SRC Technical Note
1997 - 020

September 8, 1997

Improved Data Structures for Fully Dynamic Biconnectivity

Monika Rauch Henzinger

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

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

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 previously best deterministic algorithms were $O(\min\{m^{2/3}, n\})$ in general graphs and $O(\sqrt{n})$ in plane graphs for fully dynamic biconnectivity. We improve these 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 $O(n)$.

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 modifies 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 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 the set of all 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.

$block\text{-}query(u)$**:** Return all nodes in the block of u.

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 and n be the number of vertices in the graph. Previous 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 $O(\sqrt{n}\log n \log(m/n))$ [14]. Block queries were not mentioned in previous work, but can be added to all these data structure such that they take time linear in the size of their output.

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.

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

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 [9] 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, 2] improves the running time of an update operation to $O(\sqrt{n})$. Very recently, a deterministic dynamic connectivity algorithm was given with $O(n^{1/3} \log n)$ update time and $O(1)$ query time [13] and a randomized dynamic connectivity algorithm with $O(\log^2 n)$ update and $O(\log n)$ query time [12, 15]. Both algorithms can also output all nodes connected to a given node in time linear in their number.

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, 19].

The best known dynamic algorithms in plane graphs take time $O(\log n)$ per operation for maintaining connected components by Eppstein et al. [1], $O(\log^2 n)$ for maintaining 2-edge connected components by Hershberger et al. [16, 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* graph. 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 2.33).

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 [20].

2 General graphs

2.1 The graph G' and the relaxed partition of order k

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 $O(\sqrt{m})$.

To partition G into about equally sized subgraphs (see the relaxed partition below), map G to a degree-3 graph G' 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 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 node u of G is called the *origin* of the node u'_i in G', for $1 \le i \le d-2$. We denote nodes of G' by variables with prime, like u' or u'_i and their origin by variables without prime, like u. The graph G' contains at most $2m$ nodes and at most $3m$ edges.

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 representative, 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 [13] and in a dynamic tree data structure [21].

The graph G' is maintained during insertions and deletions of edges as follows: If an edge (u, v) is inserted and the degree $deg(u)$ of u was 3 before the insertion then u is replaced by 2 vertices u'_1 and u'_2 connected by a dashed edge. If $deg(u)$ was larger than 3, then one new vertex $u'_{deg(u)-1}$ is created and connected by a dashed edge of $u'_{deg(u)-2}$. 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 number of vertices in G' might decrease by up to 2, and the number of edges by up to 3.

Let $mlog^n \ge k \ge \sqrt{m}$ be a parameter to be determined later. Note that $m/k \le k$. We build a "balanced" decomposition of G' into subgraphs of size $O(k)$ and maintain data structures based on this decomposition. Every m/k update operations the decomposition and data structures are rebuilt from scratch in linear time, adding an amortized cost of $O(k)$ to each update. The rebuilds significantly simplify the "rebalancing" operations needed to maintain the decomposition balanced during updates. The operations between two rebuilds form a *phase*.

We next describe the balanced decomposition. A *cluster* is a set of vertices that induces a connected subgraph of T'. An edge is *incident* to a cluster if exactly one of its endpoints is in the cluster. 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.

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 $k + 2m/k \le 3k$ nodes of G';

(C2) each cluster has tree degree at most 3;

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

(C4) if a dashed edge is incident to a cluster, then all boundary nodes of the cluster have the same origin; and

(C5) there are $O(m/k)$ many clusters.

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. If the representatives of a vertex u are contained in more than one cluster, u is called a *shared vertex* and each cluster containing a representative of u is called a cluster *sharing* u or u*-cluster*. We use S_u to denote the set of clusters sharing u. Condition (C4) implies that each cluster shares at most one vertex. The shared vertex of a cluster C is denoted by s_C. Together with Condition (C5) it follows that there are $O(m/k)$ shared vertices.

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 non-shared 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 make repeatedly use of the following facts.

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 path in G. Then $\sum_i deg(u_i) \leq 2n$.*

To guarantee that condition (C1)–(C4) are maintained within a phase a 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 and let it gradually "deteriorate" during updates.

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 dashed edge is incident to a cluster, then all boundary nodes of the cluster have the same origin; and

(C5') there are $O(m/k)$ clusters.

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 $O(m + n)$. 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 e. Splitting the cluster at e creates two clusters that fulfill Condition (C1')–(C4'). Each split takes time linear in the size of the cluster. The lemma follows. ▮

We discuss next how to maintain a relaxed partition during updates. We show that an update does not violate Condition (C1), (C4), or (C5). 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 a new representative $u_{deg(u)-1}$ is created, add it to the cluster of $u_{deg(u)-2}$. A deletion does not remove any vertices.

Lemma 2.3 *The update algorithm for the relaxed partition does not violate Condition (C1), (C4), or (C5). Condition (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 2, 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.

Condition (C2) and (C3): Deletions: Note first that the deletion of a non-tree edge does not invalidate Conditions (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 Conditions (C2) and (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 2, violating Condition (C2).

In conclusion, an update violates Condition (C2) and/or (C3) for at most 2 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 four and consists of exactly two tree degree-3 nodes. Splitting it into 2 clusters creates 2 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 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' 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 different vertex. If (a) and (b) hold, then one of the new tree degree-2 clusters might violate Condition (C4). Splitting these cluster in the same way as in the proof of Lemma 2.2 creates at most two additional tree-degree 2 clusters, each fulfilling Conditions (C2) – (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 2.4

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

However, 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 2.5 *Let u and v be two vertices of G.*

- *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 if and only if u'_v and v'_u are biconnected in G_1.*
- *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 if and only if 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, respectively 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 [13] to maintain the shared graphs. Two nodes of G are connected in $G \setminus s$ iff any two representatives of the nodes are connected in $G' \setminus \{s_1, \ldots, s_{deg(s)-2}\}$. Thus, graph G' without any node contractions can be used for testing connectivity.

For each 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]. 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. 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 2.6 *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 $1 \le i < p$, 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 and u and v are separated by an articulation point z. Since z belongs to $\pi_T(u, v)$ either (Q1), (Q2), or (Q3) are violated. Contradiction. ▮

For a query(u, v), let C^u be the cluster of u'_v. If C^u shares a vertex, call it s^u. Basically 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 2.6 can be tested using two cluster graphs and the shared graph of s_u: If s_u does not lie on $\pi(u, y^{(1)})$, s_u does not separate u and $y^{(1)}$, and $y^{(1)}$ belongs to $V(C^u)$. Thus, we use the cluster graph of C^u to test whether u and $y^{(1)}$ are biconnected.

[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 $V(C)$.

If s_u lies on $\pi(u, y^{(1)})$, then let x_u and y_u be the nodes incident to s_u on $\pi(u, y^{(1)})$. Let s' be $s^u_{y^{(1)}}$. Note that s_u belongs to $V(C^u)$ and that $y^{(1)}$ belongs to $V(C_{s'})$. Test in the cluster graph of C^u if u and s_u are biconnected, test in the cluster graph of $C_{s'}$ if s_u and $y^{(1)}$ are biconnected, and test in the shared graph of s_u 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_u 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 analogous 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 if and only if there is an edge between a vertex of C and a vertex of C'. We call C' the *neighbor* of C. The node C and C' are connected by a dashed edge if and only if there is a dashed edge between a vertex of C and a vertex of C'.

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 as *vertices*.

Note that 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 the nodes 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 belongs to 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 maintaining the cluster graphs and the shared graphs. Roughly speaking, a cluster graph tests (under certain conditions) whether two nodes represented by the same node of H_1 are biconnected in G and a shared graph tests whether (some of the) two nodes 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 all 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 a node C and its non-tree neighbor 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.

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

Next we define an *ancestor* for each node in H_1 or H_2. For simplicity we start with H_2. Let A be a node of H_2 at the beginning of the current phase. Then A is the *ancestor* of each node C of H_2 such that C represents a node of G that is also represented by A. Since clusters are only split, never joined, all nodes represented by C are represented by A, i.e., each node of H_2 has a unique ancestor.

To define ancestors for nodes in H_1, we pick at the beginning of a phase for each shared vertex s an arbitrary but fixed s-cluster and call it C_s. Let A be a node of H_1 created at the beginning of the current phase. We call A the *ancestor* of each node C of H_1 such that (1) C 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 $A = C_s$.

Since clusters are only split, never merged, and non-extended vertices can become expanded, but not vice versa, this uniquely assigns an ancestor to each cluster. It is possible that a cluster is its own ancestor.

Let $e = (C, D)$ be an edge of H_i. The *ancestor edge* of e at C is e itself if either e (a) existed during a rebuild, (b) was inserted into H_i after the creation of C and D or, (c) C was created after D by a split of C'

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

and e existed at the creation of C. If D was created by a split of D' after the creation C and e existed at the split, the ancestor edge of e is the ancestor edge of (C, D').

We also define the ancestor edge of e at the ancestor $a(C)$ of C, even though $a(C)$ might not exist in the current graph. The *ancestor edge* of e at $a(C)$ is e itself if either e (a) existed during a rebuild, (b) was inserted into H_i after the creation of D. If D was created by a split of D' after the rebuild and e existed at the split, the ancestor edge of e is the ancestor edge of (C, D').

Note that if in H_i, the ancestor edge of (C, D) and of (C, D') are identical, then D and D' have the same ancestor in H_i.

Data structure:

(HL7) We store at each node of H_i its ancestor and at each edge of H_i its ancestor edge at its both endpoints and at the ancestors of the endpoints .

(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 2.7 *The total number of vertices of G' in all clusters with the same ancestor is $k + 2m/k$.*

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 2 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 requirements for $I(C)$:

(IC2) The graph $I(C)$ has size $O(|V(C)|)$.

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. Thus, $V(C)$ and, hence, $I(C)$, has size $O(k)$.

We next motivate our definition of $I(C)$ and explain why it can only be used 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 C' 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 efficiently be maintained when a neighbor of C is split?

We describe next our solution to these questions. (1) If s_C and C' are represented by the same node then two nodes of $V(C)$ that are biconnected in G might not be biconnected in $I(C)$. See Figure 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 for an example.

