) *Let G_1 be the graph that results from G' by contracting every vertex on $\pi_T(u, v)$ (excluding u and v). The vertices u and v are biconnected in G iff u'_v and v'_u are biconnected in G_1.*

(2) *Let y be a node on $\pi_T(u, v)$ that does not separate u and v in G. Let G_2 be the graph that results from G' by contracting every vertex on $\pi_T(u, v)$, excluding y, u, and v. The vertices u and v are biconnected in G iff u'_v and v'_u are biconnected in G_2.*

Proof. The graphs G_1, resp., G_2, can be created from G by expanding appropriate vertices. Thus, if u and v are biconnected in G, then u'_v and v'_u are biconnected in G_1, resp., G_2.

For the other direction, assume that u and v are separated by an articulation point x in G. Then x belongs to $\pi_T(u, v)$. It follows that x is represented by one node in G_1, resp., G_2. Assume by contradiction that u'_v and v'_u are biconnected in G_1, resp., G_2, i.e., there exists a path P' between them not containing x. The corresponding path P in G connects u and v and does not contain x, which leads to a contradiction. □

Note that the lemma also holds if additional vertices of G_1, resp., G_2, are contracted.

To test the biconnectivity of u and v in G we decompose the problem into subproblems, such that each subproblem is either (a) a biconnectivity query in a graph of size $O(k + m/k)$ or (b) a connectivity query in a graph of size $O(m)$. Subproblems of type (a) can be solved efficiently since $k + m/k$ is chosen to be "small." For subproblems of type (b) we use the existing efficient data structures for maintaining connectivity dynamically. Since no data structure is known that solves both subproblems efficiently, we maintain two different data structures, called *cluster graphs* and *shared graphs*.

To be precise, for each shared vertex s, a *shared graph* is maintained. Given a shared vertex s and two of its tree neighbors x and y, the shared graph of s is used to test in constant time whether s is an articulation point separating x and y. This is equivalent to testing if x and y are disconnected in $G \setminus s$. Thus, we use the dynamic connectivity data structure [12] to maintain the shared graphs.

Recall that $V(C)$ for a cluster C denotes the set of origins in G of the nodes of G' that either (1) belong to C or (2) are connected to a node in C by a solid tree edge. We maintain for C a *cluster graph* which is built to test (in constant time) if any two nodes of $V(C)$ that are not separated by s_C are biconnected in G, where s_C is the shared vertex of C. In particular, the cluster graph can be used to test whether s_C is biconnected with another node in C. To maintain the cluster graphs we use the data structure of [11]. To test if two nodes x and y are biconnected in G, the data structure contracts in G' all vertices on $\pi_T(x, y)$ (and potentially additional vertices).

We describe next how we use these two data structures to answer a biconnectivity query. We use the following lemma.

LEMMA 2.7. *Let u and v be nodes of G and let $(x^{(i)'}, y^{(i)'})$, for $1 \le i \le p$, denote*

the solid intercluster tree edges on $\pi_{T'}(u'_v, v'_u)$, in the order of their occurrence. Then u and v are biconnected in G iff

(Q1) *u and $y^{(1)}$ are biconnected in G,*

(Q2) *$x^{(i)}$ and $y^{(i+1)}$ are biconnected in G, for $1 \leq i < p$, and*

(Q3) *$x^{(p)}$ and v are biconnected in G.*

Proof. If u and v are biconnected, then all nodes on $\pi(u, v)$ are pairwise biconnected, and thus (Q1)–(Q3) hold. If u and v are not biconnected, then u and v are separated by an articulation point z. Since z belongs to $\pi_T(u, v)$, either (Q1), (Q2), or (Q3) is violated. This is a contradiction. □

For a query(u, v), let C be the cluster of u'_v. If C shares a vertex, call it s. We test the conditions of the lemma using only a cluster graph if this is possible and using a cluster graph and a suitable shared graph otherwise.

Testing condition (Q1)*:* Condition (Q1) of Lemma 2.7 can be tested using two cluster graphs and the shared graph of s: if s does not lie on $\pi(u, y^{(1)})$, then s does not separate u and $y^{(1)}$, and $y^{(1)}$ belongs to $V(C)$. Thus, we use the cluster graph of C to test whether u and $y^{(1)}$ are biconnected.

If s lies on $\pi(u, y^{(1)})$, then let x_u and y_u be the nodes incident to s on $\pi(u, y^{(1)})$. Let s' be the node representing s closest to $y^{(1)}$ on $\pi(u, y^{(1)})$. Note that s belongs to $V(C)$ and that $y^{(1)}$ belongs to $V(C_{s'})$. Test in the cluster graph of C if u and s are biconnected, test in the cluster graph of $C_{s'}$ if s and $y^{(1)}$ are biconnected, and test in the shared graph of s if x_u and y_u are biconnected. If all tests are successful, u and $y^{(1)}$ are biconnected in G, since the last test guarantees that s does not separate u and $y^{(1)}$ and the first two tests guarantee that no other node of $\pi(u, y^{(1)})$ separates u and $y^{(1)}$.

Testing condition (Q2)*:* If $y^{(i)'}$ and $x^{(i+1)'}$ belong to the same cluster C_i and C_i does not share a vertex, then both, $x^{(i)}$ and $y^{(i+1)}$, belong to $V(C_i)$ and are not separated by a shared vertex of C_i. Thus, condition (Q2) can be tested using the cluster graph of C_i.

We show that otherwise $x^{(i)}$ and $y^{(i+1)}$ are tree neighbors of a shared vertex s_i and the shared graph of s_i can be used to test condition (Q2): either (a) $y^{(i)'}$ and $x^{(i+1)'}$ belong to the same cluster C_i that shares a vertex s_i, or (b) $y^{(i)'}$ and $x^{(i+1)'}$ belong to different clusters. In case (a), C_i is incident to two solid tree edges and one dashed tree edge, i.e., C_i has tree degree 3. By condition (C3) it follows that C_i contains only one node, i.e., $y^{(i)'} = x^{(i+1)'}$, and both are representatives of s_i. It follows that $x^{(i)}$ and $y^{(i+1)}$ are both tree neighbors of s_i. In case (b), all intercluster edges between the cluster of $y^{(i)'}$ and the cluster of $x^{(i+1)'}$ are dashed. By condition (C4) of a relaxed partition, all these dashed edges and also $y^{(i)'}$ and $x^{(i+1)'}$ belong to the same shared vertex s_i. Thus, also in this case $x^{(i)}$ and $y^{(i+1)}$ are tree neighbors of s_i. It follows that the shared graph of s_i can be used to test whether $x^{(i)}$ and $y^{(i+1)}$ are biconnected in G.

Testing condition (Q3)*:* Condition (Q3) is tested analogously to condition (Q1).

Since each test takes constant time, this leads to a query algorithm whose running time is linear in the number of solid intercluster edges on $\pi_{T'}(u'_v, v'_u)$, which is $O(m/k)$. However, we will give in section 2.11 a data structure that allows all these tests to be executed in constant time.

Our next goal is to describe *cluster graphs* and *shared graphs* in detail. Maintaining them requires a third data structure, called *high-level graphs*, which we describe first.

2.4. Overview of high-level graphs. There are two *high-level graphs*, H_1 and H_2. Basically, H_1 is a graph where each cluster is contracted to one node, and H_2 is

a copy of H_1 with intercluster dashed edges contracted as well.

To be precise, the graph H_1 contains a node for each cluster of G'. Two nodes C and C' of H_1 are connected by an edge in H_1 iff there is an edge between a vertex of C and a vertex of C'. If there exists an edge between C and C', we call C' the *neighbor* of C. The edge between C and C' in H_1 is a dashed edge iff there is a dashed edge between a vertex of C and a vertex of C'. Otherwise the edge in H_1 is *solid*.

The graph H_2 is the graph H_1 with all dashed edges of H_1 contracted.

We call vertices of H_1 or H_2 *nodes* and refer to vertices of G of G' as *vertices*.

Since each node of H_1 represents exactly one cluster, we will use the terms *node of* H_1 and *cluster* interchangeably. Each node of H_2 represents at least one cluster and it also represents the vertices of G with a representative in these clusters. Note that each vertex of G is represented by a unique node of H_2, while this does not hold for H_1: a shared vertex is represented by more than one node of H_1.

The spanning tree T' of G' induces a spanning tree T_1 on H_1 and T_2 on H_2. We say C' is a *tree neighbor* of C if there is a tree edge between C' and C in H_1. Otherwise C' is a *non-tree neighbor*.

We need the high-level graphs to define and maintain the cluster graphs and the shared graphs. Roughly speaking, a cluster graph tests (under certain conditions) whether two vertices represented by the same node of H_1 are biconnected in G, and a shared graph tests whether two vertices represented by the same node of H_2 are biconnected in G.

When maintaining cluster and shared graphs we make use of the following data structures. Details of some of these data structures are delayed until section 2.7. Let $i = 1, 2$.

(HL1) We store for each node of H_i all the vertices of G' belonging to the node,[3] and we store at each vertex of G' the node of H_i to which the vertex belongs.

(HL2) We keep the following adjacency list representation for H_i (of size $O((m/k)^2) = O(m)$): For each node of H_i we keep the list of all incident neighbors and a list of pointers to all of its positions in the lists of its neighbors. Thus, in constant time an edge between two nodes can be removed from this representation.

(HL3) We maintain a data structure that given two nodes C and C' returns the tree neighbor C'' of C such that C'' lies on $\pi_{T_i}(C, C')$. For H_1 this takes constant time; for H_2 it takes time $O(\log n)$.

(HL4) We store a data structure that implements the following query operations in H_i:

biconnected?(C,C′,C″): Given that nodes C' and C'' are both tree neighbors of a node C, test whether C' and C'' are biconnected in H_i.

blockid?(C,C′): Given that C and C' are tree neighbors in T_i, output the name of the biconnected component of H_i that contains both C and C'.

components?(C): Output the tree neighbors of C in H_i grouped into biconnected components.

Operations *biconnected?* and *blockid?* take constant time, and *components?* takes time linear in the size of the output.

(HL5) We keep a *mapping* h from H_1 to H_2 and a mapping h^{-1} from H_2 to H_1. For each node of H_2 we keep a list of pointers to all the nodes of H_1 whose contraction formed H_1, and for each node of H_1 we keep a pointer back to the corresponding node of H_2.

[3]For H_1 this is already part of (G4) and, of course, does not need to be stored twice.

(HL6) We keep an empty array of size $O(m/k)$ at each node in H_i (needed for various bucket sorts—see sections 2.5 and 2.6).

Note that using (HL3) and (HL4) one can test in constant time whether two neighbors of a node C in H_i are biconnected in H_i.

Recall that after an update operation clusters might be split to restore the relaxed partition.

LEMMA 2.8. *If a cluster C is split while restoring the relaxed partition, then the clusters containing the vertices of C when the relaxed partition is restored form a connected subgraph of H_1.*

Proof. Splitting simply regroups the partitioning of vertices into clusters but does not change the connectivity properties in G. Since the vertices in C form a connected subgraph of G before the split, they also do so after the split. Hence, the clusters containing these vertices form a connected subgraph of H_1. □

Next we need to assign "ancestors" to nodes and edges in H_1 and H_2. This is necessary for our lazy update scheme: if a node is split into two nodes, then the resulting nodes have the same ancestor. However, we cannot afford to update all cluster and shared graphs accordingly. Therefore, we will treat nodes with the same ancestor that fulfills certain additional conditions as one node in some of the cluster and shared graphs.

First we define a unique *ancestor* for each node in H_2. Let C be a node in the current graph H_2. Since clusters are only split, never joined, all nodes represented by C were represented by the same cluster A at the beginning of a phase. This node A is called the *ancestor* of C.

To define ancestors for nodes in H_1, we denote by B_s the cluster that contains $s_{num(s)}$ at the beginning of a phase for each shared vertex s. Note that if a cluster contains only representatives that were created after the last rebuild, then these representatives represent a shared vertex. Consider a cluster C of H_1. If C only contains representatives that were created after the last rebuild, then the ancestor of C is B_s. Otherwise, C has at least one representative that existed at the last rebuild. In this case, the representatives that existed at the last rebuild belonged to the same cluster A at the last rebuild. This node A is called the *ancestor* of C.

LEMMA 2.9. *Let $i = 1$ or 2. Assume a node C of H_i is split into clusters C_1 and C_2. Then C_1 and C_2 have the same ancestor as C.*

Proof. For $i = 2$ this follows immediately from the definition. For $i = 1$, we only give the argument for C_1; the same argument applies to C_2. If C_1 contains representatives that existed at the last rebuild, then these representatives all belonged to the same cluster at the last rebuild. This cluster is also the ancestor of C. Otherwise, C_1 contains only representatives that were created after the last rebuild. In this case the ancestor of C_1 is B_s. We show below inductively that representatives that are created after the last rebuild are added into clusters whose ancestor is B_s. Thus, the ancestor of C is B_s as well.

When a new representative is added for s, it is added to the cluster containing $s_{num(s)}$. We show inductively that the cluster containing $s_{num(s)}$ for the current value of $num(s)$ has B_s as ancestor. The induction goes over the number of representatives of s added to G' after the last rebuild. By the definition of ancestor for clusters with representatives that existed at the last rebuild the claim holds before a new representative was added for s. Consider next the addition of the ith new representative $s_{num(s)+1}$ of s. If the cluster containing $s_{num(s)}$ is not split, then the claim holds inductively. Otherwise two situations can arise: either the cluster C containing $s_{num(s)+1}$

after the split contains only representatives that were created after the last rebuild, or not. In the former case, B_s is the ancestor of C by definition. In the latter case, C contains representatives that belonged to B_s at the last rebuild. Thus, in either case, B_s is the ancestor of C. □

In particular, C_1 and C_2 have the same ancestor and each node has a unique ancestor.

Data structure:

(HL7) We store at each node of H_i its ancestor.

(HL8) We number the edges of H_i in the order in which they were added to H_i with the edges added during a rebuild in arbitrary order.

Recall that at the beginning of each phase a cluster contains at most k vertices of G'. This enables us to prove the following lemma.

LEMMA 2.10. *The total number of vertices of G' in all clusters with the same ancestor is $k + 2m/k$. The total number of edges incident to these vertices is $O(k)$.*

Proof. Let A be a cluster at the beginning of a phase. All nodes of a cluster with ancestor A (in G') that existed at the time of the last rebuild belonged to A at the beginning of the phase. Thus, there are at most k of them. Every other node was created by one of the m/k update operations. Since each update creates at most two new vertices, the bound follows. □

2.5. Cluster graphs. Let C be a cluster. The *cluster graph* $I(C)$ of C is used to test if two vertices u and v of $V(C)$ that are not separated by s_C are biconnected in G. This leads to a first requirement for $I(C)$:

(IC1) *If s_C does not separate u and v, then u and v are biconnected in $I(C)$ iff they are biconnected in G.*

As we see below, an amortized constant number of cluster graphs is rebuilt during each update operation. This leads to a second requirement for $I(C)$:

(IC2) *The graph $I(C)$ has size $O(|C|) = O(k)$.*

Recall that $V(C)$ denotes the set of origins in G of the nodes of G' that (1) either belong to C or (2) are connected to a node in C by a solid tree edge.

We next motivate our definition of $I(C)$ and explain why it can be used only if s_C does not separate the two nodes. Obviously, $I(C)$ has to contain all nodes of $V(C)$ and all edges between the nodes of $V(C)$. Since two nodes of $V(C)$ can be connected by a path in $V(C)$ and additionally by a path that contains nodes of a non-tree neighbor of C, we represent each non-tree neighbor of C by a node in $I(C)$ (called either *b-node* or *c-node*) and we add to $I(C)$ all edges incident to C.

However, three questions remain: (1) If C shares a vertex s_C, let C' be one of the neighbors of C connected to C by a dashed tree edge (belonging to s_C). Should $I(C)$ also contain a node representing C', i.e., should s_C and C' be represented by the same or different nodes in $I(C)$? (2) How is the set of b- and c-nodes connected by edges? (3) How can the graph be maintained efficiently when a neighbor of C is split?

Next we describe our solution to these questions. (1) If s_C and C' are represented by the same node, then two nodes u and v of $V(C)$ that are biconnected in G might not be biconnected in $I(C)$. See Figure 2.1 for an example. On the other side, if $I(C)$ contains a node for s_C and a separate node for C', then two vertices u and v of $V(C)$ that are not biconnected in G can be biconnected in $I(C)$. See Figure 2.2 for an example.