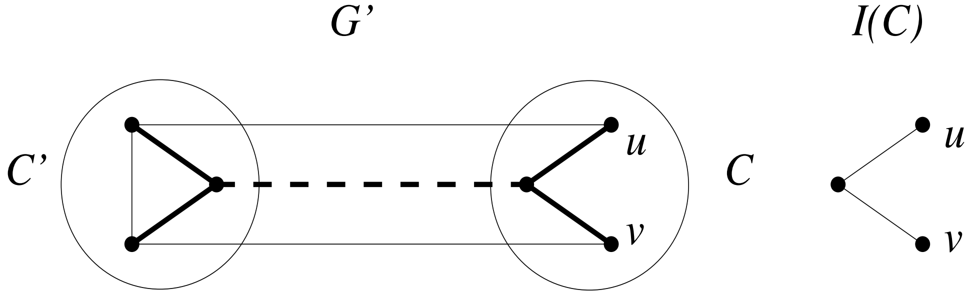

Figure 1: The graph G' and a potential graph $I(C)$. The graph G' consists of cluster C and C' (represented by circles), both sharing vertex s (represented by two nodes and the dashed line between them). Tree edges are bold or dashed. In $I(C)$, C' and s are collapsed to one node.

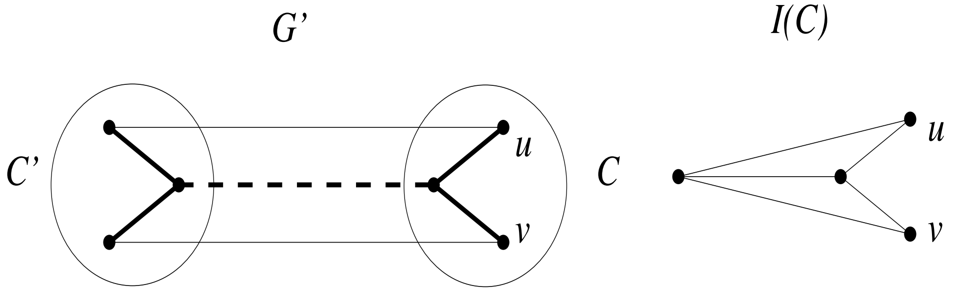

Figure 2: The graph G' and a potential graph $I(C)$. The graph G' consists of cluster C and C' (represented by circles), both sharing vertex s (represented by two nodes and the dashed line between them). Tree edges are bold or dashed. In $I(C)$, C' and s are represented by two different nodes.

However, by Lemma 2.5 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 only happen if s_C separates u and v in G. Since this case is excluded by *(IC1)*, we represent s_C and C' be separate nodes in $I(C)$.

(2) Let d be the number of neighbors of C. There are at most d b- or c-nodes in $I(C)$. Since G' is a graph of degree at most 3, $d = O(|V(C)|)$. To guarantee that $I(C)$ has size $O(|V(C)|)$, $I(C)$ will contain at most $d - 1$ many edges between b- or c-nodes. These edges will be colored and will fulfill the condition that 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 node in $I(C)$, i.e., a node in the cluster graph might represent not just one cluster, but a set of clusters. This leads to the following invariant: *Two clusters are represented by the same node in $I(C)$ or are connected by a colored path if and only if 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

- a node, called *a-node*, for each vertex with a representative in C,
- one node, called *b-node*, for each neighbor C' of C with the same ancestor A,
- one node, called *c-node*, for each maximal set X of clusters such that (a) every cluster C' in the set is a neighbor of C, and all edges (C, C') have the same ancestor edge at C, (b) all clusters in the set have the same ancestor $\neq A$, and (c) all clusters in the set have been connected in $H_1 \setminus C$ at all time and every previous set of clusters containing the vertices of the clusters in X was connected in $H_1 \setminus C$ at all times.

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 neighbors of C that share s_C.

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

- All edges between two vertices of G represented by an a-node belong to $I(C)$.
- 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 $C_{v'}$.
- 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_i is the b- or c-node representing C_i.
- 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, 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 2.8 *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. Thus, 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. Note that any two neighbors of C are connected by colored edges of $I(C)$ iff they are connected in $H_1 \setminus C$.

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 intra-cluster edges of P except for non-dashed intra-cluster edges of C, and (3) by labeling the resulting nodes of $\tilde{P}$ by clusters of G'. Any two neighbors C_1 and C_2 of C that are connected by a subpath of $\tilde{P}$ containing no neighbors of C and no nodes of C are connected in $H_1 \setminus C$ and, thus, are connected by a path of colored edges in $I(C)$. Thus, $\tilde{P}$ induces a path without y in $I(C)$ connecting u and v. Contradiction. ∎

[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.

Lemma 2.9 *For each cluster C,*

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

Proof: Obviously, there are $O(|V(C)|)$ a-nodes and 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 $O(|V(C)|)$ b- or c-nodes. Since the number of colored edges is linear in the number of b- and c-nodes, it follows that $|I(C)| = O(|V(C)|)$. ∎

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

Lemma 2.10 *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: Note first that

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

Let A be the ancestor of the clusters $C_1, \ldots, C_l$. Recall that each C_i either (1) 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 number of updates since the last rebuild, which is m/k.

Thus,

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

∎

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

- building the data structure takes time $O(|V(C)|)$, provided that the b-nodes, the c-nodes, and the red and blue edges are given,
- changing one or all of the colored edges takes time linear in their total number, provided the new colored edges are given, [7]
- 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

[6] Lemma 4.6 of [11] states the result, Section 4.1.2. 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 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 (a special case of) shared graphs, we treat red edges as blue edges in the data structure of [11].

(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 an edge pointing at each other;

(CG3) for each cluster C and neighbor C' of C, a pointer to the b- or c-node in $I(C)$ representing 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 2.11 *Let C be a cluster. There exists a cluster data structure for $I(C)$ such that*

1. *building the data structure takes time $O(|V(C)|)$, 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 (a special case of) shared graphs. There the same problem has to be solved in 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). Since at most $O(m/k)$ new clusters are created in a phase, an amortized constant number of (CG4) arrays is created, contributing an amortized cost of $O(k)$.

While we describe how to determine the changes in the b-nodes, we defer determining the changes in the c-nodes to Section 2.8. In Section 2.9 we show that after each deletion only an *amortized constant* number of c-nodes is split.

Note first that after a split of a cluster the c-nodes in the cluster graph of each resulting cluster represent only 1 cluster since they now all have a different ancestor edge. Thus, it is straightforward to build the resulting cluster graphs in time $O(k)$.

Insertion

Let u' and v' be the (potentially newly added) representatives that are incident to the newly inserted edge (u, v). By Lemma 2.4 only $C_{u'}$ and $C_{v'}$ might be split. Let $C' \in \{C_{u'}, C_{v'}\}$.

Determining the new b-nodes: Let C be a neighbor of C' with the same ancestor as C'. The split of C' might cause the b-node representing C' in $I(C)$ to be split as well. No other b-nodes can change.

We describe next how the new b-nodes are determined. (a) Any cluster created by a split "inherits", i.e. copies the set of old b-nodes from the split cluster. (b) For each split cluster C' execute the follow steps:

Bucket sort the edges incident to the split cluster in lexicographic order of its two endpoints in the updated graph H_1 (using the empty arrays stored at each cluster). (c) Each neighbor of C' with the same ancestor as C' that is incident to edges from $d > 1$ different buckets (i.e. new clusters) of its array receives d new b-nodes representing these new clusters, discards the b-node of C', and keeps all the other old b-nodes. Since $O(k)$ edges are incident to a split cluster, it takes time $O(k)$ to determine the new b-nodes.

Determining the new c-nodes: In the case of an insertion, no existing c-nodes in the cluster graphs of non-split clusters change since clusters created by a split can be represented by the c-node of the split cluster for the following reason: Let S be the set of clusters represented by the c-node of C' in $I(C)$ for some cluster C. All clusters replacing C' have the same ancestor as C', the edges from them to C have the same ancestor edge as (C, C') at C, and all are connected in $H_1 \setminus C$ with each other and with the clusters in $S \setminus \{C'\}$. Thus, the clusters in $S \setminus \{C'\} \cup \{C'', C''$ created by the split of $C'\}$ fulfill Conditions (a)-(c) of a c-node, i.e. can be represented by the same c-node. However, if $(C_{u'}, C_{v'})$ did not exist before the current operation, then a new c-node representing only $C_{v'}$ has to be added to $I(C_{u'})$ and a new c-node representing only $C_{u'}$ has to be added to $I(C_{v'})$. For a split cluster, as discussed before, each neighbor with different ancestor becomes its own c-node.

Updating data structures (CG1)–(CG3): It suffices to discuss how much each cluster graph changes. Using Lemma 2.11 and the simplicity of (CG2)–(CG3) it will follow that all updates take time $O(k)$. (1) The cluster graphs of $C_{u'}$ and $C_{v'}$ are rebuilt from scratch since a new edge is added to them and potentially the clusters are split. (2) The cluster graphs whose b-nodes changes are rebuilt from scratch. As described above, these are the cluster graphs of neighbors of a split cluster with the same ancestor as a split cluster. (3) A red edge is added to the cluster graph of each cluster that was an articulation point on $\pi_{T_1}(C_{u'}, C_{v'})$ separating $C_{u'}$ and $C_{v'}$ before the insertion.

Next we describe how to execute these steps in amortized time $O(k)$ each, given the (new) b-nodes and c-nodes. In Steps (1), (2), and (4), we rebuild cluster graphs which takes time linear in their size for (CG1) (see Lemma 2.11), (CG2), and (CG3). By Lemma 2.10, the sum of the sizes of all rebuilt cluster graphs is $O(k)$.

In Step (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). Lemma 2.11 shows that a red edge can be added to the (CG1) and (CG2) data structure of each articulation in time linear in the number of colored edges, which equals the degree of the articulation point in H_1. 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 2.4). Thus, no b-node changes. However, an amortized constant number of c-nodes is split (as we will show in Lemma 2.33). Additionally, if the deletion removes the edge $(C_{u'}, C_{v'})$ from H_1, the c-node of $C_{u'}$ might be removed from $I(C_{u'})$ and the c-node of $C_{u'}$ might be removed from $I(C_{u'})$.

Updating data structures (CG1)–(CG3): Let u' and v' be the representatives that are incident to the deleted edge (u, v). (1) If $(C_{u'}, C_{v'})$ is removed from H_1, remove $C_{v'}$ in the cluster graph of $C_{u'}$ from the list of its c-node. If the resulting list is empty, remove the c-node. Proceed in the same way with $C_{u'}$ in the cluster graph of $C_{v'}$. Then rebuild the cluster graph for $C_{u'}$, and for $C_{v'}$ from scratch. (2) A red edge is removed and the blue edges are updated in the cluster graph of each new articulation point on $\pi_{T_1}(C_{u'}, C_{v'})$ in the updated H_1. (3) The c-structure (see Section 2.8) returns an amortized constant number of c-nodes that are split. Their cluster graphs are rebuilt from scratch.