However, by Lemma 2.6 and the fact that in the latter approach all nodes on $\pi_T(u, v)$ except for s_C are contracted, it follows that the latter situation can happen

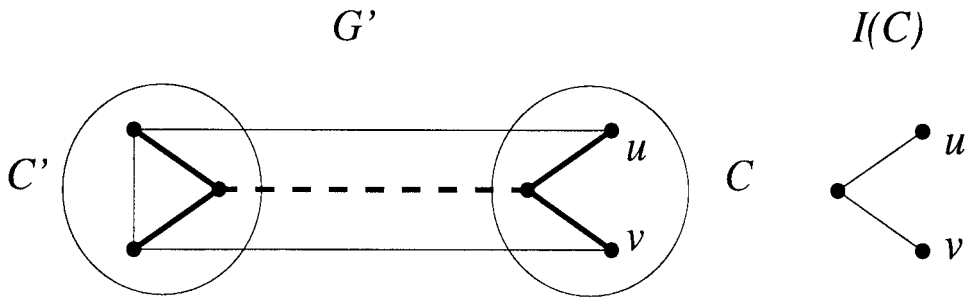

FIG. 2.1. *The graph G' and a potential graph $I(C)$. The graph G' consists of cluster C and C' (represented by circles), both sharing vertex s (represented by two nodes and the dashed line between them). Tree edges are bold or dashed. In $I(C)$, C' and s are collapsed to one node. Nodes u and v are biconnected in G but not in $I(C)$.*

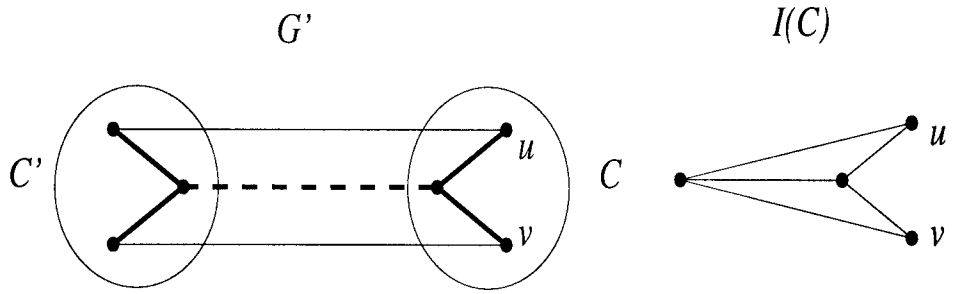

FIG. 2.2. *The graph G' and a potential graph $I(C)$. The graph G' consists of cluster C and C' (represented by circles), both sharing vertex s (represented by two nodes and the dashed line between them). Tree edges are bold or dashed. In $I(C)$, C' and s are represented by two different nodes. Nodes u and v are not biconnected in G' but are biconnected in $I(C)$.*

only if s_C separates u and v in G. Since this case is excluded by (IC1), we represent s_C and C' by separate nodes in $I(C)$.

(2) Let j be the number of neighbors of C. There are at most j b- or c-nodes in $I(C)$. Since G' is a graph of degree at most 3, $j = O(|C|)$. To guarantee that $I(C)$ has size $O(|C|)$, $I(C)$ will contain at most $j - 1$ many edges between b- or c-nodes. These edges will be colored and will fulfill the condition that two b- or c-nodes are connected by a path of colored edges iff they are connected in $H_1 \setminus C$.

(3) We will split a node representing a neighbor C' of C only if the two clusters resulting from the split of C' are disconnected in $H_1 \setminus C$. Otherwise, both resulting clusters will be represented by the same c-node in $I(C)$, i.e., a c-node in the cluster graph might represent not just one cluster, but a set of clusters. This leads to the following invariant: *Two neighbors of C are represented by the same node in $I(C)$ or are connected by a colored path iff they are connected in $H_1 \setminus C$.*

Let us now give the exact definition of a *cluster graph* $I(C)$ for a cluster C. Let A be the ancestor of C. The cluster graph contains as nodes

(1) a node, called *a-node*, for each vertex with a representative in C,

(2) one node, called *b-node*, for each neighbor C' of C with ancestor A,

(3) one node, called *c-node*, for each maximal set X of clusters such that (a) every cluster $C' \in X$ is a neighbor of C, (b) all clusters in the set are connected in

$H_1 \setminus C$, (c) all clusters in X have the same ancestor which is different from A, (d) at the creation of C, the set X contains only one cluster, and (e) at each previous point in time since the creation of C all clusters that contain the vertices in $\cup_{C' \in X} C'$ existing at this time are represented by the same c-node in $I(C)$.

Note that for each neighbor C' of C there exists a unique node in $I(C)$ representing C' (and potentially other clusters). Note further that each node of G is represented by at most one node in $I(C)$, except for s_C, which can be represented by an a-node and up to two b- or c-nodes, namely, the tree neighbors of C that share s_C.

The graph $I(C)$ contains the following edges:

(1) All edges between two vertices of G represented by an a-node belong to $I(C)$.

(2) For each edge (u, v) where u is represented by an a-node and v is not, and (u', v') is the corresponding edge in G', there is an edge (u, d) in $I(C)$, where d is the b- or c-node representing $C_{v'}$.

(3) For each tree neighbor C_1 connected to C by a dashed edge there is an edge from the b- or c-node of C_1 to the a-node of s_C.

(4) For each pair C_1 and C_2 of tree neighbors of C there is a *red* edge (d_1, d_2) if C_1 and C_2 are biconnected in H_1, where d_j is the b- or c-node representing C_j, $j = 1, 2$.

(5) For each non-tree neighbor C_1 of C with representative d_1,[4] $I(C)$ contains a *blue* edge (d_1, d_2), where d_2 represents the tree neighbor of C that lies on $\pi_{T_1}(C, C_1)$, if C is an articulation point in H_1 (separating its at most two tree neighbors), and a *blue* edge (d_1, d_3), where d_3 represents an arbitrary tree neighbor of C, otherwise.[5]

Note that $I(C)$ can contain parallel edges. They can be discarded without affecting the correctness.

We show next that the cluster graphs fulfill (IC1) and (IC2).

LEMMA 2.11. *Two neighbors of C are biconnected in H_1 iff either they are represented by the same node in $I(C)$ or their representatives in $I(C)$ are connected by a colored path.*

Proof. We show first that if there is a colored edge between two nodes d_1 and d_2 in $I(C)$, then the clusters that they represent are biconnected in H_1. For red edges this follows immediately from the definition. For a blue edge consider first the case that C is not an articulation point in H_1. In this case all neighbors of C are biconnected in H_1. Since each blue edge connects two neighbors of C, the claim holds. Consider next the case that C separates its tree neighbors C_1 and C_2 in H_1. Note that there are two vertex-disjoint paths in H_1 between C and each of its non-tree neighbors C_1: one path consists of the non-tree edge (C, C_1), and the other path is $\pi_{T_1}(C, C_1)$. Thus C_1 and C's tree neighbor on $\pi_{T_1}(C, C_1)$ are biconnected as well, i.e., the claim holds for each blue edge.

This implies that if the representative of two neighbors of C are connected by a colored path in $I(C)$, then the neighbors of C are biconnected in H_1. If two neighbors are represented by the same node in $I(C)$, then they are biconnected in H_1 by the definition of a c-node.

Next we show that if two neighbors of C are biconnected in H_1 and represented by two different nodes in $I(C)$, then these representatives in $I(C)$ are connected by a colored path. Each non-tree neighbor C_1 is biconnected with the tree neighbor of C

[4] If C has a non-tree neighbor, then C has tree degree at most 2.

[5] Note that C_1 and C are biconnected in H_1. Thus this is equivalent to requiring that for each non-tree neighbor C_1 of C there exists a blue edge (d_1, d_2), where d_2 represents a tree neighbor of C that is biconnected to C_1 in H_1 and d_1 represents C_1.

on $\pi_{T_1}(C, C_1)$ and is connected to this tree neighbor by a colored path (of length at most two). Thus it suffices to show the claim for two tree neighbors of C. But for two tree neighbors the claim holds by definition. □

LEMMA 2.12. *Let C be a cluster and let u and v be two nodes of $V(C)$. If s_C does not separate u and v in G, then u and v are biconnected in $I(C)$ iff they are biconnected in G.*

Proof. Assume first that u and v are biconnected in $I(C)$ but are separated by a node x in G. Note that x must belong to $V(C)$. Furthermore, x must be represented by at least two nodes in $I(C)$. However, each node of G is represented by at most one node in $I(C)$, except for s_C. Thus, $x = s_C$, which leads to a contradiction.

Assume next that u and v are biconnected in G, but are separated by a node y in $I(C)$. Since u and v are connected by a tree path whose (internal) nodes all belong to C, no b-node or c-node can separate u and v in $I(C)$. Thus, y must be an a-node.

Consider the path P between u and v in G that does not contain y. Let $\tilde{P}$ be the path created from P by (1) extending P to a path in G', (2) contracting all intracluster edges of P except for the nondashed intracluster edges of C, and (3) by labeling the resulting nodes of $\tilde{P}$ with their clusters of G'.

Every edge of $\tilde{P}$ either is connecting two clusters or is incident to a vertex with a representative in C. We show that $\tilde{P}$ induces a path without y in $I(C)$ connecting u and v. We split $\tilde{P}$ into subpaths. Each subpath either

(1) connects two neighbors of C and does not contain other neighbors of C or vertices with representatives in C, or

(2) is one edge connecting two vertices with representatives in C, or

(3) is a solid edge connecting a vertex with a representative in C with a neighbor of C, or

(4) is a dashed edge connecting a vertex with a representative in C with a neighbor of C.

By Lemma 2.11 the endpoints of the type-(1) subpaths are connected by a colored path in $I(C)$. By definition type-(2), (3), or (4) subpaths are contained in $I(C)$. Thus $\tilde{P}$ induces a path without y from u to v in $I(C)$. Thus we have a contradiction. □

LEMMA 2.13. *For each cluster C,*

$$|I(C)| = O(|C|).$$

Proof. Obviously, there are $O(|C|)$ a-nodes and $O(|C|)$ edges incident to them in $I(C)$. Each b-node or c-node in $I(C)$ can be charged to one of the edges that connects the b-node or c-node to a node in C. Thus, there are $O(|C|)$ b- or c-nodes. Since the number of colored edges is linear in the number of b- and c-nodes, it follows that $|I(C)| = O(|C|)$. □

We will need the following fact when bounding the time of updates.

LEMMA 2.14. *Let $C_1, \ldots, C_l$ be a set of clusters with the same ancestor. Then*

$$\sum_{i=1}^{l} |I(C_i)| = O(k).$$

Proof. By Lemma 2.13,

$$\sum_{i=1}^{l} |I(C_i)| = \sum_{i=1}^{l} O(|C_i|).$$

Let A be the ancestor of the clusters $C_1, \ldots, C_l$. Recall that either (1) each C_i contains the representative of a vertex and that representative or the (unexpanded) origin of the representative also belonged to A, or (2) C contains only representatives of the shared vertex s of A, all these representatives were created after the last rebuild, and $A = C_s$.

The number of clusters fulfilling (1) is bounded by the number of nodes of G' in A. The total number of clusters fulfilling (2) is bounded by the number of updates since the last rebuild, which is m/k.

Thus,

$$\sum_{i=1}^{l} O(|C_i|) \leq O(|\{v, v \text{ is a node of } G' \text{ in } A\}| + m/k) = O(k). \qquad \square$$

In [11][6] a *cluster data structure* for $I(C)$ is given so that

(1) building the data structure takes time $O(|C|)$, provided that the b-nodes, the c-nodes, and the red and blue edges are given;

(2) changing one or all of the colored edges takes time linear in their total number, provided the new colored edges are given;[7]

(3) testing whether two vertices of $V(C)$ that are not separated by s_C are biconnected in $I(C)$ takes constant time.

We keep as data structure

(CG1) for each cluster C a *cluster data structure* for $I(C)$;

(CG2) for each cluster C the adjacency lists of the graph $I(C)$ with the two occurrences of edges pointing at each other;

(CG3) for each cluster C, a list of pointers to the b- or c-nodes representing C in $I(C)'$ for each neighbor C'; for each b-node in $I(C')$, a pointer back to C, and for each c-node in $I(C')$, a set of pointers to the clusters represented by the c-node.

Note that given the b-nodes and c-nodes of $I(C)$, the red and blue edges of $I(C)$ can be determined in time linear in their number using the data structure (HL3) and (HL4) for H_1. This leads to the following lemma.

LEMMA 2.15. *Let C be a cluster. There exists a cluster data structure for $I(C)$ such that*

(1) *building the data structure takes time $O(|C|)$, provided that the b-nodes and the c-nodes are given;*

(2) *changing one or all of the colored edges takes time linear in their total number, given the b-nodes and c-nodes;*

(3) *testing whether two vertices of $V(C)$ that are not separated by s_C are biconnected in $I(C)$ takes constant time.*

Note: We will use the same data structure and the same update algorithm in section 2.6 for one class of shared graphs. There the same problem has to be solved in

[6] Lemma 4.6 of [11] states the result; section 4.1.2 of [11] describes the data structure. In the notation of [11], $G_3(C)$ is identical to $I(C)$ except that a non-tree neighbor C_1 of C *always* has a blue edge (C_1, C_2) to the tree neighbor C_2 of C that lies on $\pi_{T_1}(C, C_1)$, even if C is not an articulation point. Furthermore, $G_2(C) = G_3(C) \setminus \{\text{red edges}\}$, $C_T = V(C)$, and the *artificial edges* of [11] are identical to the colored edges of $I(C)$.

[7] In [11] changing a red edge actually takes no time during an update: the existence of a red edge is not recorded during an update but is checked during queries (by asking a biconnectivity query in H_1). This is possible, since only one red edge exists in a cluster graph. Since we will use the same data structure also for one class of shared graphs, we treat red edges as blue edges in the data structure of [11].

H_2 instead of H_1. Since nodes in H_2 are not guaranteed to have bounded tree degree, we will not make use of this property of H_1 in our update algorithm.

2.5.1. Updates. We show in this section that it takes amortized time $O(k)$ to update the data structures for all cluster graphs after an edge insertion or deletion in G. The major difficulty is to maintain the b-nodes and c-nodes of each cluster graph. Once it has been determined how they change, it will be quite straightforward to update data structures (CG1)–(CG3). Let C be a cluster. A c-node of $I(C)$ has to be partitioned either (i) because C is split or (ii) because conditions (a) or (b) of a c-node are no longer fulfilled for the c-node. We call the latter kind of update a *split by condition violation (CV-split)* of a c-node. We say a CV-split *occurs* at a node C of H_1 if one of the c-nodes of C is CV-split.

Section 2.8 presents a data structure that given an update operation decides which if any c-nodes have to be CV-split and for each CV-split c-node it returns the set of clusters forming each new c-node. In section 2.9 we show that during m/k update operations CV-splits occur at $O(m/k)$ different nodes of H_1. When a CV-split occurs at a node C of H_1 in a cluster graph, all data structures for the cluster graph of C are rebuilt from scratch in time $O(k)$. Thus the CV-splits add an amortized cost of $O(k)$ to the time per update. We describe below the remaining work that is necessary after an update to maintain the cluster graphs.

Insertion. Let u' and v' be the (potentially newly added) representatives that are incident to the newly inserted edge (u, v). The insertion affects the cluster graphs as follows:

(1) The clusters $C_{u'}$ and $C_{v'}$ might become neighbors because of the edge insertion and they also might be split while rebalancing the relaxed partition. In either case some b-nodes and/or c-nodes in $I(C_{u'})$ and $I(C_{v'})$ change. If this happens the data structures for $I(C_{u'})$ and $I(C_{v'})$ are rebuilt from scratch.

(2) For a cluster C' incident to a split cluster the b-node representing the split cluster might have to be partitioned into a constant number of b-nodes. If this happens the data structures for $I(C')$ are rebuilt from scratch.

(3) Nodes on $\pi_{T_1}(C_{u'}, C_{v'})$ that separated $C_{u'}$ and $C_{v'}$ in H_1 before the insertion no longer separate $C_{u'}$ and $C_{v'}$. For each such node, a suitable red edge is added to the cluster graph, without rebuilding the cluster graph from scratch.

We next discuss each case in detail. (1) If $C_{u'}$ and $C_{v'}$ become neighbors because of the edge insertion, then a new b- or c-node representing $C_{v'}$, resp., $C_{u'}$, has to be added to $I(C_{u'})$, resp., $I(C_{v'})$. Consider the split of a cluster C into a constant number of clusters C_j. Then each c-node in $I(C_j)$ represents only one cluster by part (d) of the definition of a c-node. Thus, for each neighbor of C_j simply test whether it has the same ancestor as C_j. If yes, it is represented by a b-node in $I(C_j)$, otherwise it is represented by a c-node.

(2) We describe next how the changes in the b-nodes of $I(C')$ are determined when cluster C is split into a constant number of nodes C_j. Bucketsort the edges incident to C in lexicographic order of its two endpoints in the updated graph H_1 (using (HL6)). Each neighbor of C with the same ancestor as C that is incident to edges from $l > 1$ different buckets (i.e., new clusters) of its array receives l new b-nodes representing these new clusters, discards the b-node of C, and keeps all the other old b-nodes.

(3) The articulation points on $\pi_{T_1}(C_{u'}, C_{v'})$ are found by testing in constant time each of the clusters on $\pi_{T_1}(C_{u'}, C_{v'})$ using data structure (HL4) for H_1 (before the update). Then a red edge between its tree neighbors on $\pi_{T_1}(C_{u'}, C_{v'})$ is added to each

articulation point.

We next show that this takes total time $O(k)$. In (1) it takes constant time to test whether a b-node or a c-node has to be added for the neighbor of a cluster. Each test takes constant time using (HL7). By Lemma 2.10 there are $O(k)$ many such tests. Building the data structure (CG1) for a constant number of clusters takes time $O(k)$ by Lemma 2.15; building (CG2) and (CG3) takes $O(k)$ as well.

In (2) there are $O(k)$ edges to bucketsort. If each bucket with a count larger than 1 is put in a separate list, then all new b-nodes can be determined in time $O(k)$, and (CG3) can be updated accordingly. By Lemma 2.10 the total size of all the cluster graphs rebuilt in (2) is $O(k)$. Thus building (CG1) and (CG2) for all of them takes total time $O(k)$ by Lemma 2.15.

Each test in (3) takes constant time, and adding the red edge for a former articulation point C' in H_1 takes time linear in the degree of C'. By Fact 2.1 the H_1-degree of all articulation points on $\pi_{T_1}(C_{u'}, C_{v'})$ sums to $O(m/k)$.

Deletion of a non-tree edge. The deletion of a non-tree edge does not change the spanning tree and does not split a cluster (Lemma 2.5). Thus, the deletion of a non-tree edge affects the cluster graph as follows:

(1) The clusters $C_{u'}$ and $C_{v'}$ might no longer be neighbors in H_1. In this case the corresponding b-node or c-node has to be removed from $I(C_{v'})$, resp., $I(C_{u'})$. If this happens the data structures for $I(C_{u'})$ and $I(C_{v'})$ are rebuilt from scratch.

(2) A red edge has to be removed and the blue edges have to be updated in the cluster graph of each new articulation point on $\pi_{T_1}(C_{u'}, C_{v'})$ in the updated graph H_1.

We next discuss each case in detail: in case (1) we simply need to test whether the edge $(C_{u'}, C_{v'})$ was removed from H_1. If so, and if $C_{u'}$ and $C_{v'}$ have the same ancestor, then the corresponding b-nodes are removed from $I(C_{v'})$ and $I(C_{u'})$. Otherwise, we need to test whether the c-node representing $C_{u'}$ in $I(C_{v'})$ represents further clusters. If not, then the c-node is removed. We proceed in the same way for the c-node representing $C_{v'}$ in $I(C_{u'})$.

In case (2) we determine each new articulation point C on $\pi_{T_1}(C_{u'}, C_{v'})$ by testing in constant time each of the clusters on $\pi_{T_1}(C_{u'}, C_{v'})$ using data structure (HL4) for H_1 (after the update). Then we remove all old red and blue edges from $I(C)$. Using (HL3) we determine for each neighbor C' of C the tree neighbor C'' of C such that C'' lies on $\pi_{T_1}(C, C')$, and connect C' by a blue edge with C''. Finally we determine the biconnected components of the tree neighbors of C using a *components?(C)* operation in (HL4) and add the suitable red edges.

We next show that this takes total time $O(k)$. In case (1) testing and rebuilding the data structures takes time $O(k)$ by the same argument as for insertions. In case (2) finding all articulation points takes time $O(m/k)$. Determining the new red and blue edges takes time linear in the degree of each articulation point. By Fact 2.1 the total cost for all articulation points is $O(m/k)$.

Deletion of a tree edge. Let u' and v' be the representatives that are incident to the deleted edge (u, v) and let x' and y' be the representatives that are incident to the new tree edge (x, y), if it exists. By Lemma 2.5, $C_{u'}$, $C_{v'}$, $C_{x'}$, and $C_{y'}$ are the only clusters that might be split.

The cluster graphs are affected as follows: (1) The clusters $C_{u'}$ and $C_{v'}$ might no longer be neighbors in H_1. In this case the corresponding b-node or c-node has to be removed from $I(C_{v'})$, resp., $I(C_{u'})$. If this happens the data structures for $I(C_{u'})$ and $I(C_{v'})$ are rebuilt from scratch.

(2) The clusters $C_{u'}$, $C_{v'}$, $C_{x'}$, and $C_{y'}$ might be split. The data structures (CG1)–(CG3) are rebuilt from scratch for each cluster created by the splits.

(3) For a cluster C' incident to a split cluster the b-node representing the split cluster might have to be partitioned into a constant number of b-nodes. If this happens the data structures for $I(C')$ are rebuilt from scratch.

(4) A red edge has to be removed and the blue edges have to be updated in the cluster graph of each new articulation point on $\pi_{T_1}(C_{x'}, C_{y'})$ in the updated graph H_1.

We implement (1), (2), (3), and (4) as in the cases of edge insertions or deletions of non-tree edges. Thus, the same arguments as above show that all this can be implemented in time $O(k)$.

We summarize the section with the following theorem.

THEOREM 2.16. *The given data structure*

(1) *tests in constant time whether two vertices u and v of $V(C)$ for a cluster C that are not separated by s_C are biconnected in G,*

(2) *can be updated in amortized time $O(k)$ after each update in G, and*

(3) *can be built in time $O(m)$.*

2.6. Shared graphs. We maintain a shared graph $G(s)$ for every shared vertex s. Given a shared vertex s and two of its tree neighbors x and y, the shared graph of s is used to test in constant time whether s is an articulation point separating x and y in G.

Let C_s be the node of H_2 representing s, i.e., it represents the nodes in all s-clusters. Let $\mathcal{V}(C_s) = \{v; v \in G$, and v is represented by $C_s\}$. Shared graphs are used to test whether a pair of two special vertices of G that either are represented by the same node of H_2 or are incident to the same node of H_2 (to be precise, two tree neighbors of s) are biconnected in G. Note that cluster graphs solve this problem in H_1: a cluster graph tests whether two vertices of G that are represented by or are incident to the same node of H_1 are biconnected in G, under the additional condition that no dashed edge is incident to this node of H_1. (For nodes of H_1 that are incident to a dashed edge only a restricted version of the problem is solved.) Since there are no dashed edges in H_2, we simply can define shared graphs analogously to cluster graphs and use the data structure for cluster graphs also for shared graphs. However, it is possible that $|C_s| = \Theta(m)$ and, thus, rebuilding the data structure from scratch can take time $\Theta(m)$.

This leads to the following definition. Let us call a shared vertex s *new* if s became shared by a cluster split after the last rebuild, and let it be called *old* otherwise (i.e., if it became a shared vertex during the last rebuild). Note that if s is new, then $|\mathcal{V}(C_s)| = O(k)$ by Lemma 2.10, and, thus, a solution analogous to cluster graphs is efficient.

For old shared vertices we use a new technique, which exploits the fact that the tree neighbors x and y of s are biconnected iff x and y are connected in $G \setminus s$. Thus, we maintain a "compressed" version of $G \setminus s$ in which we ask *connectivity* queries. The shared graph will be stored in a dynamic connectivity data structure. Note that there are $O(m/k) = O(k)$ many shared vertices, which implies we have to maintain $O(k)$ many shared graphs.

2.6.1. Shared graphs for new shared vertices. Let s be a new shared vertex represented by node C_s in H_2, and let A_s be the ancestor of C_s. The *shared graph* $G(s)$ contains as nodes

(1) a node, called *a-node*, for each vertex in $\mathcal{V}(C_s)$,

(2) one node, called *b-node*, for each neighbor of C_s with ancestor A_s,

(3) one node, called *c-node*, for each maximal set X of nodes of H_2 such that (a) every cluster $C' \in X$ is a neighbor of C_s, (b) all nodes in X are connected in $H_2 \setminus C_s$, (c) all nodes in X have the same ancestor which is different from A_s, (d) at the creation of C_s, the set X contains only one node, and (e) at each previous point in time since the creation of C_s all nodes of H_2 that contain the vertices in $\cup_{C' \in X} C'$ existing at this time are represented by the same c-node in $G(s)$.

Note that for each neighbor C' of C_s there exists a unique node in $G(s)$ representing C' and potentially other clusters.

The graph $G(s)$ contains the following edges:

(1) All edges between two vertices of $\mathcal{V}(C_s)$ belong to $G(s)$.

(2) For each edge (u, v), where u belongs to $\mathcal{V}(C_s)$, v does not belong to $\mathcal{V}(C_s)$, and (u', v') is the corresponding edge in G, there is an edge (u, d), where d is the b- or c-node representing $C_{v'}$ in $G(s)$.

(3) All b- or c-nodes representing tree neighbors of C that are biconnected in H_2 are connected by a tree of *red* edges.

(4) For each non-tree neighbor C_1 of C, $G(s)$ contains a *blue* edge (d_1, d_2), where d_2 represents a tree neighbor of C that is biconnected to C_1 in H_2, and d_1 represents C_1.

Note that for each non-tree neighbor C_1 of C there always exists a tree neighbor of C that is biconnected to C_1 in H_2—the tree neighbor of C that lies on $\pi_{T_2}(C, C_1)$ always is biconnected to C_1.

Since all clusters sharing s have the same ancestor, Lemma 2.10 shows that $|G(s)| = O(k)$.

A tree neighbor of s either belongs to $\mathcal{V}(C_s)$ and is represented by an a-node, or does not belong of $\mathcal{V}(C_s)$ and is represented by a b- or c-node. We need to show the following lemma.

LEMMA 2.17. *Let x and y be two tree neighbors of a new shared vertex s. Then (the representative of) x and y are biconnected in $G(s)$ iff x and y are biconnected in G.*

Proof. Note that $G(s)$ can be created by contracting edges in G. Thus, biconnectivity in $G(s)$ implies biconnectivity in G.

Vertices x and y are connected by a tree path $(u, s), (s, v)$. Thus s is the only node that could be an articulation point separating x and y. Contracting edges not incident to s cannot make s into an articulation point separating x and y. Hence, biconnectivity in G implies biconnectivity in $G(s)$. ∎

We use the same data structure as for cluster graphs to store shared graphs for new shared vertices. Given the b- and c-nodes the red edges can be found in time linear in their number using the data structure (HL4) for H_2. We determine the blue edges of $G(s)$ by connecting each non-tree neighbor C_1 of a node C to the tree neighbor of C on $\pi_{T_2}(C, C_1)$. Using the data structure (HL3) for H_2 this takes time linear in the number of blue edges times $O(\log n)$. Using the data structure of [11] results in the following lemma.

LEMMA 2.18. *Let s be a shared vertex represented by the node C of H_2. Then there exists a data structure for the shared graph of s such that*

(1) *building the data structure takes time $O(|C_s| + (m/k) \log n)$, provided that the b-nodes and the c-nodes are given;*

(2) *changing one or all of the colored edges in the data structure takes time linear in their total number times $O(\log n)$, given the b-nodes and c-nodes;*

(3) *testing whether two vertices of* $\mathcal{V}(C_s)$ *are biconnected in* $G(s)$ *takes constant time.*

These data structures are updated with the algorithm of section 2.5.1 with H_1 replaced by H_2. Since the test in (HL3) takes time $O(\log n)$, the amortized time per operation is $O(k + (m/k)\log n)$.

2.6.2. Shared graphs for old shared vertices. Let s be an old shared vertex and let C_s be the node of H_2 representing s. We cannot use the data structure of the previous section for the shared graph of s since rebuilding the data structure from scratch would take time $\Omega(|C_s|)$, which might be $\Theta(m)$. Still we use an approach similar to the one in the previous section but avoid rebuilds from scratch for the whole data structure.

We will exploit the following fact: Condition (C6) of a relaxed partition guarantees that at any time in a phase $O(m/k)$ vertices that do not belong to C_s are incident to a vertex of C_s. Only the $O(m/k)$ shared vertices and the $O(m/k)$ endpoints of edges inserted in the phase can be neighbors of a vertex in C_s and not belong to C_s.

The graphs $G(s)$. Let s be an old shared vertex,

(1) let C_s be the node representing s in H_2,

(2) let A_s be the ancestor of C_s in H_2, and

(3) let $\mathcal{V}_s = \{v, v \in G$ and v is represented by a node of H_2 with ancestor $A_s\}$.

Note that at the beginning of a phase $\mathcal{V}_s$ consists exactly of all the vertices in C_s. Later in the phase, the vertices of G in C_s are all contained in $\mathcal{V}_s$, but $\mathcal{V}_s$ can contain additional vertices.

The graph $G(s)$ contains as vertices

(1) a node, called *a-node*, for each vertex in $\mathcal{V}_s$, except for s, and

(2) a node, called *d-node*, for each vertex of G that does not belong to $\mathcal{V}_s$ but is a neighbor of a vertex in $\mathcal{V}_s$.

The graph $G(s)$ contains as edges every edge between two a-nodes or between an a-node and a d-node.

LEMMA 2.19. *Let* s *be an old shared vertex. The number of d-nodes in* $G(s)$ *is* $O(m/k)$.

Proof. By condition (C6) of a relaxed partition the number of d-nodes at the beginning of a phase connected to a vertex of $\mathcal{V}_s$ by non-tree edges is $O(m/k)$. Additionally there are $O(m/k)$ tree edges. During the phase only the endpoints of edges inserted during the phase can become neighbors of vertices in $\mathcal{V}_s$. Thus, the lemma follows. ∎

Data structure:

(S1) We store $G(s)$ in a fully dynamic connectivity data structure. This data structure allows us to execute the following operations:

(1) *insert(u,v)/delete(u,v):* Insert or delete the edge (u, v) in time $O(\sqrt{m'})$, where m' is the number of edges in $G(s)$.

(2) *insert(u):* Insert the degree-0 vertex u in time $O(\sqrt{m'})$, where m' is the number of edges in $G(s)$.

(3) *connected?(u,v):* Test whether u and v are connected in constant time.

(4) *component?(u):* Return the connected component of u in constant time.

(S2a) We keep for each connected component of $G(s)$ a list of all the d-nodes that belong to it.

(S2b) We keep for each connected component of $G(s)$ a list of all neighbors of C_s with ancestor A_s whose vertices belong to the connected component. We also store for each node of H_2 its position in the at most one such list to which it belongs.

LEMMA 2.20. *The data structures for $G(s)$ for all old shared vertices s can be built in time linear in its size at the beginning of a phase and can be updated in time $O(\sqrt{m})$ after an edge insertion or deletion in G.*

Proof. At the beginning of a phase the vertices of $G(s)$ are simply the vertices in C_s and their neighbors and are given by (HL1) and (G1). The edges are given by (G1). Building (S1) and (S2a) takes linear time; (S2b) is an empty list.

Each edge insertion or deletion in G affects the data structure for $G(s)$ of at most one old shared vertex s since the sets $\mathcal{V}_s$ are vertex-disjoint for different old shared vertices s. Since the graph $G(s)$ consists of at most m edges, its fully dynamic connectivity data structure can be updated in time $O(\sqrt{m})$.

In case of the deletion of the edge (u', v') we test after the removal of the edge from $G(s)$ whether u' and v' are still connected. If not, we test each d-node in the old connected component of u' and v' whether it is either connected to u' or to v'. In this way we construct the list of d-nodes for the two new connected components of $G(s)$. By Lemma 2.19 this requires $O(m/k)$ tests. In the same way we split the (S2b) list of the old connected component to create the (S2b) lists of the two new connected components and update the corresponding positions at the nodes of H_2.

In case of an edge insertion we insert a new d-node if one of the endpoints of the new edge does not belong to $G(s)$. Then we test whether the endpoints of the edge belonged to the same connected component before the insertion. If they did not, we combine their lists of d-nodes and their lists of neighbors of C_s. Note that these lists were disjoint before the combination since otherwise the two connected components would have shared a vertex.

Additionally we determine for each cluster split by an update operation whether its node in H_2 is split as well and whether a position in an (S2b) list is stored at the node. If so, we update the entry in the (S2b) list accordingly and store the appropriate positions at the new nodes. Since only a constant number of nodes is split by Lemma 2.5, this takes constant time.

Thus, the total time of an update is $O(\sqrt{m})$. ∎

Note that each tree neighbor x of s either belongs to C_s or is represented by a node of H_2 that is a neighbor of C_s. In the latter case we call the neighbor of C_s representing x the *x-neighbor* of C_s. In the former case we determine a node that is a neighbor of C_s and connected to x in $G(s)$ as follows: Using (S1) and (S2b) we can determine in constant time a neighbor of C_s with ancestor A_s that is in the connected component of x. If no such node exists, then using (S1) and (S2a) we can determine in constant time a d-node, and using (HL1) a node of H_2, that is connected to x in $G(s)$ if such a d-node exists. Note that in the latter case the d-node is connected to a node in C_s, i.e., the node of H_2 containing the d-node is a neighbor of C_s, since no node with ancestor A_s belongs to the connected component of x. In either case the determined neighbor of C_s is called an *x-neighbor* of C_s.

The graphs $\tilde{G}(s)$. The intuition for the graph $\tilde{G}(s)$ is as follows: Initially, and whenever $\tilde{G}(s)$ is rebuilt, each neighbor of C_s is represented by a vertex in $\tilde{G}(s)$. If this neighbor is split into two nodes C_1 and C_2 of H_2, then the corresponding vertex of $\tilde{G}(s)$ is updated (and the whole graph $\tilde{G}(s)$ is rebuilt) only if C_1 and C_2 are not connected in $H_2 \setminus C_s$ after the update. If C_1 and C_2 are connected in $H_2 \setminus C_s$ after the

update, then the vertex of $\tilde{G}(s)$ is not modified, but represents now both nodes of H_2. This would guarantee that after an edge update we only have to update the graph $\tilde{G}(s)$ of an old shared vertex s if an edge of $G(s)$ was modified (i.e., $\mathcal{V}_s$ contains an endpoint of the updated edge) or the connected components of $H_2 \setminus C_s$ are modified by the update. However, to bound the number of these graphs using the amortization lemma of section 2.9 we need to treat nodes with ancestor A_s in a special way.