Next we describe how to execute these steps in amortized time $O(k)$ each. By the same argument as for insertions, Steps (1) and (3) take amortized time $O(k)$. Step (2) is implemented very similar to the case of insertions, namely the articulation points on $\pi_{T_1}(C_{u'}, C_{v'})$ are determined by $O(m/k)$ queries in the data structure for the *updated* high-level graph H_1. Finding them takes time $O(m/k)$. By Lemma 2.11, replacing all blue and red edges in the cluster graphs of the new articulation points takes time linear in their total number, which is $O(k)$ by Lemma 2.1.

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 2.4, $C_{u'}$, $C_{v'}$, $C_{x'}$ and $C_{y'}$ are the only clusters that might be split.

Determining the new b-nodes: The new b-nodes are determined in time $O(k)$ in the same way as for insertions.

Determining the new c-nodes: The c-structure returns the clusters in which a c-node was split and the new c-nodes.

Updating data structures (CG1)–(CG3): We describe which cluster graphs have to be updated. (1) The cluster graphs of $C_{u'}$, $C_{v'}$, $C_{x'}$ and $C_{y'}$ are rebuilt from scratch. (2) Each cluster graph in which a c-node or b-node was split has to be rebuilt from scratch. (3) A red edge is removed and the blue edges are updated in the cluster graph of each new articulation point on $\pi_{T_1}(C_{x'}, C_{y'})$ in the updated graph H_1 if (x,y) exists. As discussed for insertions and for non-tree edge deletion, each of these steps take amortized time $O(k)$.

We summarize the section with the following theorem.

Theorem 2.12 *Let C be a cluster. There exists a data structure $I(C)$*

- *that tests in constant time whether two vertices u and v of $V(C)$ that are not separated by s_C are biconnected in G, and*
- *that can be updated in amortized time $O(k)$ after each update in G.*

The data structures $I(C)$ for all clusters C 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.

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 $C_s\}$, and let $\mathcal{N}(C_s) = \{C';\, C'$ is node of H_2 and is a neighbor of C_s in $H_2\}$. Shared graphs are used to test whether a pair of two special vertices of G that either are represented by 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 analogous to cluster graphs and use the data structure for cluster graphs also for shared graphs. However, it is possible that $|\mathcal{V}(C_s)| = \Theta(m)$ 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 2.7, 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. Currently in an n-node graph the fastest such data structure takes deterministic time $O(n^{1/3} \log n)$ per edge update and $O(1)$ per query [13] or randomized time $O(\log^2 n)$ and $O(\log n)$ per query [17].

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

- a node, called *a-node*, for each vertex in $\mathcal{V}(C_s)$,
- one node, called *b-node*, for each cluster in $\mathcal{N}(C_s)$ with ancestor A_s,
- one node, called *c-node*, for each maximal set of nodes of H_2 such that (a) every node C' in the set belongs to $\mathcal{N}(C_s)$, and all edges (C', C_s) of H_2 have the same ancestor edge, (b) all nodes in the set have the same ancestor $\neq A_s$, and (c) all clusters in the set have been connected in $H_2 \setminus C_s$ at all times and any previous set of clusters containing the vertices of the clusters in X was connected in $H_2 \setminus C_s$ at all times.

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

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

- All edges between two vertices of $\mathcal{V}(C_s)$ belong to $G(s)$.
- 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 $G(s)$.
- All b- or c-nodes representing tree neighbors of C that are biconnected in H_2 are connected by a tree of *red* edges.
- 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 2.7 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 2.13 *Let u and v be two tree neighbors of a new shared vertex s. Then (the representative of) u and v are biconnected in $G(s)$ iff u and v 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 u and v are connected by a tree path $(u, s), (s, v)$. Contracting edges not incident to s cannot make s into an articulation point separating u and v. Hence, biconnectivity in G implies biconnectivity in $G(s)$. Thus s is the only node that could be an articulation point separating u and v. ∎

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 2.14 *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 $O(|\mathcal{V}(C_s)| + (m/k)\log n)$, 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 in amortized time $O(k+(m/k)\log n)$ per operation with the algorithm of Section 2.5.1 with H_1 replaced by H_2.

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 $\Omega(|\mathcal{V}(s)|)$, which might be $\Theta(m)$. Still we use an approach similar to the one in the previous section but avoid rebuilds from scratch.

The data structure for new shared vertices is rebuilt from scratch if (a) an edge is added to or removed from a node of $\mathcal{V}(C_s)$, or (b) a b- or c-node is split. We want to handle both situations with a small number of edge insertions or deletions in the shared graph of s. Obviously, Case (a) requires $O(1)$ 3 edge insertions or deletions (ignoring the fact that an s-cluster might be split). For Case (b) we handle b-nodes differently from c-nodes: we add all vertices of G represented by b-nodes to $G(s)$ (i.e., we treat them like nodes in $\mathcal{V}(C_s)$ – see below), and we expand each c-node into a tree of bold edges. Note also that a c-node here represents a node of H_1, *not* of H_2: we need the fact that at most $O(k)$ edges can be incident to a node represented by c-node. We describe here the intuition behind the tree for a c-node, by giving a simplification of the exact definition. Let $Neighbor(X)$ be the set of vertices of G that belong to a cluster of the c-node X and are neighbors of a vertex in $\mathcal{V}(C_s)$. Consider the subtree of T created by all tree paths between two nodes of $Neighbor(X)$. Let the vertex set $\mathcal{V}(X)$ consist of X and each node of degree at least 3 in this subtree. The c-node X is represented in $G(s)$ by $\mathcal{V}(X)$, i.e. all nodes of $\mathcal{V}(X)$ belong to $G(s)$ and the bold tree is the above subtree with all degree-2 vertices not belonging to $Neighbor(X)$ and their incident edges replaced by one edge. Note that every edge insertion or deletion in T, in particular every split of a c-node, leads to $O(1)$ edge insertions or deletions in the bold tree. Below we need to relax our definition of the bold tree, but we still are able to show that every edge insertion or deletion in T leads to $O(1)$ edge insertions or deletions in the bold tree of a small number of shared graphs and to potentially many edge insertions and deletions in small shared graphs, i.e., there are to a few "expensive" and many "cheap" edge insertions or deletions.

Since both Case (a) and (b) can be replaced by edge insertions and deletions, we just need to store the shared graph in an efficient dynamic biconnectivity data structure. But this is exactly the problem we want to solve and recursion cannot be applied since $\mathcal{V}(C_s)$ might contain all nodes in G. However, two tree neighbors u and v of s are biconnected in G iff they are connected in $G \setminus s$. Thus, we store the shared graph in a dynamic connectivity data structure, in which an update takes time $O(n^{1/3} \log n)$.

Still there are two problems that we have ignored so far: (a) a split of an s-cluster, and (b) updating colored edges. Problem (a) might lead to a significant decrease in the number of nodes in $\mathcal{V}(C_s)$, but it would be expensive to implement in a dynamic connectivity data structure. Therefore we do not remove these nodes, i.e., the nodes in a shared graph of an old shared vertex are the nodes that belonged to $\mathcal{V}(C_s)$ *at the beginning of the phase*. Since the sets $\mathcal{V}(C_s)$ for different old shared vertices are disjoint at the beginning of the phase, they will stay disjoint throughout the phase. Note that this would not necessarily hold for *new* shared vertices, i.e., we exploit the fact that in this section we only deal with *old* shared vertices. Said differently, we will store $\mathcal{V}(A_s)$ instead of $\mathcal{V}(C_s)$, where A_s is the ancestor of C_s in H_2.

To deal with Problem (b) we build a second shared graph $\tilde{G}(s)$ of size linear in the degree of (the clusters containing the vertices of) A_s in H_1. We rebuild it from scratch whenever the colored edges change. Using Fact 2.1 as in the analysis of cluster graphs shows that the cost of updating all graphs $\tilde{G}(s)$ after an update in G takes time $O(m/k)$.

Even though neither $G(s)$ nor $\tilde{G}(s)$ can be used by itself to answer biconnectivity queries, together they can answer biconnectivity queries since they fulfill the following invariant:

Two neighbors u and v of s are biconnected in G iff u and v are connected in $G(s)$ or the representative of u and the representative of v are connected in $\tilde{G}(s)$.

The graphs $G(s)$

Let $\mathcal{A}(s) = \{A, A$ is the ancestor of an s-cluster in $H_1\}$ and let $\mathcal{C}(s) = \{C, C$ is a node of H_2 and all vertices of C belonged to $A_s\}$.

We call a set $X = \{C_1, ..., C_f\}$ of nodes of H_1 a *c-set of s* if X is a maximal set of nodes of H_1 such that

(a) every node C' in X contains a vertex that is incident to a vertex in $\mathcal{V}(A_s)$, and all edges $(h(C'), A_s)$ have the same ancestor edge in H_2 at A_s,

(b) all nodes in X have the same ancestor A such that $h(A) \neq A_s$, and

(c) all nodes $h(C_j)$, $1 \le j \le f$, have been connected in $H_2 \setminus C_s$ at all times and for every previous set Y of nodes of H_1 containing the vertices of the nodes in X all nodes in $g(D)$ with $D \in Y$ were connected in $H_2 \setminus C_s$ at all times. [8]

We denote the vertices of G contained in the nodes of a c-set X as $\mathcal{V}(X)$. A shared vertex can belong to more than one such set $\mathcal{V}(X)$ while a non-shared vertex belongs to at most one. Note that c-sets correspond to c-nodes in the cluster graph. "B-sets" are not needed, since $G(s)$ will "contain" all nodes of neighbors of C_s with ancestor A_s.

Let s be an old shared vertex, let C_s be the node representing s in H_2, and let A_s be the ancestor of C_s in H_2. The *shared graph $G(s)$* contains as nodes

[8]We say *c-set* instead of *c-set of s* when s is clear from the context.

- a node, called *a-node*, for each vertex in $\mathcal{V}(A_s) \setminus \{s\}$,
- a node, called *d-node*, for each neighbor of a vertex in $\mathcal{V}(A_s)$, and
- a node, called *e-node*, for some of the vertices in $\mathcal{V}(X)$, where X is a c-set of s.