We formalize this as follows: The graph $\tilde{G}(s)$ contains as vertices

(1) one node, called *e-node*, for each maximal set X of nodes of $H_2 \setminus C_s$ such that (a) every node in X is a neighbor of C_s, (b) all nodes in X are connected in $H_2 \setminus C_s$, (c) all nodes in X have the same ancestor which is different from A_s, (d) at the creation of C_s, the set X contains only one element, and (e) the vertices in $\cup_{C' \in X} C'$ used to belong to the same node C_{old} of H_2 and since the split of C_{old}, the graph $G(s)$ was not modified and the connected components of $H_2 \setminus C_s$ did not change;

(2) one node, called *b-node*, for each neighbor of C_s with ancestor A_s.

Each vertex in $\tilde{G}(s)$ *represents* the nodes in the set X and thus also the vertices of G contained in these nodes.

The graph $\tilde{G}(s)$ contains the following edges.

(1) All vertices of $\tilde{G}(s)$ that contain vertices that are connected in $G(s)$ are connected by a tree of *yellow* edges in $\tilde{G}(s)$.

(2) All vertices of $\tilde{G}(s)$ whose nodes are connected in $H_2 \setminus C_s$ are connected by a tree of *green* edges in $\tilde{G}(s)$.

Let x be a vertex of $\mathcal{V}_s$ and a tree neighbor of s. Note that by definition all nodes of $\tilde{G}(s)$ representing an x-neighbor are connected in $\tilde{G}(s)$ by yellow edges, i.e., belong to the same connected component of $\tilde{G}(s)$.

Data structure:

(S3) We store $\tilde{G}(s)$ in an adjacency list representation and label each node with its connected component.

(S4) We store for each node C of H_2 an array with one entry per old shared vertex. The array stores for each old shared vertex s the vertex in $\tilde{G}(s)$ that represents C in $\tilde{G}(s)$ if such a vertex exists and *null* otherwise.

The next lemma shows how to use $G(s)$ and $\tilde{G}(s)$ to test whether two neighbors of s are biconnected in G.

LEMMA 2.21. *Two tree neighbors x and y of s are biconnected in G iff*

(1) *either x and y are connected in $G(s)$, or*

(2) *the connected component of the vertices of $\tilde{G}(s)$ representing x-neighbors is identical to the connected component of the vertices of $\tilde{G}(s)$ representing y-neighbors.*

Proof. Each edge in $G(s)$ or $\tilde{G}(s)$ corresponds to a path in G that does not contain s, and each vertex $u \in \mathcal{V}_s \setminus \{s\}$ is either contained in the u-neighbor or connected to every u-neighbor by a path in $G(s)$ that does not contain s. Additionally connectivity of x and y in $G \setminus \{s\}$ implies biconnectivity in G. Thus, if x and y are connected in $G(s)$ or if a vertex of $\tilde{G}(s)$ representing an x-neighbor of C_s is connected with a vertex of $\tilde{G}(s)$ representing a y-neighbor of C_s in $\tilde{G}(s)$, then x and y are biconnected in G.

To show the other direction consider a path P in G that connects x and y and does not contain s. If all vertices of P belong to $\mathcal{V}_s \setminus \{s\}$, then x and y are connected in $G(s)$.

Otherwise, recall that all vertices of $G(s)$ representing an x-neighbor (resp., y-neighbor) form a connected component of $\tilde{G}(s)$. It follows that it suffices to show that one of the vertices of $\tilde{G}(s)$ representing an x-neighbor is connected in $\tilde{G}(s)$ to one

of the vertices of $\tilde{G}(s)$ representing a y-neighbor. In this case P contains a d-node d_x that is connected to x by a path in $G(s)$. If the path from x to d_x contains only vertices of C_s (excluding d_x), then let b_x be the vertex of $\tilde{G}(s)$ representing d_x and let u_x be d_x. If the path from x to d_x contains vertices not in C_s, let b_x be the vertex of $\tilde{G}(s)$ representing the first such node u_x on the path. In either case b_x represents an x-neighbor. Define u_y and b_y in the same way.

Consider the subpath P' of P between the u_x and u_y. Partition P' into subpaths such that each subpath is a maximal sequence of edges such that either (a) each edge is incident to a vertex of C_s or (b) no edge is incident to a vertex of C_s. Note that the endpoints of each subpath belong to nodes in H_2 that are neighbors of C_s.

Since P does not contain s, each subpath fulfilling (a) corresponds to a path in $G(s)$. Thus the vertices of $\tilde{G}(s)$ representing the endpoints of the subpath are connected by a path of yellow edges in $\tilde{G}(s)$. Each subpath fulfilling (b) connects the nodes of H_2 containing the endpoints of the subpath by a path in $H_2 \setminus C_s$. Thus the vertices of $\tilde{G}(s)$ representing these nodes of H_2 are connected by a path of green edges in $\tilde{G}(s)$. It follows that b_x is connected to b_y in $\tilde{G}(s)$. □

The first condition is tested in constant time using (S1). To test the second condition we determine the x-neighbor and y-neighbor using (S1), (S2a), (S2b), and (HL1). We determine the vertices of $\tilde{G}(s)$ representing the x-neighbor and the y-neighbor using (S4) and then test their connected components in $\tilde{G}(s)$ using (S3). All this takes constant time.

Let $deg_2(C_s)$ be the degree of C_s in H_2.

LEMMA 2.22. *If only the green edges of $H_2 \setminus C_s$ change, then $\tilde{G}(s)$ and the data structures* (S3) *and* (S4) *can be updated in time $O(deg_2(C_s) \log n)$.*

Proof. Discard all old green edges. To compute the new green edges, map each neighbor of C_s to a tree neighbor of C_s in H_2 using (HL3) and determine the biconnected component of this tree neighbor using *blockid?*-queries in (HL4) for H_2. Then bucketsort the neighbors according to these biconnected components using (HL6). For each neighbor in a biconnected component determine its vertex in $\tilde{G}(s)$ using (S4) and connect it by a green edge to the vertex in $\tilde{G}(s)$ of the previous neighbor in the same biconnected component. Then compute a spanning forest of the green edges by performing a depth-first search on the graph of green edges, and discard all green edges not in the spanning tree. Finally recompute the connected components of $\tilde{G}(s)$. This takes time $O(\log n)$ per neighbor and constant time per green edge. Since the number of green edges is linear in the number of neighbors, the lemma follows. □

LEMMA 2.23. *Let s be an old shared vertex. At the beginning of a phase or whenever $G(s)$ has changed, the graph $\tilde{G}(s)$ can be constructed in time $O((m/k) \log n)$.*

Proof. Note first that the size of $\tilde{G}(s)$ is linear in its number of vertices of $\tilde{G}(s)$, which is bounded by $deg_2(C_s)$.

The vertices of $\tilde{G}(s)$ are the neighbors of C_s and are given by (HL2). To determine the yellow edges, process the d-nodes belonging to the same connected component of $G(s)$ as follows. Map each d-node in (S2a) for this connected component to its node in H_2 using (HL1). Append to this list the neighbors of C_s stored in (S2b) for this connected component. Now process each node C in this list as follows: Determine the vertex of $\tilde{G}(s)$ representing C and connect it by a yellow edge to the vertex of $\tilde{G}(s)$ representing the previous node on the list. This takes time linear in the length of the list. Then compute a spanning tree of yellow edges by performing a depth-first search on the graph of yellow edges. Discard the yellow edges that do not belong to the spanning tree. This takes time linear in the number of yellow edges, which is linear

in the length of the list. The length of the list is linear in the number of d-nodes in $G(s)$ in the connected component and the length of the (S2b) list for the connected component. By Lemma 2.19 there are $O(m/k)$ many d-nodes in $G(s)$, i.e., in the (S2a) lists for all connected components. Also there are $O(m/k)$ many neighbors, i.e., in the (S2b) lists for all connected components. Thus, the total time for determining all yellow edges in $\tilde{G}(s)$ is $O(m/k)$.

The green edges are computed in time $O((m/k)\log n)$ as in Lemma 2.22. □

LEMMA 2.24. *Constructing the graphs $\tilde{G}(s)$ for all old shared vertices at the beginning of a phase takes time $O(m \log n)$. Building the data structures* (S3) *and* (S4) *for all s at the beginning of a phase takes time $O(m)$.*

Proof. By Lemma 2.23, constructing one graph $\tilde{G}(s)$ takes time $O((m/k)\log n)$, for a total of $O((m/k)^2 \log n) = O(m \log n)$.

When $\tilde{G}(s)$ is given, then building (S3) for s takes time linear in the size of $\tilde{G}(s)$. To build (S4) we allocate and initialize with *null* all the necessary arrays. This takes time $O((m/k)^2) = O(m)$. Then we process each graph $\tilde{G}(s)$ and set the entry for s in the array of node C of H_2 to the vertex of $\tilde{G}(s)$ representing C if it exists. □

LEMMA 2.25. *The graphs $\tilde{G}(s)$ and the data structures* (S3) *and* (S4) *for all old shared vertices s can be updated in amortized time $O((m/k)\log n)$ after an edge insertion or deletion in G.*

Proof. We will show that after the insertion or deletion of the edge (u, v) in G the only old shared vertices s for which $\tilde{G}(s)$ has to be updated are the ones (1) where $\mathcal{V}_s$ contains u or v, (2) where a node of H_2 that is split by the update has ancestor A_s, or (3) where C_s is an articulation point on $\pi_{T_2}(C_u, C_v)$ before or after the current update. For type-(3) old shared vertices either (i) a vertex of $\tilde{G}(s)$ has to be split and the green and yellow edges have to be recomputed or (ii) only the green edges have to be updated. If a yellow edge has to be changed without splitting a vertex, then $\mathcal{V}_s$ must contain u or v, i.e., it is a type-(1) old shared vertex.

To guarantee that all graphs that have to be changed are indeed updated we use the following update algorithm:

(1) Rebuild from scratch the graphs $\tilde{G}(s)$ for all old shared vertices s where $\mathcal{V}_s$ contains either u or v (type (1) above), where C_s has the same ancestor as a split node in H_2 (type (2) above), or where a vertex of $\tilde{G}(s)$ is split (type (3(i)) above).

(2) Update the green edges in the graph $\tilde{G}(s)$ for the remaining old shared vertices where C_s is an articulation point on $\pi_{T_2}(C_u, C_v)$ before or after the update (case (3(ii)) above).

To find all type-(2) old shared vertices, we determine for each split node of H_2 its ancestor A using (HL7) and test for each old shared vertex s whether the ancestor of C_s equals A. The old shared vertices for which this test returns true are the type-(2) old shared vertices.

We discuss next how to determine all type-(3(i)) and type-(3(ii)) old shared vertices. There are two kinds of vertices in $\tilde{G}(s)$, e-nodes and b-nodes. If a b-nodes has to be split, then s is also a type-(2) old shared vertex and will be updated correctly. Thus, it suffices to determine all type-(3(i)) old shared vertices s where an e-nodes has to be split. In section 2.8 we give a data structure that determines all e-nodes that have to be split in time $O((m/k)\log n)$. Furthermore, we show in section 2.9 that only an amortized constant number of e-nodes has to be split. We determine all articulation points on $\pi_{T_2}(C_u, C_v)$ using (HL4). This gives all type-(3) old shared vertices. We remove the ones that are of type (2). In the remaining set the ones that contain an e-node that has to be split are the type-(3(i)) but not type-(2) old shared

vertices; the rest are the type-(3(ii)) old shared vertices.

Whenever $\tilde{G}(s)$ is rebuilt, we also rebuild (S3) from scratch. The data structure (S4) is updated as follows:

(1) For each new node that is created by a split of an old node during the phase, the array of the old node is copied to create the array for the new node.

(2) For each old shared vertex s where $\tilde{G}(s)$ is rebuilt, the entry of s in the array of every neighbor of C_s is replaced by the vertex representing the neighbor in $\tilde{G}(s)$.

Lemma 2.23 shows that for type-(1), type-(2), and type-(3(i)) old shared vertices the graph $\tilde{G}(s)$ can be constructed in time $O((m/k)\log n)$. Lemma 2.22 shows that for type-(3(ii)) old shared vertices the green edges can be updated in time $O(deg_2(C_s)\log n)$. Determining all old shared vertices s such that a vertex of $\tilde{G}(s)$ has to be split or that $A_s = A$ takes time $O(m/k)$ per split cluster. By Lemma 2.5 there are a constant number of split clusters per update, i.e., the total time spent to determine type-(2) and type-(3(i)) old shared vertices is $O(m/k)$. Determining all articulation points and all type-(3(ii)) old shared vertices takes time $O(m/k)$. Building (S3) takes time $O(deg_2(C_s))$. The first type of update of (S4) creates a new array and takes time $O(m/k)$; the second type takes time $O(deg_2(C_s))$. Thus, updating the graphs $\tilde{G}(s)$ and the data structure (S3) for type-(1), (2), and (3(i)) old shared vertices takes time $(O(m/k)\log n)$ each; updating the graphs $\tilde{G}(s)$ and the data structure (S3) for type-(3(ii)) old shared vertices takes time $O(deg_2(C_s)\log n)$ each.

We show next that there are only an amortized constant number of type-(1), (2), and (3(i)) old shared vertices whose graphs $\tilde{G}(s)$ have to be updated, for a total time of $O((m/k)\log n)$ to update them. There are at most two type-(1) old shared vertices. There is at most one type-(2) old shared vertex per split node C of H_2, since the fact that C has ancestor A_s implies that the vertices of C belong to $\mathcal{V}_s$ and the sets $\mathcal{V}_s$ are disjoint for different old shared nodes. By Lemma 2.43 and Lemma 2.44 there are an amortized constant number of type-(3(i)) old shared vertices, where an e-node has to be split.

By Fact 2.1 the degree of all articulation points on a path in H_2 sums to $O(m/k)$. Thus, $deg_2(C_s)$ for all type-(3(ii)) old shared vertices sums to $O(m/k)$. Hence, updating all graphs $\tilde{G}(s)$ and building (S3) for the type-(3(ii)) old shared vertices takes time $O((m/k)\log n)$. Finally, a new array in (S4) is created for a constant number of new nodes of H_2. Summing all the cost gives a total time of $O((m/k)\log n)$ to update all graphs $\tilde{G}(s)$ and their data structures (S3) and (S4).

We still need to show the above claim about which graphs $\tilde{G}(s)$ have to be changed and how. A graph $\tilde{G}(s)$ has to be updated if (A) a vertex, (B) a yellow edge, or (C) a green edge has to be changed.

(A) If a b-node of $\tilde{G}(s)$ has to be changed, then a node in H_2 with ancestor A_s was split (case (2) above). If an e-node of $\tilde{G}(s)$ has to be changed, then either condition (a), (b), or (c) of an e-node does not hold anymore. If (a) does not hold anymore, then the current update deleted an edge of $G(s)$, i.e., $\mathcal{V}_s$ contains an endpoint of the updated edge (case (1) above). If (b) does not hold anymore, then C_s is an articulation point separating the endpoints of the updated edge before or after the current update (case (3) above). If (c) does not hold, then again either the current update modified an edge of $G(s)$ (case (1) above), or C_s is an articulation point separating the endpoints of the updated edge before or after the current update (case (3) above).

(B) If a yellow edge but no vertex of $\tilde{G}(s)$ has to be changed, then an update occurred in the graph $G(s)$. It follows that $\mathcal{V}_s$ contains at least one of the endpoints of the updated edge (case (1) above).

(C) If a green edge has to be changed, then C_s is an articulation point separating the endpoints of the updated edge before or after the current update (case (3) above).

This completes the proof of the lemma. □

We summarize the section with the following theorem.

THEOREM 2.26. *The given data structure*

(1) *tests in constant time whether two tree neighbors x and y of a shared vertex s are biconnected in G,*

(2) *can be updated in amortized time $O((m/k)\log n + \sqrt{m})$ after each update in G, and*

(3) *can be built in time $O(m \log n)$.*

2.7. The high-level graphs. In this section we give the details of the high-level graph data structures and explain how they are updated. For (HL1), (HL2), (HL5), (HL6), and (HL7) the details are given in section 2.4 and it is obvious how to update them in time $O(k)$ per update in G, provided the change in the cluster partition is known. The description in section 2.2 gives an $O(k)$-time algorithm to update the cluster partition. Thus, we concentrate in this section on the details of (HL3) and (HL4).

2.7.1. The data structure (HL3). Consider the graph H_i. Given the two nodes C and C' of H_i we use (HL3) to find the tree neighbor of C on $\pi_{T_i}(C, C')$. For H_2 (HL3) consists of a dynamic tree data structure of T_2. To find the tree neighbor, root T_2 at C' and return the parent of C. Update (HL3) by executing link and cut operations whenever T_2 changes. It takes time $O(\log n)$ to test or update (HL3).

For H_1, (HL3) consists of a degree-k ET-tree data structure of T_1. To find the tree neighbor proceed as follows. For the node C', and the tree neighbors of C, label one of the leaves representing the node with the name of the node. Then traverse the ET-tree from all labeled leaves in lockstep, labeling each internal node with the concatenation of the label of its left child and the label of its right child. At the root the label incident to C''s label is the desired tree neighbor. To update the ET-tree, whenever T_1 changes, split and join the ET-tree accordingly. Since the ET-tree has $O(1)$ depth and C has $O(1)$ tree neighbors, a test takes $O(1)$ time. Each update takes time $O(k)$.

2.7.2. The data structure (HL4). Recall that (HL4) implements the following query operations in H_i:

(1) *biconnected?(C,C′,C″):* Given that nodes C' and C'' are both tree neighbors of a node C test whether C' and C'' are biconnected in H_i.