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

- Every edge incident to a vertex of $\mathcal{V}(A_s) \setminus \{s\}$ belongs to $G(s)$.
- For every c-set X the vertices in $\mathcal{V}(X) \cap G(s)$ are connected by a tree of *bold* edges.

We call the subtree generated by the tree paths in T between all vertices that are d-nodes the *subtree generated by the d-nodes*. We sometimes identify a d- or e-node with its vertex in G.

For efficiency we need to require that the bold edges fulfill the following *bold-tree invariant*:

(I1) *Each e-node is incident to at least three bold edges.*

(I2) *At each point in time a bold edge either*

- *represents an edge that used to be in T but has since been deleted exclusive or*
- *represents (we say* covers*) the tree path in (the current) T between its endpoints*

such that each edge in T is covered by at most 1 bold edge in $G(s)$ at each point in time and each deleted edge is represented by at most 1 bold edge in $G(s)$ during the whole phase.

Invariant (I1) guarantees that there are fewer e-node than d-nodes. The crucial observation for efficiency is: If a bold edge e represents a deleted edge e' *and* its endpoints are disconnected in $G \setminus \{x, x \in C_s\}$ then we can afford to remove e from $G(s)$. As we show below the removal of e from $G(s)$ can be charged to the deletion of e'. However we cannot afford to discard of the remaining bold edges that represent a deleted edge. *Data structure:*

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

- *insert(u,v)/delete(u,v):* Insert or delete the edge (u, v) in time $O(m'^{1/3} \log n)$, where m' is the number of edges in $G(s)$.
- *connected?(u,v):* Test whether u and v are connected in constant time.
- *component?(u):* Return the connected component of u in constant time.

(S2) We also keep a list of c-sets of $G(s)$ and store at each c-set the corresponding ancestor edge. For each c-set, i.e., each bold tree, we keep a degree-$m'^{1/3}$ ET-tree data structure whose leaves form an ET-traversal of the bold tree, where m' is the number of edges in $G(s)$. We store at the root of the ET-tree the name of the c-set. Splitting or joining an degree-$m'^{1/3}$ tree takes time $O(m'^{1/3})$, traversing a path from a leaf to the root takes constant time[18].

(S3) For each cluster A that existed at the beginning of the phase we keep a list of pointers to all (S2) data structures representing a c-set with ancestor A. For each c-set we keep a pointer back to its entry in the list and to the cluster A.

(S4) For each bold edge we keep a bit *outdated?* that is set to true iff the edge represents a deleted edge.

(S5) For each connected component of $G(s)$ we keep a d-node or e-node belonging to the component.

The data structure (S5) is needed to determine for each node x of $\mathcal{V}(A_s) \setminus \{s\}$ a c-set connected to x in $G(s)$ (if such a c-set exists), called the *representative* of x. Together with (S1) and (S2) the representative of x can be found in constant time.

The number of nodes and edges in a graph $G(s)$ is linear in the degree of the vertices contained in $\mathcal{V}(A_s)$. Thus, the total space for (S1) – (S5) for all old shared graphs is $O(m)$.

The graphs $\tilde{G}(s)$

The graph $\tilde{G}(s)$ contains as nodes

- a *c-node* for each c-set in $G(s)$.
- All c-nodes that are connected in $G(s)$ are connected by a tree of *yellow* edges in $\tilde{G}(s)$.
- All c-nodes of c-sets whose elements are connected in $H_1 \setminus C(s)$ are connected by a tree of *green* edges in $\tilde{G}(s)$.

Data structure:

(S6) We store $\tilde{G}(s)$ in an adjacency list representation and label each node with its connected component.

(S7) We keep one empty array of size n to guarantee that various bucketsorts can be executed in time linear in the number of sorted elements.

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 2.15 *Two tree neighbors u and v of s are biconnected in G iff u and v are connected in $G(s)$ or the representative of u and the representative of v are connected in $\tilde{G}(s)$.*

Proof: Each edge in $G(s)$ or $\tilde{G}(s)$ corresponds to a path in G that does not contain s, and each vertex $u \in \mathcal{V}(A_s) \setminus \{s\}$ is connected to its representative by a path in G that does not contain s. Since connectivity of u and v in $G \setminus \{s\}$ implies biconnectivity in G, the claim follows.

To show the other direction consider a path P in G that connects u and v and does not contain s. Partition P into subpaths such that each subpath is a maximal path either (a) consisting only of edges incident to nodes of $\mathcal{V}(A_s) \setminus \{s\}$, or (b) containing only edges whose (both) endpoints are represented by the same c-node of $\tilde{G}(s)$, or (c) not fulfilling (a) or (b).

Note that the endpoints of each subpath fulfilling (a) or (b) are also connected in $G(s)$. In remains to be shown that the representatives of the endpoints of each subpath $\tilde{P}$ fulfilling (c) are connected in $\tilde{G}(s)$. Note that $\tilde{P}$ connects two vertices represented by the c-nodes d_1 and d_2 and that it does not contain a vertex of an s-cluster. Thus, the nodes of H_1 represented by d_1 and d_2 are connected in $H_1 \setminus \mathcal{C}(s)$. Thus, they are connected in $\tilde{G}(s)$. ∎

Note that (S1), (S2), (S5), and (S6) allow us to execute these tests in constant time.

Updates in $G(s)$

We need to show that $G(s)$ can be built in time linear in its size and that updating all graphs $G(s)$ takes amortized time $O(k \log n + \sqrt{m} \log n)$ per edge insertion or deletion into G.

Given a spanning tree T, an *ET-traversal of T* is a sequence of vertices of T, in the order in which they are encountered by a depth-first traversal of T. Each vertex v occurs $\deg(v)$ times. Given a set X of vertices of T, an *ET-traversal of X in T* is the restriction of the ET-traversal of T to the vertices in X with consecutive multiple copies at the same vertex removed. An *ET-path* of X in T is a set of edges connecting every consecutive pair of vertices of the ET-traversal of X in T. During updates we make use of the fact that an ET-path of X can be computed in time $O(|X|)$.

We first describe how to (re)build $G(s)$ in time linear in its size times $O(\log n)$. The a-nodes and their incident edges are given by (G1) and (G3). We describe how to find the d-nodes and bold edges for a c-set X. Note that at the beginning of a phase X consists of exactly one node A of H_1. Determine all edges between A and A_s as follows: Bucket sort all edges incident to a cluster in $\mathcal{A}(s)$ according to the node of $H_1 \setminus \mathcal{A}(s)$ to which their non-$\mathcal{A}(s)$ endpoint belongs. The set of endpoints of the edges in the bucket of A are the d-nodes in $\mathcal{V}(X)$.

Each endpoint, i.e. vertex in G, is represented by at least one leaf in the ET-tree of T in (G3). For each vertex in $\mathcal{V}(X)$ pick an arbitrary such leaf and label it with the vertex. Then traverse the ET-tree of T from each chosen leaf to the root and label each internal node of the ET-tree with the concatenation of the labels of its children in left-to-right order. The label of the root gives an ET-path of $\mathcal{V}(X)$ in T.

We use the ET-path to determine the initial bold edges: The first edge of the ET-path becomes the first bold edge e_1. Let $\{e_1, ..., e_{j-1}\}$ be the bold edges already determined. Using the dynamic tree of (G3) we marked every edge on $\pi(x_l, y_l)$ with e_l for every edge $e_l = (x_l, y_l)$ with $1 \le l \le j-1$. We maintain the invariant that the marked edges form a subtree of T. Let $e = (x, y)$ be the next edge on the ET-path such that x belongs to the marked subtree. Note that $\pi(x, y)$ consists of a (possibly empty) sequence of marked edges followed by a (possibly empty) sequence of unmarked edges. Let z be the vertex at which the edges switch from marked to unmarked if such a vertex exists or $z = y$ otherwise. We find z by rooting the dynamic tree at x and determining the first node z incident to a marked edge on $\pi(y, x)$. If $z = y$, we test whether the edge incident to y on $\pi(y, x)$ is marked with an edge ending at y. If not, let (z_1, z_2) be the marking. Remove the bold edge (z_1, z_2) and add the bold edges (z_1, y), (y, z_2). Then continue with the next edge on th ET-path. Otherwise, i.e., $z \ne y$, if z is a d-node or e-node, add the bold edge $e_j = (z, y)$ and mark its path. If z is not a d-node or e-node, it is incident to two tree edges, marked with the same bold edge (z_1, z_2). Remove the bold edge (z_1, z_2), add the bold edges (z_1, z), (z, z_2), (z, y), mark the corresponding paths, and make z an e-node and y a d-node. Then continue with the next edge on the ET-path. Finding the bold edges takes time linear in the number of d-nodes. Thus, $G(s)$ and (S1) and (S2) can be built in time linear in the size of $G(s)$ times $O(\log n)$.

While building $G(s)$ give each cluster A represented by a c-set a pointer to this c-set and store at the c-set a pointer back to its position in the (S3)-list of A and to A.

All *outdated?*-bits are set to false.

To build (S5) ask a *component*?(u)-query for each d- or e-node in $G(s)$ and store the node at the component if no node is stored there yet.

Obviously, building (S3) – (S5) takes time linear in the size of $G(s)$.

Next we describe how to update the $G(s)$ after an update in G. Let m' denote the size of $G(s)$. Note that one non-bold edge is inserted or deleted in at most two graphs $G(s)$, taking time $O(m'^{1/3}\log n) = O(k)$ each. We are left with discussing how the bold edges change. There are three cases to consider: (A) a d-node x is no longer connected to a node in $\mathcal{V}(A_s)$, and it and potentially one e-node has to be removed from $G(s)$; (B) a vertex x that is not yet a d-node becomes connected to a node in $\mathcal{V}(A_s)$ and it and potentially one e-node has to be added to $G(s)$; (C) a tree edge covered by a bold edge in a c-set X is deleted without splitting X; or (D) a c-set has to be split since its corresponding nodes in H_2 are no longer connected in $H_2 \setminus C_s$.

(A) If the d-node x is incident to at least three bold edges, it simply becomes an e-node. If it is incident to two bold edges (z_1, x) and (x, z_2), these edges are removed and (z_1, z_2) is added to $G(s)$. The *outdated?*-bit of (z_1, z_2) is set to the logical-*or* of the *outdated?*-bit of the two deleted edges. If x is incident to one bold edge, this edge is deleted and if this creates an e-node of degree 2, this node is deleted as described for the degree-2 case of x. Note that the bold-tree invariant is not violated. If a c-set becomes empty, it is removed from the corresponding (S3) list. Before removing x we test one node in each other c-set to determine whether it is connected to x. If so, it is stored at the connected component of x. Updating (S1)-(S5) takes time $O(m'^{1/3}\log n + m/k)$.

(B) Let x' be the node of G' incident to the new edge and let $C' = C_{x'}$ be a node in H_1. If no other vertex of C' is incident to a vertex of $\mathcal{V}(A_s)$, the new edge is its own ancestor edge. A new 1-element c-set is formed and a (S2) and (S3) data structure is created. Finally we determine the connected component of x and store x as d-node in the corresponding (S5) data structure.

Otherwise, we determine the ancestor edge e' of $(h(C'), A_s)$ using (HL7) and search for the c-set X with ancestor edge e'. For each bold edge $e' = (w, z)$ of X whose *outdated?*-bit is not set, mark in the dynamic tree of T using (G3) all edges on $\pi(w, z)$. Let y be the endpoint of a bold edge of x. To update the bold edges, use (one iteration of) the algorithm for determining bold edges during a rebuild to process the addition of the edge (x, y) to the marked subtree. The *outdated?*-bit of all new edges is set to *false*. Each operation in the dynamic tree takes time $O(\log n)$. Finding e' and X takes time $O(m/k)$. Updating (S1), (S2), and (S4) takes time $O(k\log n + m'^{1/3}\log n)$, (S3) and (S5) are unchanged.

(C) The bold edge e representing the path containing the deleted edge e' in X now represents the deleted edge e'. Thus, only (S4), i.e. the *outdated?*-bit of e, has to be updated. Let A be the ancestor of the cluster containing e. Using (S3) find all bold edges whose *outdated?*-bit is $false$ of a c-set of A in any shared graph and test them to determine to ones that represent a path containing e'. Set their *outdated?*-bit to $true$. To test an edge first split the ET-tree of T in (G3) at e', resulting in two ET-trees, one for each subtree of $T \setminus e'$. Test whether the endpoints of the edge belong to different ET-trees by traversing the paths from a leaf representing each endpoint to the root.

By Lemma 2.7 there are $O(k)$ bold edges to test, for a total time of $O(k)$.

(D) In the fourth case, the split of a c-set, the c-structure in Section 2.8 returns (an amortized constant number of) old shared graphs $G(s)$ such that one of the c-sets of s is split by the update. Additionally, it gives for each split c-set X the resulting new c-sets X_1 and X_2. The split of the c-set X of s may lead to deletions and insertions of bold edges incident to the nodes of $\mathcal{V}(X)$. We describe first how to find these edges and how to update (S1), (S2), and (S4) accordingly. Afterwards, we describe how to update (S3) and (S5). Let A be the H_1-ancestor of all nodes in X.

(1) *Remove the bold edges connecting X_1 and X_2.* A bold edge has to be deleted if its endpoints belong to different connected components of $H_2 \setminus C_s$. For this test map both endpoints to their nodes in H_2 and test whether they are biconnected using (HL3) and (HL4).

Determine all existing bold edges incident to the vertices of $\mathcal{V}(X)$ by traversing the leaves of the ET-tree of X. Test each bold edge. Let B be the set of bold edges to be deleted. Delete B by removing the edges from (S1) and splitting the ET-trees (S2) suitably. This results in various ET-subtrees. Note that both endpoints of each remaining bold edge belong to the same ET-subtree. Finally, if a resulting bold tree contains an e-node with degree less than 3, then remove the node and update its incident edges as described using (HL3) and (HL4).

(2) *Reconnect the resulting pieces into two bold trees such that the bold-tree invariant holds.* Let X_1 and X_2 be the resulting c-sets. For $i = 1, 2$ our goal is to reconnect the different ET-trees representing X_i. However, the remaining bold edges whose *outdated?*-bit is $true$ complicate the process: two vertices may belong to the same ET-tree but some edges on the tree path between them may not be covered. Thus it is possible that the vertices of the same ET-tree do not belong to the same subtree of covered edges, but are partitioned into various vertex-disjoint subtrees of covered edges.

This leads to the following algorithm. For $i = 1, 2$ find an ET-path of $\mathcal{V}(X_i)$ in T. Wlog the path starts in the first ET-tree. Process the edge on the ET-path in order. Let $e = (x, y)$ be the next edge to process. If x and y belong to the same ET-tree, do nothing. Otherwise, assume x belongs to the current ET-tree and y belongs to the p-th ET-tree. Determine the subtree of covered edges of the current ET-tree closest to y on $\pi(x, y)$ and an e- or d-node z' in this subtree. Determine the vertex z of this subtree closest to y on $\pi(z', y)$. Determine the vertex w closest to z on $\pi(z, y)$ that belongs to a different subtree of covered edges. This subtree must belong to the q-th ET-tree. Add z and w as e-nodes to $G(s)$ and update the bold edge covering their two incident covered edges suitably, if z and w do not already belong to $G(s)$. Add the bold edge (z, w) and join the current and q-th ET-tree. Repeat with x replaced by w until the current and the p-th ET-tree have been joined.

We implement this algorithm as follows: We keep two different costs, i.e. markings, of the dynamic tree of T in (G3). $Cost_1$ is set to 1 for each e-node or d-node belonging to the current ET-tree. $Cost_2$ is set to 1 for an edge iff the edge is covered by any bold tree edge of X. We use $cost_1$to determine z' and $cost_2$ to determine z and w. When joining two ET-trees, $cost_1$ of the d- and e-nodes in the q-th ET-tree is set to 1. The *outdated?*-bit of all new bold edges is set to *false*.

Lemma 2.16 *The update algorithm maintains the bold-tree invariant.*

Proof: The invariant obviously holds after each rebuild. If a d-node is no longer connected to or starts to be connected to a vertex of $\mathcal{V}(A_s) \setminus \{s\}$ we update the bold tree to maintain (I1) without violating (I2), since the path (x, z) is uncovered before the update.

Whenever T changes without splitting a c-set, then a tree edge e must have been deleted. Bold edges not containing e in the tree path between their endpoints continue to represent their tree path. The at most one bold edge whose tree path contains e now represents the deleted edge e. Thus, the invariant continues to hold.

Whenever T changes and a c-set X is split, we update (I2) without violating (I1): we remove some bold edges representing deleted edges and e-nodes incident to fewer than three bold edges, if z and /or w do not yet belong to $G(s)$, we add them and update the bold edges covering their incident marked edges suitably, and we add edges (z, w) covering a previously uncovered path. Thus, the bold-tree invariant continues to hold. ∎

To update (S5) we determine a vertex of X_i and store X_i at the component of the vertex for $i = 1, 2$.

Running Time Analysis of Case (D):

In the data structure (S1) and (S2) it takes time $O(m'^{1/3}\log n)$ to add or delete a bold edge, for a total of $O(|B|m'^{1/3}\log n)$. As we show below, (a) we spend time $O(k\log n + |B|m'^{1/3}\log n)$ per old shared graph to determine which edges to add and delete. We charge $O(k\log n)$ time to the current update operation in G.

We also show (b) that using the bold-tree invariant, the cost of $O(|B|m'^{1/3}\log n)$ can be charged to either the current or $|B|$ previous edge deletions in G such that the total cost ever charged to an edge deletion in G by *all* old shared graphs is $O(\sqrt{m}\log n)$. It follows that the total cost of updating $G(s)$ can be amortized over $|B|+1$ edge deletions in G, adding only an amortized cost of $O(k\log n + \sqrt{m}\log n)$ to each deletion. We are left with proving (a) and (b).

To prove (a) we analyze the time for an update. We repeatedly use the fact that there are $O(k)$bold edges, d-nodes, and e-nodes representing a c-set, and also $O(k)$ non-bold edges incident to d-nodes of a c-set. In Step (1) testing a bold edge takes time $O(\log n)$ for a total of $O(k\log n)$. Removing the edges in B decreases the degree of the e-nodes by at most $2|B|$ and, thus, leads to the removal of $O(|B|)$ e-nodes. Thus, $O(|B|)$ edges are removed and added to (S1) and (S2), for a total time of $O(|B|m'^{1/3}\log n)$. Thus, Step (1) takes time $O(k\log n + |B|m'^{1/3}\log n)$ altogether.

The cost of the operations in Step (2) are: Building an ET-path for $\mathcal{V}(X_i)$ takes time $O(|\mathcal{V}(X_i)| = O(k)$ since the ET-tree in (S2) has constant depth. We spend time $O(\log n)$ for each new bold edge, for a total of $O(|B|\log n)$. Additionally each of the $O(k)$ d-and e-nodes has its $cost_1$ set to 1, for a total of $O(k/logn)$. Finally, there are $O(|B|)$ edge insertions and deletions in $G(s)$. Thus, the total time for Step (2) is $O(k\log n + |B|m'^{1/3}\log n)$.

The time to update (S3) and (S5) is constant. Since the number of bold edges in all c-sets of A is linear in the number of nodes of A, the time for updating (S4) is $O(k)$. Thus, the total time spent in $G(s)$ after an update in G is $O(k\log n + |B|m'^{1/3}\log n)$.

As described before, we charge $O(k\log n)$ to the current update in G. Next we have to account for the remaining $O(|B|m'^{1/3}\log n)$ cost. If $m' < m^{3/4}$ and the number of edges between a vertex in an s-cluster and a vertex in A is at most $k/m^{1/4}$ throughout the phase, then

$$O(|B|m'^{1/3}\log n) = O((k/m^{1/4})m^{1/4}\log n) = O(k\log n).$$

Thus, we charge this cost to the current update in G as well. Otherwise we charge the cost of $O(|B|m'^{1/3}\log n)$ to $|B|$ (specific) previous edge deletions in G. To show (b) we need to prove that the total cost ever charged to an edge deletion in G by *all* old shared graphs is $O(\sqrt{m}\log n)$.

Note that each deleted bold edge e must represent a deleted edge e' of G, since its endpoints are no longer connected in $G \setminus \{x, x$ belong to an s-cluster $\}$. We charge the cost of deleting e to e'. By the bold-tree invariant, at most one bold edge in $G(s)$ ever represent each deleted edge e. Thus at most one deletion in $G(s)$ is ever charged to e'.