(2) *blockid?(C,C′):* Given that C and C' are tree neighbors in T_i, output the name of the biconnected component of H_i that contains both C and C'.

(3) *components?(C):* Output the tree neighbors of C in H_i grouped into biconnected components.

As we show below, *biconnected?* and *blockid?* take constant time, and *components?* takes time linear in the size of the output.

We say a node C of H_i is *avoidable on the tree path P* iff C and two of its tree neighbors, called D and D', belong to P and there is an edge in $H_i \setminus C$ between the subtree of $T_i \setminus C$ containing D and the subtree containing D'. Note that if C is avoidable, then C does not separate D and D'. However, if C does not separate D and D' it does not follow that C is avoidable.

To implement (HL4) we build a compressed graph $H_i(C)$ of $H_i \setminus C$ for each node C in H_2. Let C be a node in H_i. The graph $H_i(C)$ contains a node for each tree

neighbor of C in H_i. There is an edge between two tree neighbors D and D' of C iff C is avoidable on $\pi_{T_i}(D, D')$.

We keep the following data structures. The second is needed to efficiently maintain the $H_i(C)$.

(HL4-1) For $i = 1, 2$, and for each node C we store $H_i(C)$ in a dynamic connectivity data structure.

(HL4-2) A 2-dimensional topology tree [5] of T and one ambivalent data structure [6] are maintained. They implement the following operations in time $O((m/k) \log n)$:

(1) *insert&return_avoidable(u,v):* Return all nodes on $\pi(C_u, C_v)$ that become avoidable on $\pi(C_u, C_v)$ by the insertion of edge (u, v), where C_u and C_v are the endpoints in H_i of the newly inserted edge.

(2) *delete&return_unavoidable(u,v):* Return all nodes on $\pi(C_u, C_v)$ that become unavoidable on $\pi(C_u, C_v)$ by the deletion of edge (u, v), where C_u and C_v are the endpoints in H_i of the newly deleted edge.

We show how to implement the operations of (HL4) using (HL4-1).

(1) *biconnected?(C,C′,C″)*: Return *connected?(C′, C″)* in $H_i(C)$.

(2) *blockid?(C,C′)*: Return *component?(C′)* in $H_i(C)$.

(3) *components?*: Return all connected components of $H_i(C)$, and for each connected component output all its nodes.

The correctness of this implementation is shown by the following lemma.

LEMMA 2.27. *Two tree neighbors D and D' of C in H_i are biconnected in H_i iff they are connected in $H_i(C)$.*

Proof. If D and D' are connected in $H_i(C)$, then every edge on the path connecting D and D' corresponds to a path in $H_i \setminus C$. Thus they are biconnected in H_i. If D and D' are biconnected in H_i, they are connected by a path in $H_i \setminus C$. Every edge on this path either lies in a subtree of $T_i \setminus C$ or connects two subtrees. The edges connecting two subtrees give a path in $H_i(C)$ connecting D and D'. □

Updates in $H_i(C)$. Consider an insertion of edge (u, v) in G and let C_u and C_v be the nodes in H_i incident to the edge. The graph $H_i(C)$ has to be modified only for the nodes that become avoidable on $\pi(C_u, C_v)$. These nodes can be found with one *insert&return_avoidable* operation in (HL4-2). Let C be such a node. Note that exactly one edge is added to $H_i(C)$, namely, the edge between the tree neighbors of C on $\pi(C_u, C_v)$.

An edge deletion in G is handled analogously.

To analyze the running time recall that the operation in (HL4-2) takes time $O((m/k) \log n)$. The update in a graph $H_i(C)$ takes time $O((deg_T(C))^{1/3} \log deg_T(C))$, where $deg_T(C)$ is the degree of C in T_i. Since $\sum_C deg_T(C) = O(k)$, the cost for updating all $H_i(C)$ is $O(k)$. This shows the following theorem.

THEOREM 2.28. *There exists a data structure that implements the operations biconnected?, blockid?, and components? in time linear in their output. The data structure can be updated in time $O(k + (m/k) \log n)$ after each update operation in G.*

2.7.3. The data structure (HL4-2). We present now a data structure that implements *insert&return_avoidable* and *delete&return_unavoidable* in time $O((m/k) \log n)$.

We will actually show a slightly more general result: we give a data structure that can be updated in time $O(m/k)$ after each update in G and that can return all avoidable nodes on a tree path P in H_i in time $O((m/k) \log n)$. Obviously this data structure can be used to execute the above operations in time $O((m/k) \log n)$.

In this section we assume that T_1 is rooted at a node R.

The data structure consists of two parts: (1) a 2-*dimensional topology tree* and (2) an *extended ambivalent data structure*. Both are slight variations of data structures defined in [5, 6].

Let $e = (D, C)$ be an edge of T_1. We denote by $ST(D, C)$ the subtree of $T_1 \setminus e$ that contains D. Obviously a node C of H_1 is avoidable on a tree path P iff there exists an edge between $ST(D, C)$ and $ST(D', C)$, where D and D' are the neighbors of C on P. We call a node on P that is not an endpoint of P an *internal* node of P.

The 2-dimensional topology tree and the extended ambivalent data structure are based on H_1, not H_2. Thus, to test avoidability in H_2 we need to reduce it to testing avoidability in H_1. For each path P in T_2, let P_{T_1} denote the corresponding path in T_1 starting, resp., ending, at an arbitrary node of H_1 that maps to the start node, resp., end node, of P. Recall that each node C of H_2 is created by a set of H_1-nodes that are connected by a path of dashed edges. For an internal node C on P, let $C(P_{T_1})$ and $C(P_{T_1})'$ denote the extreme-most nodes of that dashed path on P_{T_1}. We use the following lemma.

LEMMA 2.29. *Let C be a node of H_2 on a tree path P in T_2. Let D, resp., D', be the neighbor of $C(P_{T_1})$, resp., $C(P_{T_1})'$, on P_{T_1} that does not map to C. Then C is avoidable on P iff there is an edge between $ST(D, C(P_{T_1}))$ and $ST(D', C(P_{T_1})')$.*

Proof. Note that the edges incident to the subtree containing $h(D)$, resp., $h(D')$, in $H_2 \setminus C$ are identical to the edges incident to $ST(D, C(P_{T_1}))$, resp., $ST(D', C(P_{T_1})')$. □

As we show below, the 2-dimensional topology tree can test in time $O(m/k)$ whether there is an edge between $ST(D, C(P_{T_1}))$ and $ST(D', C(P_{T_1})')$. We extend the ambivalent data structure of [6] such that it can test in time $O(m/k)$

(1) for all but $O(\log n)$ nodes C of H_2 on a path P of T_2 whether there is an edge between $ST(D, C(P_{T_1}))$ and $ST(D', C(P_{T_1})')$; and

(2) for all but $O(\log n)$ nodes C of H_1 on a path P of T_1 whether there is an edge between $ST(D, C)$ and $ST(D', C)$.

Thus, to test the avoidability on a path P we use the ambivalent data structure to get the avoidability information for all but $O(\log n)$ nodes of P and we use the 2-dimensional topology tree for the remaining nodes.

Note that the term *tree edge* refers to an edge in T, T', or T_i, never to an edge in a topology tree.

The 2-dimensional topology tree. Given a restricted partition of order k we call each cluster of the partition a *level-0 cluster* or *basic cluster.* A *level-i cluster* is

(1) the union of two level-$(i-1)$ clusters that are connected by a tree edge such that one of them has tree degree 1 or both have tree degree 2, or

(2) one level-$(i-1)$ cluster if the previous rule does not apply.

A *topology tree* TT is a tree such that each node C at level i corresponds to a level-i cluster. If C is the union of two clusters C_1 and C_2 at level $i-1$, then C_1 and C_2 are the children of C and the tree edge (C_1, C_2) is stored at C. If C consists of one level-$(i-1)$ clusters C at level i, then C_1 is the only child of C in the topology tree. A *rooted topology tree* TT is a topology tree with the additional condition that R is only unioned when no other unions are possible.[8]

A 2-*dimensional topology tree* $2TT$ for TT is a tree that contains a node $C \times D$ at level i for every pair (C, D) of level-i clusters in TT. A level-$(i-1)$ node $C_1 \times D_1$

[8] We will exploit the rootedness in the next section.

is a child of a level-i node $C \times D$ iff C_1 is a child of C and D_1 is a child of D. We call each node in a topology tree or a 2-dimensional topology tree a *topology node*.

We keep TT and $2TT$. We store at every node $C \times D$ of $2TT$ with $C \neq D$ a bit that is set to 1 iff there is a non-tree edge between cluster C and cluster D.

Next we show how to use $2TT$ to test whether an edge exists between $ST(D, C)$ and $ST(D', C')$. Let (D, C) be a tree edge in T_1. The topology nodes *representing* $ST(D, C)$ are the nodes of TT (1) that are children of ancestors of C but are not ancestors of C, and (2) whose leaf descendants in TT belong to $ST(D, C)$. Since TT has depth $O(\log n)$ [5], $ST(D, C)$ is represented by $O(\log n)$ topology nodes.

LEMMA 2.30. *For $D \neq D'$ let (D, C) and (D', C') be edges of T_1 such that D and D' do not belong to $\pi_{T_1}(C, C')$. Let $X_1, \ldots, X_p$ be the topology nodes representing $ST(D, C)$ and let $Y_1, \ldots, Y_q$ be the topology nodes representing $ST(D', C')$ such that X_i and Y_i are level-i topology nodes.*

There is a non-tree edge between $ST(D', C')$ and $ST(D, C)$ iff a bit is set to 1 at a node $X \times Y$ of $2TT$ such that either

(1) *$X = X_i$ for some $1 \leq i \leq p$ and Y is a level-i descendant of a node Y_j for some $1 \leq j \leq q$, or*

(2) *$Y = Y_j$ for some $1 \leq j \leq q$ and X is a level-j descendant of a node X_i for some $1 \leq i \leq p$.*

Proof. Note that $ST(D, C)$ and $ST(D', C')$ are disjoint. It follows that the subtrees of TT rooted at $X_1, X_2, \ldots, X_p$ are disjoint from the subtrees rooted at $Y_1, Y_2, \ldots, Y_q$.

Let (A, B) be the non-tree edge between $ST(D, C)$ and $ST(D', C')$ with $A \in ST(D, C)$ and $B \in ST(D', C')$. Let X_i (Y_j) be the lowest ancestor of A (B) in TT that is a topology node representing $ST(D, C)$ ($ST(D', C')$). If $i \leq j$, there exists a level-i cluster Y which is a descendant of Y_j such that there is an edge between X_i and Y. It follows that the bit stored at $X_i \times Y$ is set to 1. If $j > i$, a symmetric argument applies.

If a bit is set to 1 at a node $X \times Y$ of $2TT$ such that either (1) $X = X_i$ for $1 \leq i \leq p$ and Y is a level-i descendant of a node Y_j for $1 \leq j \leq q$ or (2) $Y = Y_j$ for $1 \leq j \leq q$ and X is a level-j descendant of a node X_i for $1 \leq i \leq p$, then there is an edge between X and Y and, thus, between $ST(D, C)$ and $ST(D', C')$. □

Let X_i and Y_j be defined as in the lemma. Let $\mathcal{Y}_i = \{Y | Y$ is a level-i descendant of a topology node Y_j for some $i \leq j \leq q\}$ and let $\mathcal{X}_j$ be defined symmetrically. Note that the subtree in $2TT$ induced by the set of nodes $\{X_i \times Y$, for all $1 \leq i \leq p$, and all $Y \in \mathcal{Y}_i\}$ is isomorphic to a subtree of TT. The same holds with the roles of X and Y reversed. Thus, Lemma 2.29 shows how to check the avoidability of C on a path P by checking the bits of $O(m/k)$ nodes in $2TT$. Finding the topology nodes X_i and Y_i for all i takes time $O(\log n)$. As was shown in [5], $2TT$ can be maintained in time $O(k + m/k)$ after each update operation in G.

LEMMA 2.31. *A 2-dimensional topology tree can test in time $O(m/k)$ whether a node C is avoidable on a path P in a high-level graph H_i. It can be updated in time $O(k)$.*

The extended ambivalent data structure. To determine the avoidability of all but $O(\log n)$ nodes on a path P in H_i, for $i = 1, 2$, we simply extend TT and $2TT$ with additional labels to construct the *extended ambivalent data structure*. We will use two types of avoidability information, one for H_1 and one for H_2. Our approach is to partition T_i into complete paths and to keep avoidability information for each complete path. Then we show that each path P consists of subpaths of $O(\log n)$ complete paths.

Thus, P's avoidability can be determined from the avoidability information of these complete paths. To be precise let $P = \pi_{T_i}(A, B)$ in H_i. Let $P_1 = P$ if $i = 1$, and let $P_1 = P_{T_1}$ if $i = 2$. We partition P_1 at the least common ancestor LCA of its endpoints A_1 and B_1 into the paths $P_A = \pi_{T_1}(A_1, LCA)$ and $P_B = \pi_{T_1}(B_1, LCA)$. Note that both paths are *increasing*, i.e., they consist of a directed path toward the root R of T_1. We show below how to test the avoidability of all but $O(\log n)$ nodes of an increasing path by breaking it into $O(\log n)$ complete paths.

Recall that T_1 is stored in a rooted topology tree TT. Note that T_1 induces a rooted spanning tree TT_j of the nodes at each level j of TT whose root is R. Note further that when given TT, R, and also the least common ancestor between any two basic clusters can be determined in time $O(\log n)$.

We now give the necessary definitions. For a basic cluster X_1 let the graph $G(X_1)$ be (1) the graph induced by the vertices of X_1, if the tree degree of X_1 is 1 or 3, and (2) the graph induced by the vertices of X_1 with all vertices between the two boundary nodes contracted to one vertex, otherwise.

To construct complete paths we first need to introduce partial paths. We define the *partial path* of a basic cluster X_1 to be the (unique) endpoint $x(X_1)$ in $G(X_1)$ of the tree edge incident to X_1. In the following we often identify X_1 and $x(X_1)$. Note that if X_1 shares a vertex s, then the partial path of X_1 consists of s. The *partial path*[9] of a level-i cluster X_1 with $i > 0$ consists of

(1) *Case A:* the partial path of X_2, if X_1 consists of one level-$(i-1)$ cluster X_2,

(2) *Case B:* the concatenation of the partial path of X_2 and of X_3, if X_1 is the union of X_2 and X_3, and neither X_2 nor X_3 has tree degree 3.

(3) *Case C:* the vertex $x(X_3)$ and the two tree edges incident to it that are not incident to X_2 if X_1 is the union of X_2 and X_3, and X_2 has tree degree 1 and X_3 has tree degree 3. In this case the *complete path* of X_1 consists of the partial path of X_2 concatenated with the vertex $x(X_3)$. In all previous cases, the complete path of X_1 is not defined. Note that X_3 is the parent of X_2 in TT_j.

(4) *Case D:* an empty path if X_1 is the union of X_2 and X_3, and X_2 has tree degree 1 and X_3 has tree degree 1. In this case the *complete path* of X_1 consists of the partial path of X_2 and the partial path of X_3.

For every complete path not stored at the root of TT note that one endpoint has tree degree 1, and one has tree degree 3 (namely, the vertex of X_3). The endpoints of the complete path stored at the root of TT either both have tree degree 1 or one has tree degree 1 and one has tree degree 3. We call the endpoint with tree degree 1 the *tail* and the endpoint with tree degree 3 the *head* of the complete path. A tree-degree 3 node belongs to two complete paths; in one it is an internal node and in one it is a head. All other nodes belong to exactly one complete path.

If $x(X_1), \ldots, x(X_p)$ is the sequence of nodes on a partial or complete path P^c, then either $X_1, \ldots, X_p$ is an increasing path in T_1 or $X_1, \ldots, X_j$ and $X_p, X_{p-1}, \ldots, X_j$ are increasing paths, for some $1 < j < p$.

Next we show that P_A and also P_B consist of $O(\log n)$ increasing subpaths of complete paths. We will store avoidability information for the complete paths and use it to test the avoidability of P_A and P_B except for the nodes that are heads in the complete paths. Let $P_1^c, P_2^c, \ldots, P_l^c$ be the complete paths whose intersection with P_A is nonempty such that the head of P_j^c belongs to P_{j+1}^c. Let X_j be the topology node in TT at which P_j^c is stored. Note that all topology nodes whose partial path contains a vertex of P_j^c are true descendants of X_j in TT. Note further that the head

[9] A path is formed by a list of vertices.

of P^c_{j-1} belongs to P^c_j. Thus, X_{j-1} is a true descendant of X_j. Since TT has depth $O(\log n)$ it follows that P_A is contained in the union of $O(\log n)$ complete paths, i.e., $l = O(\log n)$. The same holds for P_B.

We use the algorithm described in the previous section to test the avoidability of LCA on P_A and P_B and for the heads of the complete paths. For all remaining nodes on P_A and P_B we use the extended ambivalent data structure. It consists of further labels for the 2-dimensional topology tree $2TT$ and search trees for the partial and complete paths. The labels and search trees will be oblivious of the rooting of T_1, which is important for the efficiency of rebuilds.

Every node $A \times B$ with $A \neq B$ is labeled with two additional labels *maxcov* and *shared* that are explained later. Each node $A \times A$ of $2TT$ is labeled with a pointer to the partial path and complete path (if it exists) of A. The partial and complete paths are stored in shared search trees as follows:

(1) The partial path of a level-0 cluster X is represented by one node $x(X)$.

(2) In Case A, the search tree of the partial path of X_1 is identical to the search tree of the partial path of X_2.

(3) In Case B, the partial path of X_1 consists of a (root) node pointing to the roots of the search trees of the partial paths of X_2 and X_3.

(4) In Case C, the partial path of X_1 consists of one node. The complete path of X_1 consists of a (root) node pointing to the roots of the search trees of the partial paths of X_2 and X_3.

(5) In Case D, the partial path of X_1 is empty. The complete path of X_1 consists of a (root) node pointing to the roots of the search trees of the partial paths of X_2 and X_3.

Since the topology tree has depth $O(\log n)$, every search tree has depth $O(\log n)$. A vertex v in the balanced search tree of a partial or complete path is labeled with two bits $somecov_i(v)$ for $i = 1, 2$.

Let C be an internal node on the increasing path Q in H_i whose avoidability we have to test. Let D and D' be the neighbors of C on Q such that D is the child and D' is the parent of C in T_i. Lemma 2.37 below shows that

(1) for $i = 1$, if a complete path P^c exists to which $x(D')$, $x(C)$, and $x(D)$ belong, then $somecov_1(v)$ is set to 1 for an ancestor v of $x(C)$ in P^c iff C is avoidable on Q, and

(2) for $i = 2$, let $C_1, \ldots, C_l$ form an increasing subpath of Q_{T_1} with $C_1 = C(Q_{T_1})$ and $C_l = C(Q_{T_1})'$. If a complete path exists to which $x(D')$, $x(D)$, and $x(C_q)$ for all $1 \leq q \leq l$ belong, then for all C_q in P^c, $somecov_2(v_q)$ is set to 1 for an ancestor v_q of $x(C_q)$ in P^c iff C is avoidable on Q.

If no such complete path exists, then some C_q is the head of a complete path and hence C is tested for avoidability using the 2-dimensional topology tree.

Note that P^c is the lowest ancestor of the least common ancestor of C for H_1 (resp., C_1 for H_2) and D in TT that has a complete path. It can be found in time $O(\log n)$.

Thus, if l nodes of an increasing path of H_1 lie on a complete path, we can test their avoidability except for the head of the complete path in time $O(l + \log n)$. Since the nodes of P_A and P_B are contained in $O(\log n)$ complete paths, we can test the avoidability of all nodes on P_A or P_B excluding LCA and the heads in time $O(m/k + \log^2 n) = O(m/k)$ for $k \leq m/\log^2 n$.

LEMMA 2.32. *Given a path P in H_i the extended ambivalent data structure can test the avoidability of all but $O(\log n)$ nodes on P in time $O(m/k)$.*

Let us now define *somecov*, *maxcov*, and *shared* and prove Lemma 2.37. For a cluster A the *projection* of a non-tree edge (u, v) with $u \in A$ and $v \notin A$ onto the partial or complete path P^c of A is the node x on P^c such that the tree path from u to x in $G(A)$ does not contain any other node on P^c.

Recall that every node $A \times B$ with $A \neq B$ is labeled with a constant number of labels: (1) For each tree edge E incident to A, there exists a label $maxcov(A, B, e)$ which is the node with maximum distance from e on the partial path of A that is avoidable because of a non-tree edge between A and B, assuming that the tree edge e incident to A lies on the tree path between A and B. (2) For each shared vertex s of A, there exists a label $shared(A, B, s)$ which is a bit that is set to 1 iff A shares s and there is an edge between A and B whose projections onto the partial path of A and of B are not nodes belonging to s.

Note that for each subpath of a complete path P^c there exist $O(\log n)$ nodes in the search tree of P^c whose leaf descendants form exactly a subpath of P^c. We say we *set the somecov$_i$ bits of a subpath* when we set the *somecov$_i$* bits of these $O(\log n)$ nodes, excluding the nodes representing the endpoints.

The *somecov$_i$* bits in the partial and complete paths are defined bottom-up. No basic cluster has a complete path and the partial path of every basic cluster consists of one node whose *somecov* bit is set to 0. The partial and complete path of a level-$(j + 1)$ cluster X_1 is computed with the help of the *maxcov* and *shared* labels at the nodes of $2TT$ as follows. If X_1 has two children let $e = (x, y)$ be the tree edge connecting them.

(1) In Case A, the partial path of X_1 is identical to the partial path of this child.

(2) In Case B, the partial path of X_1 is built by adding a node pointing to the balanced search trees of X_2 and X_3. The *somecov$_i$* bits of this node are unset. We set the *somecov*$_1$ bit to 1 for the path p between $maxcov(X_2, X_3, e)$ and $maxcov(X_3, X_2, e)$. We remove from p all but one representative of the shared vertices of the endpoints of p. This results in a subpath p'. If e is solid or if e is dashed, belonging to the shared vertex s, and $shared(X_2, X_3, s)$ is 1, then we also set the *somecov*$_2$ bits of p'. If e is dashed and $shared(X_2, X_3, s)$ is 0, then we split p' at the representatives of s and remove all but one representative of s from each of the resulting subpaths. We set the *somecov*$_2$ bits for these subpaths.

(3) In Case C, the partial path of X_1 consists of a tree of one node whose *somecov$_i$* bits are set to 0. The complete path of X_1 consists of the partial path of X_2 unioned with the partial path of X_3 which consists only of the node $x(X_3)$. We describe next which *somecov$_i$* bits of this complete path are set. We set the *somecov*$_1$ bits of the subpath p between $x(X_3)$ and $maxcov(X_2, Y, e)$ for any level-j cluster Y in $TT_j \setminus X_2$. We remove from p all but one representative of the shared vertices of the endpoints of p. We set the *somecov*$_2$ bits of this subpath.

(4) In Case D, the partial path of X_1 is empty. The complete path of X_1 consists of the partial path of X_2 unioned with the partial path of X_3. We set the *somecov*$_1$ bits of the subpath p between $maxcov(X_2, X_3, e)$ and $maxcov(X_3, X_2, e)$. We remove from p all but one representative of the shared vertices of the endpoints of p. This results in a subpath p'. If e is solid or if e is dashed, belonging to the shared vertex s, and $shared(X_2, X_3, s)$ is 1, then we also set the *somecov*$_2$ bits of p'. If e is dashed and $shared(X_2, X_3, s)$ is 0, then we split p' at the representatives of s and remove all but one representative of s from each of the resulting subpaths. We set the *somecov*$_2$ bits for these subpaths.

Given the *maxcov* and *shared* labels, the above description also is an algorithm

to build the partial paths of X_1 from the partial paths of the children of X_1 in time $O(\log n)$ and the complete path of a level-j cluster in time linear in the number of level-j nodes in TT.

Next we show how to use the $somecov_i$ bits to test the avoidability of a node e. We start with the $somecov_1$ bits.

LEMMA 2.33. *Let P^c be a complete path and let D, D', and C be basic clusters such that $x(C), x(D)$, and $x(D')$ belong to P^c. Then $somecov_1(v)$ is set for an ancestor v of $x(C)$ in P^c iff there exists a non-tree edge between $ST(C, D)$ and $ST(C, D')$.*

Proof. Consider the lowest level node X_1 such that the partial or complete path P^x of X_1 contains $x(c), x(D)$, and $x(D')$. Let $j+1$ be the lowest level of X_1 at which a $somecov_1$ bit is set for an ancestor v of $x(C)$. Then X_1 has two children X_2 and X_3 in TT, connected by an edge e in TT_j. Without loss of generality (w.l.o.g.) $x(C)$ is a node of the partial path of X_2. Then $somecov_1(v)$ is set in the path of X_1 because $x(C)$ is an internal node of the subpath of P^x between $maxcov(X_2, X_3, e)$ and the first node of X_3 on P^x. By the definition of $maxcov$ there exists an edge between $ST(C, D)$ and $ST(C, D')$.

Assume next that an edge exists between $ST(C, D)$ and $ST(C, D')$. Let X_1 be the least common ancestor of D and D' in TT and let X_2 and X_3 be its two children. W.l.o.g. the partial path of X_2 contains $x(C)$ and $x(D)$. Since there exists a non-tree edge between X_2 and X_3 whose projection is $x(D)$, $x(C)$ lies between $maxcov(X_2, X_3, e)$ and the first node of X_3 on the partial or complete path P^x of X_1. Thus the $somecov_1$ bit is set for an ancestor of $x(C)$ in the partial or complete path of X_1 and, hence, also in P^c. □

Next we discuss under which conditions $somecov_2(v)$ is set for an ancestor v of $x(C)$.

LEMMA 2.34. *Let C be a basic cluster and let P^c be a complete path containing C. If C is not connected in T_1 to its neighbors in P^c by a dashed edge of P^c, then $somecov_2(v)$ is set for an ancestor v of $x(C)$ in P^c iff $somecov_1(v)$ is set.*

Proof. The lemma follows immediately from the definition of $somecov_2$. □

LEMMA 2.35. *Let P^c be a complete path and let $C_1, \ldots, C_l$ be basic clusters such that $x(C_1), \ldots, x(C_l)$ forms a maximal subpath of P^c sharing the same vertex s. Let j be the lowest level such that the $somecov_2$ bits are set for an ancestor for every node $x(C_1), \ldots, x(C_l)$. Then all nodes $x(C_1), \ldots, x(C_l)$ belong to the partial or complete path of the same cluster at level j.*

Proof. Let P^c be stored at a level j^* node. The claim obviously holds for level j^*. Assume it does not hold for a level $j < j^*$. Then there exists at least one node $x(C_q)$ that is an endpoint of its partial path on level j and in this partial path there exists an ancestor of $x(C_q)$ whose $somecov_2$ bit is set. Note that $x(C_q)$ was the endpoint of this partial path for every level $\leq j$. Note further that the $somecov_2$ bit is never set to 1 for an ancestor of an endpoint of a partial path. Thus at level j, the $somecov_2$ bit is not set for any ancestor of $x(C_q)$. This is a contradiction. □

LEMMA 2.36. *Let P^c be a complete path and let D, D', and $C_1, C_2, \ldots, C_l$ be basic clusters such that $x(D), x(C_1), x(C_2), \ldots, x(C_l), x(D')$ is a subpath of P^c and $x(C_1), x(C_2), \ldots, x(C_l)$ forms a maximal subpath of P^c sharing the same vertex. Then, for all $1 \leq q \leq l$, $somecov_2(v)$ is set for an ancestor v of $x(C_q)$ in P^c iff there exists a non-tree edge between $ST(C_1, D)$ and $ST(C_l, D')$.*

Proof. Let s be the vertex shared by the cluster C_q to which the dashed edges incident to C_q belong. Consider the lowest level $j+1$ at which, for all $1 \leq q \leq l$, $somecov_2(v_q)$ is set for an ancestor v_q of $x(C_q)$. By Lemma 2.35, all $x(C_q)$ belong to

the partial or complete path P^x of the same level-$(j+1)$ cluster X_1. Then X_1 has two children X_2 and X_3 connected by an edge e in TT_j. Note that one of the $somecov_2(v_q)$ bits was set while constructing P^x.

If neither X_2 nor X_3 has tree degree 3, a $somecov_2(x(C_q))$ bit was set when constructing the path for X_1 because either (1) $x(D), x(D')$, and all nodes $x(C_q)$ are nodes on the path between $maxcov(X_2, X_3, e)$ and $maxcov(X_3, X_2, e)$, and e does not belong to s, or (2) there exists an edge between X_2 and X_3 whose projections do not belong to s. In either case there exists a non-tree edge between $ST(D, C_1)$ and $ST(C_l, D')$.

If X_2 has tree degree 1 and X_3 has tree degree 3, a $somecov_2(x(C_q))$ bit is set while constructing the path for X_1 only if all nodes $x(C_q)$ are internal nodes on the path between $maxcov(X_2, Y, e)$ and $x(X_3)$ for a level-j cluster Y in $TT_j \setminus X_2$. It follows that there is a non-tree edge between $ST(C_1, D)$ and $ST(C_l, D')$.

Assume next that an edge exists between $ST(C_1, D)$ and $ST(C_l, D')$. Let X_1 be the least common ancestor of D and D' and let X_2 and X_3 be its two children. If neither X_2 nor X_3 has tree degree 3, we consider two cases. If the tree edge e between X_2 and X_3 does not belong to s, the $somecov_2$ bits of the subpath $x(C_1), \ldots, x(C_l)$ including endpoints are set because the tree edge lies between $maxcov(X_2, X_3, e)$ and $maxcov(X_3, X_2, e)$. If e belongs to s, the $somecov_2$ bits of the subpath are set because $shared(X_2, X_3, s)$ is 1.

If X_2 has tree degree 1 and X_3 has tree degree 3, then $x(X_3) = x(D')$ or $x(X_3) = x(D)$, i.e., e is solid. The subpath $x(C_1), \ldots, x(C_l)$ is internal to the path between $maxcov(X_2, Y, e)$ and $x(X_3)$ for some level-j cluster Y in $TT_j \setminus X_2$. Thus, the $somecov_2$ bits of the subpath $x(C_1), \ldots, x(C_l)$ including endpoints are set. □

LEMMA 2.37. *Let C be an internal node on the increasing path Q in T_i and let D and D' be the neighbors of C on Q such that D is the child and D' is the parent of C.*

(1) *For $i = 1$, if a complete path P^c exists to which $x(D')$, $x(C)$, and $x(D)$ belong, then $somecov_1(v)$ is set to 1 for an ancestor v of $x(C)$ in P^c iff C is avoidable on Q.*

(2) *For $i = 2$, let $C_1, \ldots, C_l$ form an increasing subpath of Q_{T_1} with $C_1 = C(Q_{T_1})$ and $C_l = C(Q_{T_1})'$. If a complete path exists that contains $x(D')$, $x(D)$, and $x(C_q)$ for all $1 \le q \le l$, then for all C_q in P^c $somecov_2(v_q)$ is set to 1 for an ancestor v_q of $x(C_q)$ iff C is avoidable on Q.*

Proof. Let P^c be C's complete path. We discuss first the case $i = 1$. Lemma 2.33 shows that $somecov_1$ is set for an ancestor of $x(C)$ iff there is a non-tree edge between $ST(C, D)$ and $ST(C, D')$. By the definition of avoidability, the latter holds iff C is avoidable on Q.

Next we discuss the case $i = 2$. Lemma 2.34 shows the claim if $l = 1$. If $l > 1$, then C represents at least two clusters in H_1. Lemma 2.36 shows that for all C_q in P^c $somecov_2(v_q)$ is set to 1 for an ancestor v_q of $x(C_q)$ iff there exists a non-tree edge between $ST(C_1, D)$ and $ST(C_l, D')$. The latter holds iff C is avoidable on Q. □

LEMMA 2.38. *The extended ambivalent data structure can determine the avoidability of all but $O(\log n)$ nodes on a path P in H in time $O(m/k)$.*

Updates. Next we show how to maintain the *maxcov* and *shared* values and the partial and complete paths in time $O(k)$ after each update operation in G. First, we discuss the *maxcov* and *shared* values. By definition an update (u, v) operation affects only the $maxcov(X, Y, e)$ and $shared(X, Y, s)$ values iff either X or Y contains either u, v, x, or y, where (x, y) is the new tree edge. Consider the subtree of $2TT$ induced by marking all nodes $A \times B$ such that A and B contain either u, v, x, or y.

Since this subtree forms a structure which is isomorphic to two copies of TT, the total number of affected *maxcov* and *shared* values is $O(m/k)$. As was shown in [6] for the *maxcov* values and as we show below for the *shared* values each such value at an internal node of $2TT$ can be computed in constant time from the values of its children and in time $O(k)$ for a basic cluster. Thus, updating all *maxcov* and all *shared* values takes time $O(k)$.

Lemma 2.39 shows that the only clusters whose partial or complete paths are affected by updates are the ones that are ancestors in TT of the basic cluster containing u, v, x, or y. Thus, the partial and complete paths of at most two clusters at each level have to be updated. We discuss below how to restore the partial and complete paths of the clusters that do not contain u, v, x, or y, but are children of clusters containing u, v, x, or y. The partial path of a cluster C can be computed in time $O(\log n)$ from the partial paths of the children of C and the *shared* and *maxcover* values. The complete path of a cluster C at level j can be computed in time linear in the number of level j nodes in TT from the partial path of the child of C with tree degree 1 and from the *shared* and *maxcover* values. Since TT has depth $O(\log n)$ and size $O(m/k)$, all affected partial and complete paths can be updated in time $O(m/k + \log^2 n) = O(m/k)$ for $k \leq m/\log^2 n$.

Whenever we build the partial or complete path we keep a *back log* that stores for each node X_1 of $2TT$ all the operations that were executed to build the partial or complete path of X_1 from the partial or complete path of its children. Whenever we execute an update operation, we walk top-down in TT and restore the path of the suitable clusters and their children. The partial and complete path of the root of TT are given. Assume inductively the partial and complete path of a node X at level j are restored. Undo the operations in the back log of X to restore the partial and complete paths of the children of X. Then recurse on the suitable child(ren) of X. The same argument as above shows that this takes time $O(m/k)$. Note also that modifications in the back log of one child does not affect the back log of its sibling.