How much cost is charged to e' during the whole phase by *all* old shared graphs? Number the old shared graphs that have a deletion charging to e' and let J be the set of these indices. Let m'_i be the size of the i-th old shared graph. For each $i \in J$ either (i) $m'_i \geq m^{3/4}$ or (ii) $m_i \leq m^{3/4}$ and $b_i \geq k/m^{1/4}$ at some point(s) in the phase, where b_i is the number of bold edges of the c-set X in the i-th old shared graph such that a deletion in X is charging to e'.

Let J_1 be the set of indices of old shared graphs in case (i) and let $J_2 = J \setminus J_1$. Since $\sum_i m'_i = O(m)$, $|J_1| = O(m^{1/4})$. The cost of all old shared graphs contributing to (i) is

$$O(\sum_{i\in J_1} m_i'^{1/3}\log n) = O(\sqrt{m}\log n).$$

Since throughout the phase there are $O(k) + m/k = O(k)$ edges incident to the vertices represented by a c-set, $|J_2| = O(m^{1/4})$. Additionally, for $i \in J_2$, $m_i'^{1/3} = O(m^{1/4})$. Thus, the cost of all old shared graphs contributing to (ii) is

$$O(\sum_{i \in J_2} m_i'^{1/3} \log n) = O(m^{1/4} m^{1/4} \log n) = O(\sqrt{m} \log n).$$

We summarize the result in the following lemma.

Lemma 2.17 *The data structures for all shared graphs $G(s)$ can be built in time $O(m \log n)$ and can be updated in amortized time $O(k \log n + \sqrt{m} \log n)$ after each rebuild. They can test in constant time whether two nodes are connected in a given graph $G(s)$.*

Updates in $\tilde{G}(s)$

We show how to build $\tilde{G}(s)$ in time linear in its size and how to update all graphs $\tilde{G}(s)$ in time $O(k)$ after an update in G.

The nodes of $\tilde{G}(s)$ are given by (S2). The yellow edges are determined as follows: Using (S2) determine for each c-set X an arbitrary d-node or e-node v_X representing it in $G(s)$. Bucketsort the nodes v_X according to their connected components using (S7). For each bucket connect all c-sets in the bucket by a chain of yellow edges.

To determine the green edges determine a vertex of c for each c-set and map it to a node of H_2 using (HL1), and map each node of H_2 to a tree neighbor of C_s using (HL3). Finally bucketsort the c-nodes according to the biconnected component of the tree neighbor in H_2 to which they were mapped using *block-id?*-queries in (HL4) for H_2. For each bucket, connect all c-sets in the bucket by a chain of green edges.

The rebuild takes time linear in the number of c-nodes and the number of nodes in H_1 for a total of $O(m)$ for all old shared graphs.

Next we describe the insertion or deletion of an edge in G. Using the above rebuild algorithm $G(s)$ can be updated in time $O(m/k)$ if a yellow edge or a c-set changes and in time linear in the number of c-nodes if a green edge changes. We argue next that a yellow edge or a c-set changes in an amortized constant number of old shared graphs and that a green edge changes in clusters that are articulations on a path in H_2 (either before or after the update) plus a constant additional number of clusters. By Fact 2.1, the total time spent is $O(m/k)$.

Consider an update of edge (u, v) in G. The yellow edges or a c-node of an old shared graph $\tilde{G}(s)$ have to be modified if a change in the corresponding graph $G(s)$ occurred. Recall that this happens only in an amortized constant number of graphs. Thus, yellow edges and c-nodes have to be updated only in an amortized constant number of graphs.

Let D_u be the node of H_2 containing u. The green and yellow edges are completely recomputed for $\tilde{G}(s)$ such that u or v is contained in an s-cluster. Additionally, the green edges of an old shared graph $\tilde{G}(s)$ have to be modified if the connected component of $H_1 \setminus \mathcal{C}(s)$ change which happens iff the connected components of $H_2 \setminus C_s$ change. The latter happens (a) after the deletions of an edge (u, v) iff C_s becomes an articulation point on $\pi_{T_2}(D_u, D_v)$, and (b) after the insertion of an edge (u, v) iff C_s was an articulation point on $\pi_{T_2}(D_u, D_v)$ before the insertion and is no longer an articulation point after the insertion.

It follows that the green edges only have to be updated for shared graphs of old shared nodes s such that C_s is an articulation point on $\pi_{T_2}(D_u, D_v)$ either before or after the update plus at most 2 additional old shared graphs.

This leads to the following lemma.

Lemma 2.18 *The data structures for all graphs $\tilde{G}(s)$ can be built in time $O(m)$ and can be updated in amortized time $O(k)$ after each rebuild. They can test in constant time whether two nodes are connected in a given graph $\tilde{G}(s)$.*

We summarize the section with the following theorem.

Theorem 2.19 *Let s be a shared vertex. There exists a data structure*

- *tests in constant time whether two tree neighbors u and v of s are biconnected in G, and*
- *that can be updated in amortized time $O(k \log n + \sqrt{m} \log n)$ after each update in G.*

The data structures for all shared vertices 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 constant time 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)

Given the node C and its non-tree neighbor C' 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, 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. 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:

- *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.

As we show below, *biconnected?* and *blockid?* takes 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'.

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 latter 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 $O((m/k) \log n)$:

– *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.

– *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).

- *biconnected?(C,C',C")*: Return *connected?*(C', C'') in $H_i(C)$.
- *blockid?(C,C')*: Return *component?*(C') in $H_i(C)$.
- *components?*: Return all connected component 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 2.20 *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 sequence of edges connecting two subtrees gives a path in $H_i(C)$. ∎

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 2.21 *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 nodes 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 2.22 *Let C be a node of H_2 on a tree path P in T_2. Let D respectively D' be the neighbor of $C(P_{T_1})$ respectively $C(P_{T_1})'$, on P_{T_1} and not on $\pi_{T_1}(C(P_{T_1}), C(P_{T_1})')$. 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 subtree containing $h(D)$, respectively $h(D')$, in $H_2 \setminus C$ are identical to the edges incident to $ST(D, C(P_{T_1}))$, respectively $ST(D', C(P_{T_1})')$. ▮

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 $O(m/k)$

- 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
- 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. [9]

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 $C_1 \times D_1$ 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.

We show next 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 that (1) are not ancestors of C, but are children of 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 2.23 *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 $1 \leq i \leq p$ 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 $1 \leq i \leq p$.

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

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

Y_j such that $Y \neq X_i$ and 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 with $1 \leq j \leq q\}$ 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 2.22 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 2.24 *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 one node 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 towards the root. 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 rooted at a node R and 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 $O(m/k)$.

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 edges 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* of a level-i cluster X_1 with $i > 0$ consists of

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

- *Case C:* the node of 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 and the vertex of 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.

- *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 actually 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 in the order in which they occur as leaves of P^c in the search tree, 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 $1 < j < p$.

We show next that P_A and also P_B consist of $O(\log n)$ increasing subpaths of complete paths. Thus, we can use the avoidability information for the complete paths to test the avoidability of P_A and P_B except for the nodes that are heads in the complete paths. Recall that P_A and P_B are increasing paths in T_1. It follows that the intersection of P_A (P_B) and a complete path P^c forms a continuous subpath of P_A (P_B). Let $P_1^c, P_2^c, \ldots, P_l^c$ be the complete paths whose intersection with P_A is non-empty such that the head of P_j^c belongs to P_{j+1}^c. Let X_j be the topology node in TT corresponding to P_j^c. 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 of P_j^c which is the partial path of X_j belongs to P_{j+1}^c. Thus, X_j is a true descendant of X_{j+1}. 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:

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

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

- 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.

- In Case C, the partial path of X_1 consists 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.

- 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 2.30 below shows that

- for $i = 1$, let P^c be the complete path 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
- for $i = 2$, let C_1, ..., C_l form an increasing subpath of Q_{T_1} with $C_1 = C(Q_{T_1})$ and $C_l = C(Q_{T_1})'$, and let P^c denote the complete path to which $x(D')$, $x(D)$ and $x(C_q)$ for $1 \le q \le l$ 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.

Note that P^c is the lowest ancestor of the least common ancestor of C (resp C_1) 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 $k \le m/\log^2 n$.

Lemma 2.25 *Given a path P in H_i the extended ambivalent data structure can test the avoidability of all but one node on P in time $O(m/k)$.*

Let us now define $somecov$, $maxcov$, and $shared$ and prove Lemma 2.30. 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 two labels: (1) $maxcov(A, B, e)$ 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 and (2) $shared(A, B, s)$ 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 of 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.

- In Case A, the partial path of X_1 is identical to the partial path of this child.
- 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 $maxcov(X_3, X_2, e)$.

Removing from p all dashed edges belonging to the shared vertex of e if existent or the shared vertex of the endpoints of p splits p into at most two subpaths. We set the $somecov_2$ bits for these subpaths. If e is dashed and belongs to the shared vertex s, and $shared(X_2, X_3, s)$ is 1, we also set the $somecov_2$ bits of the subpath of p containing the dashed edges of s.

- 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 $TT_j \setminus X_2$.

  Removing from p all dashed edges belonging to the shared vertices of the endpoints of p results in at most one subpath of p. We set the $somecov_2$ bits for this subpath.

- 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 $maxcov(X_3, X_2, e)$. Removing from p all dashed edges belonging to the shared vertex of e or the shared vertices of the endpoints of p results in at most two subpaths of p. We set the $somecov_2$ bits for these subpaths. If e is dashed and belongs to the shared vertex s, and $shared(X_2, X_3, s)$ is 1, we also set the $somecov_2$ bits of the subpath of p containing the dashed edges of s.

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.

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

Lemma 2.26 *Let P^c be a complete path and let D, D', and C be basic clusters such that $(x(C), x(D))$ and $(x(C), x(D'))$ are edges on 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 $j+1$ at which a $somecov_1$ bit is set for an ancestor v of $x(C)$. Let X_1 be the level-$(j+1)$ cluster whose partial or complete path P^x contains $x(C)$. Then X_1 has two children X_2 and X_3 in TT, connected by an edge e in TT_j. Wlog $x(C)$ is a node of the partial path of X_2. Thus, $somecov_1(v)$ is set 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' and let X_2 and X_3 be its two children. Wlog 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. ▮

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

Lemma 2.27 *Let C be a basic cluster and let P^c be a complete path containing C. If C is not incident to 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 2.28 *Let P^c be a complete path and let $C_1, ..., C_l$ be basic clusters such that $x(C_1), ..., x(C_l)$ forms a maximal subpath of dashed edges of P^c belonging to the shared vertex s. Let j be the lowest level such that the $somecov_2$ bits are set for an ancestor for every node $x(C_1), ..., x(C_l)$. Then all nodes $x(C_1), ..., x(C_l)$ 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 incident to an intercluster dashed edge on level j and in the partial path containing $x(C_q)$ there exists an ancestor whose $somecov_2$ bit is set. Note that $x(C_q)$ was the endpoint of the partial path of every level $\leq j$ and that the ancestors of the endpoints of a partial path always have the $somecov_2$-bit set to 0. Thus at level j, the $somecov_2$ bit is not set for any ancestor of $x(C_q)$. Contradiction. ∎

Lemma 2.29 *Let P^c be a complete path and let D, D', and $C_1, C_2, ..., C_l$ be basic clusters such that $x(C_1), x(C_2), ..., x(C_l)$ forms a maximal subpath of dashed edges of P^c and $(x(D), x(C_1))$ and $(x(C_l), x(D'))$ are solid edges on P^c. 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 shared vertex 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 2.28, 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 have tree degree 3, a $somecov_2(x(C_q))$ bit was set when constructing of 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 $x(C_1), ..., x(C_l)$ are set because it 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 $x(C_1), ..., x(C_l)$ 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 $x(C_1)$, ..., $x(C_l)$ are set. ▮

Lemma 2.30 *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.*

For $i = 1$, let P^c be the complete path 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.

For $i = 2$, let C_1, ..., C_l form an increasing subpath of Q_{T_1} with $C_1 = C(Q_{T_1})$ and $C_l = C(Q_{T_1})'$, and let P^c denote the complete path containing $x(D')$, $x(D)$ and $x(C_q)$ for $1 \leq q \leq l$. 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 2.26 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.

We discuss next the case $i = 2$. Lemma 2.27 shows the claim if $l = 1$. If $l > 1$, then C represents at least two clusters in H_1. Lemma 2.29 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. ▮

Updates

We show next 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. 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. Since the clusters at all levels that contain u, v, x, or y form 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 2.31 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 2 clusters at each level have to be updated. 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. (We discuss below how to restore the partial and complete paths of the clusters that are children of clusters containing u, v, x, or y). 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. Note that this takes time proportional to building the back log, which is $O(m/k)$. Note also that modifications in the back log of one child does not affect the back log of its sibling.

Lemma 2.31 *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: Note first that a $somecov$ bit is set 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 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 cluster with tree degree 2 shares at most two vertices, at most 4 $shared(A_1, B_1, .)$ 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.

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

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.

Note that A_3 has tree degree at least 2, since A_3 is adjacent to A_2 and A_3 is incident to the tree edge incident to A_1. 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 the case 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 case 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, A_3 shares s, and B_1 has two children B_2 and B_3.

Note that A_3 has tree degree at least 2, since A_3 is adjacent to A_2 and A_3 is incident to the tree edge incident to A_1. 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 the case 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)

$$shared(A_1, B_1, s) = shared(A_3, B_2, s) or\, shared(A_3, B_3, s) or\, shared(B_2, A_2, s) or\, edge(A_2, B_3).$$

(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)

$$shared(A_1, B_1, s) = shared(A_3, B_2, s) or\, shared(A_3, B_3, s) or\, shared(B_2, A_2, s) or\, shared(B_3, A_2, s).$$

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

$$shared(A_1, B_1, s) = shared(A_3, B_2, s) or\, shared(A_3, B_3, s) or\, edge(A_2, B_2) or\, edge(A_2, B_3).$$

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 finishes the proof of the following lemma.

Lemma 2.32 *The extended ambivalent data structure can determine the avoidability of all but one node on a path P in H in time $O(m/k)$. It 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), the c-nodes of a shared graph $G(s)$ for a new shared vertex s (see Section 2.6.1), or the c-sets for an old shared vertex s (see Section 2.6.2) determine which c-nodes or c-sets are split by an update operation. For this problem we give a data structure, called *c-structure* and show that each update splits only an amortized constant number of c-nodes or c-sets. We first recall the definitions of c-nodes and c-sets and then exactly define the operations of the c-structure.

Given a node C in H with ancestor A a c-node of the cluster graph $I(C)$ represents a maximal set X of nodes of H_1 such that (a) every node C' in X is a neighbor of C, and all edges (C, C') have the same ancestor edge in H_1 at C, (b) all nodes in X have the same ancestor $\neq A$, and (c) all nodes have been connected in $H_1 \setminus C$ at all times and every previous set of clusters containing the vertices of the clusters in X was connected in $H_1 \setminus C$ at all times.

The definition for a c-node of the shared graph $G(s)$ of a new shared vertex s is identical with H_1 replaced by H_2.

A c-set of an old shared vertex s is a set X of nodes of H_1 such that (a) every node C' in X contains a vertex that is incident to a vertex in $\mathcal{V}(A_s)$ and all edges $(h(C'), A_s)$ have the same ancestor edge in H_2 at A_s, (b) all nodes in X have the same ancestor A such that $h(A) \neq A_s$, and (c) all nodes $h(C')$ with $C' \in X$ have been connected in $H_2 \setminus C_s$ at all times and for every previous set Y of nodes of H_1 containing the vertices of the nodes in X all nodes in $g(C")$ with $C" \in Y$ were connected in $H_2 \setminus C_s$ at all times.

We generalize all three definitions. Let H and H' be high-level graphs such that either $H = H'$ or $H = H_1$ and $H' = H_2$. Let g be the identity function if $H = H'$ and let $g = h$ otherwise.

Let C be a node of H' and let A be the ancestor of C in H'. Let a be the identity function if $H = H'$ and let a be the mapping from a node of H' to its ancestor otherwise.

A c-set is maximal set X of nodes of H such that

(a) every node C' in X contains a vertex that is incident to a vertex in $a(C)$ and all edges $(g(C'), a(C))$ have the same ancestor edge in H' at $a(C)$;

(b) all nodes in X have the same ancestor A' such that $g(A') \neq A$; and

(c) all nodes in $g(C')$ with $C' \in X$ have been connected in $H' \setminus C$ at all times and for every previous set Y of nodes of H containing the vertices of the nodes in X all nodes in $g(C")$ with $C" \in Y$ were connected in $H' \setminus C$ at all times.

We call $(g(C'), a(C))$ the *ancestor edge of the c-set*.

Note that any two different c-sets are disjoint. Given a high-level graph H, a set $\mathcal{C}$ of nodes $a(C)$ for the nodes C of H' and the data structures for H and H', the c-structure executes the following operations.

- $split(C, C_1, C_2, u, v)$ 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. Return a (possibly empty) list of split c-sets and for each split c-set return the element list of the two resulting c-sets.
- $add(C_1, C_2)$ where C_1 and C_2 are nodes of H. Add one edge between C_1 and C_2 and return a (possibly empty) list of modified c-sets and for each modified c-set return its new element list.
- $remove(C_1, C_2)$ 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 modified c-sets and for each modified c-set return its new element list.

We show below that each operation can be executed in time $O((m/k) \log n)$. Note that a split operation is called whenever a node in H' is partitioned by a $delete(u, v)$ or $insert(u, v)$ operation. As shown in Section 2.2 a constant number of nodes of H' are split after each update in G. Additionally either an add or a $remove$ is executed in the c-structure. Thus maintaining and using the c-structure increases the cost per update in G only by a constant factor.

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

(T1) For each node of $\mathcal{C}$ we keep a list of its c-sets. For each c-set we keep a list of its nodes of H, and for each node in H we keep a pointer back to its position in the list for all its c-sets.

At the beginning of a phase each c-set consists of one node: every neighbor of a node in $\mathcal{C}$ forms its own c-set.

We implement the operations as follows: $split(C, C_1, C_2, u, v)$: Determine all articulation points on $\pi(C_u, C_v)$ in (the updated) H'. For each articulation point D search all c-sets incident to $a(D)$ to determine whether C is contained in one. If not, go to the next articulation point. If C is contained in a c-set X and both C_1 and C_2 are incident to D, replace C by C_1 and C_2 in X. Otherwise, only one of C_1 or C_2 is incident to D and C replaced by it in X. Note that all nodes in X still fulfill (a) and (b). Using (HL3) determine for each node C' in X the tree neighbor of D on $\pi(g(C'), D)$ and bucketsort the node according to the blockid of "its" tree neighbor using (HL4). This results in either 1 or 2 non-empty buckets. In the latter case, split the list of X accordingly and return X and the two resulting lists. For all other c-set containing C, i.e., the ones at a node $a(D)$ such that D is not an articulation point on $\pi(C_u, C_v)$ replace C by either C_1 or C_2 or both, depending on which of the new nodes are incident to D.

If $a(g(C))$ is replaced by $a(g(C_1))$ and $a(g(C_2))$ in $\mathcal{C}$, copy the c-sets of $a(g(C))$; remove all the nodes of H that are not incident to a node of $a(g(C_i))$, and then test each c-set whether it has to be split as above.

$add(C_1, C_2)$: If the ancestor of $g(C_1)$ equals the ancestor of $g(C_2)$ stop. Otherwise, search the c-sets of $a(g(C_1))$ to determine whether C_2 belongs to one of them. If not, create a new c-set consisting only of C_2. Repeat with the roles of C_1 and C_2 exchanged.

$remove(C_1, C_2)$: If the ancestor of $g(C_1)$ equals the ancestor of $g(C_2)$, stop. Otherwise determine the c-set of $a(g(C_1))$ to which C_2 belongs and remove C_2 from it. Repeat with the roles of C_1 and C_2 exchanged. Finally determine all articulation points D on $\pi(g(C_1), g(C_2))$ in (the updated) H'. Test all c-sets for each of them whether they have to be split.

Since c-sets are disjoint, the time spent by an add is linear in the number of nodes C of H incident to a vertex in C_1 with $h(C) \neq C_1$ and the number of nodes C of H incident to a vertex of C_2 with $h(C) \neq C_2$. This is $O(m/k)$.

The time spent in $split$ or $remove$ per node D in H' is $O(\log n)$ times the number of nodes C in H incident to a vertex in D with $h(C) \neq D$. Since the nodes D are articulation points in H', each node of H is counted at most twice. Additionally the operation spends time $O((m/k)\log n)$ for $a(C_1)$ and $a(C_2)$ and $split$ takes time $O(m/k)$ to remove C from all its c-set and to replace it by C_1 and/or C_2. Thus, the total time spent is $O((m/k)\log n)$.