LEMMA 2.39. *An insert*(u, v) *and a delete*(u, v) *operation only modifies the balanced tree of partial or complete paths of clusters containing* u, v, x, *or* y, *where* (x, y) *is a new tree edge.*

Proof. A *somecov* bit is set at a node in the balanced tree representing the partial path of a cluster C only if there exists an edge internal to C that covers the corresponding nodes. For a cluster not containing u, v, x, or y neither the partial path nor the non-tree edges internal to the cluster have changed. Thus, the balanced search tree of its partial path does not have to be updated.

Next we discuss complete paths. If a cluster C which has a complete path does not contain u, v, x, or y, then the partial path of its child C' with tree degree 1 and the non-tree edges incident to C' are not affected by the above argument. The modifications to this partial path that create the data structure for the complete path of C depend only on the projection of edges incident to C' onto the partial path of C'. Since the partial path of C' and the non-tree edges incident to C' did not change, the balanced search tree of the complete path of C is not affected by the operation. □

We are left with showing how to compute *shared*(A_1, B_1, s) from the *shared* values of the children of A_1 and B_1 and information stored at their children in constant time. Recall that for each pair of clusters at the same level *shared*(A_1, B_1, s) is 1 iff A_1 shares s and there is an edge between A_1 and B_1 whose projection onto the partial path of A_1 and onto the partial path of B_1 is not s. If A_1 does not share a vertex s,

then $shared(A_1, B_1, s)$ is not defined. Since each basic cluster and each cluster with tree degree 1 or 3 shares at most one vertex and each nonbasic cluster with tree degree 2 shares at most two vertices, at most four $shared(A_1, B_1, .)$ values are defined for every pair of clusters A_1 and B_1. We distinguish cases depending on the number of children of A_1 and of B_1 under the assumption that A_1 shares the vertex s.

Case 1: A_1 and B_1 are basic clusters.

Then $shared(A_1, B_1, s) = 0$, since every edge incident to A_1 is projected onto s when it is projected onto the partial path of A_1.

Case 2: A_1 and B_1 are clusters at level $j > 0$.

Case 2.1: A_1 has one child A_2 and B_1 has one child B_2.

Then $shared(A_1, B_1, s) = shared(A_2, B_2, s)$.

Case 2.2: A_1 has one child A_2 and B_1 has two children B_2 and B_3.

If neither B_2 nor B_3 has tree degree 3, then $shared(A_1, B_1, s) = shared(A_2, B_2, s)$ *or* $shared(A_2, B_3, s)$.

If the tree degree of B_3 is 3 and the tree degree of B_2 is 1, then $shared(A_1, B_1, s) = shared(A_2, B_2, s)$ if B_3 does not share s and 0 if B_3 shares s.

Case 2.3: A_1 has two children A_2 and A_3, A_3 shares s, and B_1 has one child B_2.

W.l.o.g. A_3 is incident to the tree edge incident to A_1. Thus, A_3 has tree degree at least 2. If the tree degree of A_3 is 3, then $shared(A_1, B_1, s) = 0$, since every edge incident to A_1 is projected onto s when it is projected onto the partial path of A_1.

If the tree degree of A_3 is 2, then we distinguish between the cases that A_2 shares s and that A_2 does not share s. If A_2 shares s, then

$$shared(A_1, B_1, s) = shared(A_2, B_2, s),$$

since the projection of every edge incident to A_3 onto the partial path of A_1 is s.

If A_2 does not share s, then we distinguish between the cases that B_2 shares s and that B_2 does not share s. If B_2 shares s, then

$$shared(A_1, B_1, s) = shared(A_3, B_2, s) \, or \, shared(B_2, A_2, s).$$

(Note that $shared(A_2, B_2, s)$ is not defined in this case, but $shared(B_2, A_2, s)$ is defined.)

If B_2 does not share s, then

$$shared(A_1, B_1, s) = shared(A_3, B_2, s) \, or \, edge(A_2, B_2),$$

where $edge(A_2, B_2) = 1$ iff there exists an edge between A_2 and B_2 iff $maxcov(A_2, B_2, e)$ (for any tree edge e incident to A_2) is defined.

Case 2.4: A_1 has two children A_2 and A_3 and B_1 has two children B_2 and B_3.

W.l.o.g. A_3 is incident to the tree edge incident to A_1. Thus, A_3 has tree degree at least 2. If the tree degree of A_3 is 3, then $shared(A_1, B_1, s) = 0$, since the projection of every non-tree edge incident to A_1 onto the partial path of A_1 is s.

If the tree degree of A_3 is 2, then we distinguish between the cases that A_2 shares s and that A_2 does not share s. If A_2 shares s, then

$$shared(A_1, B_1, s) = shared(A_2, B_2, s) \, or \, shared(A_2, B_3, s),$$

since the projection of every non-tree edge incident to A_3 onto the partial path of A_1 is s.

If A_2 does not share s, then we distinguish between the case that (1) B_2 shares s and B_3 does not share s, that (2) B_3 shares s and B_2 does not share s, that (3) both share s, and that (4) both do not share s.

In case (1) (B_2 shares s and B_3 does not share s)

$$\begin{aligned} shared(A_1, B_1, s) = {} & shared(A_3, B_2, s) or\ shared(A_3, B_3, s) or \\ & shared(B_2, A_2, s) or\ edge(A_2, B_3). \end{aligned}$$

(Note that $shared(A_2, B_2, s)$ is not defined in this case, but $shared(B_2, A_2, s)$ is defined.) The case (2) is symmetric to case (1).

In case (3) (B_2 and B_3 share s)

$$\begin{aligned} shared(A_1, B_1, s) = {} & shared(A_3, B_2, s) or\ shared(A_3, B_3, s) or \\ & shared(B_2, A_2, s) or\ shared(B_3, A_2, s). \end{aligned}$$

In case (4) (B_2 and B_3 do not share s)

$$\begin{aligned} & rclshared(A_1, B_1, s) \\ & \quad = shared(A_3, B_2, s) or\ shared(A_3, B_3, s) or\ edge(A_2, B_2) or\ edge(A_2, B_3). \end{aligned}$$

This shows that the $shared(A_1, B_1, .)$ bit can be computed in constant time from the *shared* and *maxcov* values of the children of A_1 and B_1 and it finishes the proof of the following lemma.

LEMMA 2.40. *The extended ambivalent data structure can be updated in time* $O(k)$.

2.8. The c-structure. In this section we address the following problem. Given the c-nodes of a cluster graph (see section 2.5) or the c-nodes of a shared graph $G(s)$ for a new shared vertex s (see section 2.6.1), determine which c-nodes are CV-split by an update operation. For this problem we give in this subsection a data structure, called *c-structure* and show in the next subsection that in a phase $O(m/k)$ splits of c-node occur because of violation of condition (a) or (b) (called *CV-splits*). Additionally we show that the e-nodes of a shared graph $\tilde{G}(s)$ for an old shared vertex s (see section 2.6.2) fulfill the conditions of a c-node and therefore the same data structure and proof apply.

First we recall the definitions of c-node and e-node; next we show that an e-node also fulfills the conditions of a c-node, and then we define the c-structure exactly .

Let $H = H_1$ for c-nodes in cluster graphs and $H = H_2$ for c-nodes in shared graphs. Given a node C in H with ancestor A a c-node represents a maximal set X of nodes of H such that

(a) every node $C' \in X$ is a neighbor of C,
(b) all nodes in X are connected in $H \setminus C$,
(c) all nodes in X have the same ancestor which is different from A,
(d) at the creation of C, the set X contains only one node, and
(e) at each previous point in time since the creation of C all nodes of H that contain the vertices in $\cup_{C' \in X} C'$ existing at this time are represented by the same c-node.

We show next that every e-node in the graph $\tilde{G}(s)$ of an old shared vertex s fulfills the conditions of a c-node with $H = H_2$. Thus the amortization lemma of the next section also applies to e-nodes.

LEMMA 2.41. *Every e-node in the graph $\tilde{G}(s)$ of an old shared vertex s fulfills the conditions of a c-node with $H = H_2$.*

Proof. By definition every e-node fulfills conditions (a)–(d) of a c-node. We only have to show that it also fulfills condition (e).

Let X be an e-node. By definition the vertices in $\cup_{C' \in X} C'$ used to belong to the same node C_{old} of H_2 and since the split of C_{old}, the graph $G(s)$ was not modified and the connected components of $H_2 \setminus C_s$ did not change. Thus, before the last change in $G(s)$ or $H_2 \setminus C_s$ the nodes in X belonged to the same node of H_2 and thus were represented by the same e-node.

Assume by contradiction that the nodes in X were represented by different e-nodes at some point since the last change in $G(s)$ or $H_2 \setminus C_s$. While the nodes were represented by different e-nodes they either must violate condition (a) or (b) of an e-node, since conditions (c), (d), and (e) continue to hold. However, now they fulfill conditions (a) and (b), i.e., either $G(s)$ or $H_2 \setminus C_s$ must have changed, which is a contradiction. ∎

Thus, e-nodes are just a special case of c-nodes and we will just use the term c-node in the following to denote c-nodes as well as e-nodes.

A c-node of C is *CV-split* iff conditions (a) or (b) of a c-node are no longer fulfilled.

Given the high-level graph H and its data structures the c-structure maintains the c-nodes of each node in H under the following operations:

(1) *c-split* (C, C_1, C_2, u, v), where C is a node of H split by the *delete*(u, v) or *insert*(u, v) operation, C_1 and C_2 are the two nodes of H created by the split. Split the node C into C_1 and C_2.

(2) *c-add* (C_1, C_2), where C_1 and C_2 are nodes of H. Add one edge between C_1 and C_2.

(3) *c-remove* (C_1, C_2), where C_1 and C_2 are nodes of H. Remove the edge between C_1 and C_2 and return a (possibly empty) list of CV-split c-nodes and for each newly created c-node return its element list.

We use the following data structure for the c-structure, which uses $O((m/k)^2)$ space.

(T1) For each node of H we keep a list of its c-nodes. For each c-node we keep a list of its elements. For each node in H we keep a list of all the c-nodes it belongs to. The position of the node in the list of the c-node and the position of the c-node in the list of the node point to each other.

We keep two c-structures, namely, one with $H = H_1$ to determine the CV-splits in c-nodes of the cluster graphs, and one with $H = H_2$ to determine the CV-splits in c-nodes of the shared graph for new shared vertices.

2.8.1. Implementing the c-structure. We implement the operations as follows.

c-split (C, C_1, C_2, u, v): This requires (i) updating the c-nodes of C and (ii) updating the c-nodes containing C. (i) Discard all c-nodes of C. Each neighbor of C_1 (resp., C_2) forms a 1-element c-node for C_1 (resp., C_2). (ii) Use (T1) to determine all c-nodes X to which C belongs. Replace C by either C_1 or C_2 or both in X, depending on which of the new nodes are incident to D. Note that all nodes in X still fulfill (a), (c), (d), and (e) of a c-node. By Lemma 2.8 all nodes continue to fulfill (b) as well.

c-add (C_1, C_2): If C_1 and C_2 have the same ancestor, do nothing. Otherwise, search the c-nodes to which C_1 belongs to determine whether C_2 is one of them. If not, add to the c-nodes of C_2 a 1-element c-node consisting of C_1. Repeat with the roles of C_1 and C_2 exchanged.

c-remove (C_1, C_2): If C_1 and C_2 have the same ancestor, do nothing. Otherwise, determine the c-node of C_2 to which C_1 belongs and remove C_1 from it. If this c-node becomes empty, discard it. If the c-node is modified, output it and its new element list. Repeat with the roles of C_1 and C_2 exchanged. Finally determine all articulation points D on $\pi(C_1, C_2)$ in the (updated) graph H using (HL4). Test as follows for each c-node X of D whether (b) is violated, and if so, how to partition X. Using (HL3) determine for each node C' in X the tree neighbor of D on $\pi(C', D)$ and bucketsort the node according to the blockid of "its" tree neighbor using (HL4) and (HL6). This results in either one or two nonempty buckets. In the former case (b) is not violated. In the latter case, split the list of X according to the two buckets and report the CV-split of X and return the two resulting lists.

To analyze the running time note that the intersection of two different c-nodes of C is empty. The time spent by a *c-split* or *c-add* operation is linear in the number of c-nodes of a node in H, which is $O(m/k)$. In *c-remove* we spend the time $O(m/k)$ to determine all articulation points and then the following time per articulation point D: $O(\log n)$ per non-tree neighbor of D to bucketsort it and constant time to remove it from the bucket again. Since the nodes D are articulation points on a path in H, this sums up to $O((m/k)\log n)$ by Lemma 2.1. Additionally the *c-remove* operation spends time $O(m/k)$ to update the c-nodes of C_1 and the c-nodes of C_2. Thus, the total time spent is $O((m/k)\log n)$.

2.8.2. Updating the c-structure. At the beginning of a phase each c-node consists of one node: every neighbor of a node in H forms its own c-node.

Whenever an edge is inserted in G and H changes, then first the data structures of H are updated and then a *c-add* and potentially afterward a constant number of *c-split*'s are executed in the c-structure. Whenever an edge is deleted from G and H changes, then first the data structures of H are updated and then potentially a constant number of *c-split*'s and afterward a *c-remove* are executed in the c-structure. If an internal tree edge of a cluster C is deleted, then this implies that first the cluster is split at this tree edge and afterward the tree edge is deleted. Each operation can be implemented in time $O((m/k)\log n)$. Since there are only a constant number of them per update in G, this gives a total time of $O((m/k)\log n)$ to update the c-structure.

THEOREM 2.42. *The c-structure*

(1) *can be updated in time* $O((m/k)\log n)$ *after each update in* G, *and returns all the c-nodes CV-split by the update and the resulting c-nodes, and*

(2) *can be built in time* $O(m)$.

2.9. The amortization lemma. We show next that during a sequence of l updates in a phase $O(l)$ CV-splits of c-nodes occur. A similar, but less general lemma was shown in [11].

LEMMA 2.43. *During* l *updates in* G *in a phase at most* $2l$ *CV-splits of a c-node occur because of violation of condition* (a).

Proof. Condition (a) is violated if a node C' belongs to a c-node of node C, but C' is no longer incident to C. This is only possible if C as well as C' contains an endpoint of the update edge. Since all c-nodes of C are disjoint, condition (a) is violated for at most one c-node of C. Similarly, condition (a) is violated for at most one c-node of C' and for no c-nodes at other nodes. □

LEMMA 2.44. *During* l *updates in* G *in a phase* $O(l + m/k)$ *CV-splits occur because of the violation of condition* (b) *for* $l \geq m/k$.

Proof. We construct a bipartite graph K consisting initially of $O(m/k)$ blue nodes, $O(m/k)$ red nodes, and $O(m/k)$ edges between red and blue nodes. A red node

is incident to at least one blue node. We show that during a sequence of l updates in G the number of blue nodes in K increases by $O(l)$ (Proposition 2.45), each CV-split of a c-node increases the number of connected components of K by at least one (Proposition 2.46), and no other operation decreases the number of components (Proposition 2.51). Thus, there are at most $O(l + m/k)$ splits of c-nodes during l updates in G.

The proof will exploit the following fact: Whenever a c-node at C containing D_1 and D_2 is CV-split, then there is no c-node at another node C' that contains both D_1 and D_2. (Otherwise the path "through" C' would connect D_1 and D_2 in $H \setminus C$.) Thus, the split discards the last common c-node of D_1 and D_2. We will show that an even stronger property holds: Assume the relation $r(D_1, D_2)$ holds iff D_1 and D_2 have a common c-node. Let r^* be the transitive closure of r. Then whenever the c-node containing D_1 and D_2 is split, then $r^*(D_1, D_2)$ does not hold after the split. The graph K is constructed so that this fact implies that the connected components of K increase.

Recall that an edge of H consists of a set of edges of G'. An edge of H is called *new* if all edges of G' in its set are new, i.e., have been inserted after the last rebuild. All other edges of H are *old.* We "treat" new edges in a special way to guarantee that edge insertions do not decrease the number of connected components of K. Note that there are at most l new edges in H at each point in time.

We next define K.

(1) For each node C in H and each c-node X at C, K contains a red node (C, X).

(2) For each node D in H, K contains a blue node D.

(3) For each node D in a c-node X at C such that (D, C) is new there exists a blue node (D, X). These nodes are called *special.*

(4) Let D be in the c-node X at C. If both a red node (C, X) and a blue node (D, X) exist, there exists an edge between (C, X) and (D, X). If (D, X) does not exist, there is an edge between (C, X) and D.

By abuse of notation we will equate a blue node in K with the node of H represented by the blue node. Note that every edge in K corresponds to an edge of H. Thus, if two blue nodes are connected in K, their nodes are connected in H.

There are four events that modify K: (A) a *c-split* operation, (B) a *c-add* operation, (C) a *c-remove* operation, and (D) the change of an edge from old to new.

Next we describe each event in detail:

(A) A *c-split (C, C_1, C_2,u,v)* operation. (1) Every red node (C, X) is removed. For each $D \in X$ incident to C_1 we create a red node $(C_1, \{D\})$, and if the blue node (D, X) exists, it is replaced by a blue node $(D, \{D\})$. The new red node is connected to $(D, \{D\})$ if it exists and to D otherwise. We proceed in the same way with C_2. (2) The blue node C and all blue nodes (C, X) are split into two nodes and connected to the appropriate neighbors of the split nodes.

(B) A *c-add (C_1, C_2)* operation. It might add a new red node at $(C_1, \{C_2\})$, a new blue node $(C_2, \{C_2\})$, and connect them by an edge. It might do the same with the roles of C_1 and C_2 reversed.

(C) A *c-remove (C_1, C_2)* operation. Let X be the c-node at C_1 containing C_2. Remove the edge between the red node (C_1, X) and the corresponding blue node representing C_2. If the blue node (C_2, X) exists, remove it. If $X = \{C_2\}$, also remove (C_1, X). Proceed in the same way with the roles of C_1 and C_2 reversed. Finally for each split c-node X' of an articulation point D on

$\pi(C_1, C_2)$ in H replace the red node (D, X') by two red nodes, one for each new c-node and connect their blue neighbors suitably.

(D) An old edge (D, C) of H becomes new. If D belongs to the c-node X at C, then add a new blue node (D, X) with edge to (C, X) and remove the edge from D to (C, X). Then proceed in the same way with the roles of D and C reversed.

We prove next the three missing propositions.

PROPOSITION 2.45. *During l update operations the number of blue nodes increases by $O(l)$.*

Proof. A sequence of l update operations in G leads to at most $7l$ *c-split* operations, l *c-add* operations, and l *c-remove* operations. At each point there are at most $2l$ special blue nodes. Next we bound the number of nonspecial blue nodes. A *c-add* operation, a *c-remove* operation, and the change of an edge from old to new do not increase the number nonspecial blue nodes. A *c-split* increases the number of nonspecial blue nodes by at most 1. Thus, the number of nonspecial blue nodes increases by $O(l)$. □

Next we show that the CV-split of a c-node increases the number of connected components by at least 1.

PROPOSITION 2.46. *A CV-split of a c-node increases the number of connected components by at least* 1.

Proof. Consider the split of the c-node X at node C'. A c-node is CV-split only during a *c-remove* operation. So consider the removal of edge (C_1, C_2). Let $\{X_1, X_2, \ldots, X_p\}$ be all the c-nodes that are CV-split at C'. Let D_1 and D_2 be the tree neighbors of C' on $\pi(C_1, C_2)$ and let Y_1 and Y_2 be the blue nodes representing D_1 and D_2 and incident to the red node (C', X_i) for some $1 \leq i \leq p$ in K. To update K, each red node (C', X_j) is replaced by two new red nodes, one representing each new c-node. Each (blue) neighbor of (C', X_j) is connected to exactly one of the new c-nodes depending on which new c-node it belongs to. Obviously, D_1 and D_2 are connected to different new red nodes. As we show in Proposition 2.50 after the *c-remove* operation there exists no path in K anymore between D_1 and D_2, i.e., the number of connected components in K has increased by at least 1. □

We are left with proving Proposition 2.50 and showing that the number of connected components of K does never decrease. We first need some intermediate results.

PROPOSITION 2.47. *Every blue node (D, X) has degree* 1 *in K.*

Proof. Let X be a c-node of node C of H. By definition of K, a blue node (D, X) can only be adjacent to node (C, X). □

PROPOSITION 2.48. *Let C be a node in H. Each blue node representing a node D in H is incident to at most one red node (C, X), and X is the c-node to which D belongs at C.*

Proof. The proof follows from the construction of K since each node D belongs to at most one c-node at C. □

PROPOSITION 2.49. *If two blue nodes are connected in K, then they represent nodes with the same ancestor.*

Proof. If there is a path in K between the two blue nodes Y and Y', then let $B_1, \ldots, B_j$ be the blue nodes on this path with $Y = B_1$ and $Y' = B_j$. Since B_i and B_{i+1} are adjacent to the same red node, they belong to the same c-node at that node and thus have the same ancestor. By the transitivity of the ancestor relation the claim follows. □

PROPOSITION 2.50. *Consider the operation c-remove(C_1, C_2). Let $\{X_1, X_2, \ldots,$*

$X_p\}$ be all the c-nodes that are CV-split at a node C' of H. Let D_1 and D_2 be the tree neighbors of C' on $\pi(C_1, C_2)$. Let Y_1 and Y_2 be the blue nodes representing D_1 and D_2 that are incident to a red node (C', X_i) for some $1 \leq i \leq p$. Then after the c-remove operation no path exists connecting Y_1 and Y_2.

Proof. Consider first the case that either (D_1, C') or (D_2, C') is new. By Proposition 2.47 every path between Y_1 and Y_2 contains (C', X_i) and hence is disconnected after the remove operation.

Assume next that both edges are old, i.e., $Y_1 = D_1$ and $Y_2 = D_2$, and assume that a path P exists between them after the *c-remove* operation. Since the c-node of D_1 and D_2 was CV-split, after the deletion of (C_1, C_2) every path in H connecting D_1 with D_2 in H contains C'. We will show that the existence of P implies the existence of a path in $H \setminus C'$ connecting D_1 and D_2, which gives the contradiction.

For this we show (1) that no blue node representing C' belongs to P, and (2) that the blue nodes incident to a red node (C', X') on P are connected in $H \setminus C'$ after the update.

(1) Since D_1 and D_2 belonged to a c-node at C', their ancestor differs from the ancestor of C'. By Proposition 2.49 the nodes of H represented by the blue nodes of P all have the same ancestor. Thus, no blue node representing C' belongs to P.

(2) Let F_k and F'_k be the two nodes incident on P to the kth red node (C', X') for some c-node X'. Then F_k and F'_k both belong to the same c-node X'. It follows that F_k and F'_k are connected in $H \setminus C'$ after the deletion of edge (C_1, C_2).

From (1) it follows that P forms a path without a blue node representing C': (2) shows that every red node on P representing C' can be avoided by a path in $H \setminus C'$. Let l be the number of red nodes representing C' on P. Note that the subpaths of P between F'_k and F_{k+1}, the subpath from D_1 to F_1, and the subpath from F'_l to D_2 contain no edge incident to C' and thus correspond to paths in $H \setminus C'$. It follows that P induces a path in $H \setminus C'$ between D_1 and D_2 after the deletion of (D_1, D_2), which is a contradiction. □

PROPOSITION 2.51. *The number of connected components of K never decreases.*

Proof. As shown in Proposition 2.46 a *c-remove* operation does not decrease the number of connected components.

A *c-split* operation consists of two parts. In part 1 the red node (C, X) is replaced by many red nodes, each being connected at most to all the nodes that (C, X) was connected to. This does not decrease the number of connected components. In part 2 the blue node C and all blue nodes (C, X) are each split into two new nodes such that each new node is connected to at most all the nodes that the original blue node was connected to. So again, the number of connected components is not decreased.

A *c-add* (C_1, C_2) operation might add a new blue node (C_2, X), a new red node (C_1, X), an edge between them, and the same with the roles of C_1 and C_2 reversed. Since they do not connect to the rest of K, an *add* operation does not decrease the number of connected components in K either.

Note that an old edge (D, C) of H can become new, but not vice versa. If this happens an edge is removed from K, a new blue node (D, X) is added and is connected to (C, X), where X is the c-node of C to which D belongs. The same happens with the roles of D and C reversed. Thus the number of connected components does not decrease. □

This completes the proof of the lemma. □

2.10. 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 the blocks to which the tree edges adjacent to the vertex belong. We can find the blocks in $I(C)$ for every tree edge internal or incident to the cluster C in time $O(k)$ whenever we recompute $I(C)$. To compute all the blocks in G, we have to determine which blocks of different cluster graphs form the same block of G, i.e., have to be combined.

Perform a depth-first traversal of the spanning tree T_2 of H_2. For each tree edge $e = (u, v)$ with $u \in C_1$ and $v \in C_2$ such that u is a parent of v in the (rooted) depth-first search (dfs) tree, test for each tree edge (x, w) with $x \in C_2$ and $w \in C_3$ whether u and w are biconnected: if C_2 is also a node of H_1, then test whether the shared vertex of C_2 separates u and w and if not use the internal data structure of C_2 to test the biconnectivity of u and w in G. If C_2 is no node of H_1, then $v = x$ is a shared vertex. In this case test the biconnectivity of u and w using the shared graph of v. If we recursively know all tree edges of T_2 in the dfs subtree of edge (x, w) that belong to the same block as (x, w), then we can construct for (u, v) the set of all tree edges of T_2 in the dfs subtree of (u, v) that belong to the same block as (u, v). The dfs takes time $O(m/k)$.

When the dfs is completed we combine the blocks of all the tree edges in the same set and mark all the edges in T' accordingly. Thus the total cost is proportional to the number of tree edges in T', which is $n - 1$.

THEOREM 2.52. *A complete block query in a graph of n vertices can be answered in time $O(n)$.*

2.11. Biconnectivity queries. Given a *query*(u, v) operation, let $x^{(i)}$ and $y^{(i)}$ be defined as in Lemma 2.7. The lemma shows that u and v are biconnected in G iff

(Q1) u and $y^{(1)}$ are biconnected in G,
(Q2) $x^{(i)}$ and $y^{(i+1)}$ are biconnected in G, for all $1 \leq i < p$, and
(Q3) $x^{(p)}$ and v are biconnected in G.

Condition (Q2) holds iff $e_i = (x^{(i)}, y^{(i)})$ and $e_{i+1} = (x^{(i+1)}, y^{(i+1)})$ belong to the same block of G for all $1 \leq i < p$. This is equivalent to the requirement that e_1 belongs to the same block as e_p. Thus, it suffices to determine and test e_1 and e_p and to test (Q1) and (Q3).

By definition all edges e_i are solid intercluster tree edges, i.e., tree edges on the T_2-path between the node C_u in H_2 and the node C_v in H_2. Therefore, we keep the following data structure.

(HL9) We store a least common ancestor data structure [10] for T_2 rooted at a leaf R, such that least common ancestor queries between any two nodes of H_2 can be answered in constant time. If C is the least common ancestor of C and D, then the data structure also returns in constant time the tree edge, incident to C on $\pi(C, D)$. We also keep at each node of H_2 the tree edge to its parent.
(HL10) We store at each solid intercluster tree edge its block in G.

Both data structures are recomputed from scratch after each update in G. The computation of (HL10) proceeds in the same way as a complete block query: we perform a dfs on T_2 that determines the sets of solid intercluster tree edges that belong to the same block. This takes time $O(m/k)$. The time to build both data structures is thus $O(m/k)$.

Determining e_1, e_p, $y^{(1)}$, and $x^{(p)}$: We first use (HL9) to determine the least common ancestor of C_u and C_v in H_2. If C_u is the least common ancestor, the data structure returns the tree edge incident to C_u on $\pi(C_u, C_v)$. This is edge e_1; the edge from C_v to its parent is the edge e_p. If the least common ancestor of C_u and C_v is a

third node, then e_1 is the edge from C_v to its parent and e_p is the edge from C_u to its parent. This also provides $y^{(1)}$ and $x^{(p)}$.

Testing conditions (Q1) *and* (Q3): We test conditions (Q1) and (Q3) in constant time as described in section 2.3.

Testing condition (Q2)*:* To test condition (Q2) we simply test with (HL10) whether e_1 and e_p belong to the same block.

THEOREM 2.53. *The given data structure can answer a biconnectivity query in constant time.* The total update cost is

(1) time $O(k)$ for restoring the relaxed partition,

(2) amortized time $O(k)$ to update all cluster graphs,

(3) amortized time $O((m/k)\log n + \sqrt{m})$ to update all shared graphs,

(4) time $O(k + (m/k)\log n)$ to update all data structures for high-level graphs (HL1)–(HL10), and

(5) time $O((m/k)\log n)$ to update all c-structures.

Thus, choosing $k = \sqrt{m \log n}$ gives the following update time.

THEOREM 2.54. *The given data structure can be updated in amortized time* $O(\sqrt{m \log n})$ *after an edge insertion or deletion.*

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, where insertions are required to maintain the planarity of the embedding. We modify the extended topology tree data structure of [14] 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 [5]. 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 are 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 $D = O(\log n)$ [5]. In the following, *node* denotes a vertex of the topology tree.

In [14] 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 $G(u,v)$. Afterward 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 versa, let $e(C,C')$ be the set of all edges between C and C'. Otherwise, assume w.l.o.g. 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 [14]. 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 $p(C)$, and 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.

(1) 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 $\pi(y,z)$.

(2) 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*.

(3) 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 an 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
   (a) $\pi(u_1, u_p)$ intersects the projection of an LCA-bundle on $p(C)$,
   (b) u_1 is the extreme vertex of the projection of an LCA-bundle or an internal edge,
   (c) u_p is the extreme vertex of the projection of an LCA-bundle or an internal edge, and
   (d) 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 v_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 [14] 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 doubly 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 subbundles in a list have direct access to the projection edge and

only the first and last projection edges 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 [14] 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 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 two projection lists, there are at most four projection lists. The recipe stores these projection lists (i.e., whether a bundle is contained in one or two 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 two 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 two coverage graphs by an edge.

In both cases we then remove all subbundles between Y and Z. If one of the supernodes that was incident to a removed subbundle is no longer incident to a bundle, we replace it by a superedge. Afterward 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 two) 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.

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 [14] 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 lemmas show that two vertices u and v are not biconnected in G iff 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*

(1) *contracting connected subgraphs into one vertex,*

(2) *replacing the only two edges* (a, b) *and* (b, c) *incident to a vertex* b *by the edge* (a, c),

(3) *replacing parallel edges, and*

(4) *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 *iff* 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 that 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*

(1) *contracting connected subgraphs into one vertex,*

(2) *replacing the only two edges* (a, b) *and* (b, c) *incident to a degree-2 vertex* b *by the edge* (a, c),

(3) *collapsing parallel edges, and*

(4) *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 operations 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 as red all the nodes that are small supernodes of $G(u, v)$.

(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 roots are 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 roots are uncolored and connected by a tree edge to x to a yellow node. Otherwise contract all subtrees whose roots are 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 steps (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, and 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 $1 < i < q$. W.l.o.g. 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 $1 \le i \le q$. 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 iff 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 insert(u,v) or query(u,v) operation consists of three 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 [14]. 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. Steps one and two are identical to [14] and take time $O(\log n)$.

In step three of [14] 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 [14] 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 is 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 have to be updated only if one of these subbundles is 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 has to be updated only 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 changes only if this subbundle is modified. □

This results in the following theorem.

THEOREM 3.9. *The given data structure can answer biconnectivity queries in time $O(\log n)$ and can be updated in time $O(\log^2 n)$ after an edge insertion or deletion under the requirement that the edge insertions maintain the planarity of the embedding.*

REFERENCES

[1] D. EPPSTEIN, G. F. ITALIANO, R. TAMASSIA, R. E. TARJAN, J. WESTBROOK, AND M. YUNG, *Maintenance of a minimum spanning forest in a dynamic planar graph*, J. Algorithms, 13 (1992), pp. 33–54.

[2] D. EPPSTEIN, Z. GALIL, AND G. F. ITALIANO, *Improved Sparsification*, Tech. Report 93-20, Department of Information and Computer Science, University of California, Irvine, CA, 1993.

[3] D. EPPSTEIN, Z. GALIL, G. F. ITALIANO, AND A. NISSENZWEIG, *Sparsification—a technique for speeding up dynamic graph algorithms*, J. ACM, 44 (1997), pp. 669–696.

[4] D. EPPSTEIN, Z. GALIL, G. F. ITALIANO, AND T. H. SPENCER, *Separator-based sparsification* II: *Edge and vertex connectivity*, SIAM J. Comput., 28 (1999), pp. 341–381.

[5] G. N. FREDERICKSON, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[6] G. N. FREDERICKSON, *Ambivalent data structures for dynamic* 2*-edge-connectivity and k smallest spanning trees*, SIAM J. Comput., 26 (1997), pp. 484–538.

[7] M. R. Henzinger and M. L. Fredman, *Lower bounds for fully dynamic connectivity problems in graphs*, Algorithmica, 22 (1998), pp. 351–362.
[8] Z. Galil and G. F. Italiano, *Fully dynamic algorithms for* 2*-edge connectivity*, SIAM J. Comput., 21 (1992), pp. 1047–1069.
[9] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
[10] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.
[11] M. R. Henzinger, *Fully dynamic biconnectivity in graphs*, Algorithmica, 13 (1995), pp. 503–538.
[12] M. R. Henzinger and V. King, *Maintaining minimum spanning trees in dynamic graphs*, in Proceedings of the 24th International Colloquium on Automata, Languages, and Programming, Bologna, Italy, 1997, pp. 594–604.
[13] M. R. Henzinger and H. La Poutré, *Certificates and fast algorithms for biconnectivity in fully-dynamic graphs*, in Proceedings of the Third Annual European Symposium on Algorithms, Patras, Greece, 1995, pp. 171–184.
[14] J. Hershberger, M. Rauch, and S. Suri, *Fully dynamic* 2*-edge-connectivity in planar graphs*, Theoret. Comput. Sci., 130 (1994), pp. 139–161.
[15] J. Holm, K. de Lichtenberg, and M. Thorup, *Poly-logarithmic fully-dynamic algorithms for connectivity, minimum spanning tree,* 2*-edge, and biconnectivity*, in Proceedings of the 31st Annual Symposium on the Theory of Computing, Dallas, TX, 1998, pp. 79–89.
[16] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, *Complexity models for incremental computation*, Theoret. Comput. Sci., 130 (1994), pp. 203–236.
[17] M. H. Rauch, *Improved data structures for fully dynamic biconnectivity*, in Proceedings of the 26th Annual Symposium on the Theory of Computing, Montreal, Canada, 1994, pp. 686–695.
[18] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–381.