2.9 The amortization lemma

We show next that during a sequence of l updates after a rebuild $O(l)$ c-sets are split for $l \geq m/k$.

A similar lemma was shown in [11]. However, the previous lemma only applies to c-nodes. The lemma presented in this section applies to c-sets, which are a generalization of c-nodes.

Lemma 2.33 *During l updates in G after a rebuild at most $11l + O(m/k)$ splits of a c-set occur for $l \geq m/k$.*

Proof: Let H, H', g, and a be defined as above. 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 at most $11l$, each split of a c-set increases the number of connected components of K by at least one and no other operation decreases the number of components. Thus, there are at most $O(m/k) + 11l$ splits of c-sets during l updates in G.

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.

- For each node $a(C)$ in $\mathcal{C}$ and each c-set X at $a(C)$, K contains a red node $(a(C), X)$.
- For each node D in H, K contains a blue node D.
- For each node D in a c-set X at $a(C)$ such that $(g(D), a(C))$ is new there exists a blue node (D, X). These nodes are called *special*.
- Let D be in the c-set X at $a(C)$. If both a red node $(a(C), X)$ and a blue node (D, X) exist, there exists an edge between $(a(C), X)$ and (D, X). If (D, X) is not a blue node, there is an edge between $(a(C), X)$ and D.

Note that every edge in K corresponds to an edge of H'. Thus, connectivity in K implies connectivity in H'.

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

Next we describe each event in detail:

(A) A $split(C, C_1, C_2, u, v)$ operation. (1) 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. (2) Every red node $(a(g(C)), X)$ is replaced by a set of red nodes $(a(g(C)), \{D\})$, one for each node $D \in X$, 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. Thus each new red node has degree 1. (3) For each split c-set X' of an articulation point D on $\pi(C_u, C_v)$ in H' replace the red node $(a(D), X')$ by two red nodes, one for each new c-set.

(B) An $add(C_1, C_2)$ operation. It might add a new red node at $(a(g(C_1)), \{C_2\})$, 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 $remove(C_1, C_2)$ operation. Let X be the c-set at $a(g(C_1))$ containing C_2. Remove the edge between $(a(g(C_1)), X)$ and the corresponding node representing C_2. If (C_2, X) exists, remove it. If $X = \{C_2\}$ also remove $(a(g(C_1)), X)$. It removes the c-set X at $a(g(C_1))$, if it only contains C_2. Proceed in the same way with the roles of C_1 and C_2 reversed. Finally for each articulation point on $\pi(g(C_1), g(C_2))$ proceed as in step (A3).

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

We first show that the split of a c-set X at a node $a(C')$ of $\mathcal{C}$ increases the number of connected components by at least 1. A c-set is split (a) during $split$ operations and (b) during $remove$ operations. (a) Since in steps (A1) - (A3) repeatedly a node is replaced by a set of nodes that are connected to a subset of the neighbors of the original node, the number of connected component of K does not decrease.

Let $A = a(g(C))$. We show next that depending on whether $a(C') = A$ or not, either (A2) or (A3) actually increases the number of connected components by 1. If $a(C') = A$, then let D_1 and D_2 be in X, but belonging to different c-sets of A after the split. Note that after step (A1) a blue node representing D_1 is connected in K to a blue node representing D_2 since they both belong to X, i.e., are both connected to the red node (A, X). Furthermore, $g(C)$ is an articulation point separating $g(D_1)$ from $g(D_2)$ in H'. Since connectivity in K implies connectivity in H', every path between a blue node representing D_1 and a blue node representing D_2 in K contains a red node (A, X') for some X'.

In step (A2) all of these red nodes (A, X') are replaced by degree-1 nodes. This implies that every path between a blue node representing $g(D_1)$ and a blue node representing $g(D_2)$ is split at the node (A, X'). Hence no blue node representing D_1 is connected to any blue node representing D_2 after the split, i.e., the number of connected components of K increases by at least one.

If $a(C') \neq A$, then C belongs to the c-set X of $a(C')$ and the split of C into C_1 and C_2 partitions X. Thus C' is an articulation point in H' separating C_1 and C_2. Wlog let C_1 and C_2 be the blue nodes incident to $(a(C'), X)$. We show below in Claim 2.37 that in this case after step (A1) all paths in K connecting C_1 and C_2 contain the red node $(a(C'), X)$. In step (A3) $(a(C'), X)$ is replaced by two nodes, one representing each new c-set. All neighbors of $(a(C'), X)$ belonging to the same connected component of $K \setminus (a(C'), X)$ are connected to the same new red node and the connected components of C_1 and C_2 are connected to different new red nodes. This disconnects C_1 and C_2 and thus the number of connected components in K is increased by 1.

(b) If the removal of edge (C_1, C_2) splits X at $a(C')$ then let D_1 and D_2 be two elements of X such that $g(D_1)$ and $g(D_2)$ are the tree neighbors of $a(C')$ on $\pi(g(C_1), g(C_2))$ and wlog let D_1 and D_2 be the blue nodes incident to the red node $(a(C'), X)$ in K. (The case that either (D_1, X) or (D_2, X) are incident to $(a(C'), X)$ works in the same way). As we show in Claim 2.37 every path in K between D_1 and D_2 contains the red node $(a(C'), X)$. The same argument as above shows that updating K as in step (A3) disconnects D_1 and D_2 and, thus, increases the number of connected components of K as above.

We are left with proving Claim 2.37 and showing that the number of connected components of K does never decrease.

Claim 2.34 *Every blue node (D, X) has degree 1 in K.*

Proof: We show the claim by induction on l. For $l = 0$ the claim holds since no blue nodes (D, X) exist. Assume the claim holds for $l = i$ and consider update $i + 1$. Let $(a(C), X)$ be the unique red node incident to a blue node (D, X) before the update. If update $i+1$ does not split $a(C)$ or D the claim holds also for (D, X) after the update. If it splits $a(C)$, then a is the identity function and thus $b' = (C, D)$ is its own ancestor edge after the split. It follows that (D, X) is replaced in K by (D, X') which in turn has degree 1 in K. If update $i + 1$ splits D into D_1 and D_2 then (D, X) is replaced by at most two nodes (D_1, X) and (D_2, X), each of which is adjacent only to $(a(C), X)$. If the update inserts the new edge $b = (C, D)$ then the blue nodes (D, X) and (C, X'), the red nodes $(a(D), X')$ and $(a(C), X)$ and the edges $((D, X), (a(C), X))$ and $((C, X'), (a(D), X'))$ are added to K. Thus the claim holds in either case. ∎

Claim 2.35 *For every node C in H' each blue node representing a node D in H is incident to at most one red node $(a(C), X)$, and X is the c-set to which D belongs at $a(C)$.*

Proof: Follows from the construction of K since each node D belongs to at most one c-set at $a(C)$. ▮

Claim 2.36 *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, it follows from the construction of K that B_i and B_{i+1} have the same ancestor edge at the red node, and thus ancestor. By the transitivity of the ancestor relation the claim follows. ▮

We show next the crucial property of K while processing the split of a c-set.

Claim 2.37 *While processing the split of a c-set at $a(C')$, let C' be an articulation point in H' separating $g(C_1)$ from $g(C_2)$. Let Y_1 be a blue node representing C_1 and let Y_2 be a blue node representing C_2. If Y_1 and Y_2 are adjacent to a red node $(a(C'), X)$, then every path in K between Y_1 and Y_2 contains $(a(C'), X)$.*

Proof: The claim obviously holds if either $(g(C_1), a(C'))$ or $(g(C_2), a(C'))$ is new.
Assume next that both edges are old, i.e. $Y_1 = C_1$ and $Y_2 = C_2$ and consider a path P between them. Since connectivity in K implies connectivity in H, P must contain a red vertex $(a(C'), X')$. We are left with showing that it also contains $(a(C'), X)$.

Assume otherwise and let D_i and D_i' be the two nodes incident to the ith red node $(a(C'), X_i)$ for some c-set X_i on P. Then D_i and D_i' both belong to $X_i \neq X$. Since a split operation splits at most one c-set at $a(C')$, X_i is not split, i.e., $g(D_i)$ and $g(D_i')$ are connected in $H' \setminus C'$.

By Claim 2.36 the nodes of H represented by the blue nodes of P all have the same ancestor. Since the ancestor of C_1 and C_2 differ from the ancestor of any node in $g^{-1}(C')$, the blue node of any node in $g^{-1}(C')$ does not belong to P. Thus the subpaths of P from Y_1 to D_1, from D_i' to D_{i+1} for $1 \leq i < p$, and from D_p' to Y_2 contains no node representing any node in $g^{-1}(C')$ and, hence, correspond to a path in $H' \setminus C'$. It follows that P induces a path in H' between $g(C_1)$ and $g(C_2)$ not containing C', which is a contradiction. ▮

Claim 2.38 *The number of connected components of K never decreases.*

Proof: As discussed above, a split of a c-set does not decrease the number of connected components of K. A *remove* operation either removes c-sets or elements of c-set and thus removes nodes and edges of K or it splits c-sets. Thus it does not decrease the number of connected components of K.
An $add(C_1, C_2)$ operation might add a new blue node (C_2, X), a new red node $(a(g(C_1)), X)$, 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 of H' can become new, but not vice-versa. If this happens a new blue node is added for each element in the c-set and connected to the red node representing the c-set. Thus the number of connected components does not decrease. ▮

Claim 2.39 *After l update operations at most $11l + O(m/k)$ blue nodes exist.*

Proof: Initially there are $O(m/k)$ many blue nodes. A sequence of l update operations in G leads to at most $7l$ $split$ operations, l add operations, and l $remove$ operations. An add increases the number of blue nodes by at most 2, a $remove$ does not increase the number of blue nodes. A $split$ increases the number of non-special blue nodes by at most 1. At each point there are at most $2l$ special blue nodes. Thus there are at most $11l + O(m/k)$ blue nodes in K. ▮

Each red node is connected to a blue node and, by Claim 2.39, K contains at most $11l+O(m/k)$ blue nodes. Thus, K contains at most $11l + O(m/k)$ connected components in K after l update operations. Since none of the operations in G decreases the number of connected components of K and each split of a c-set increases the number of connected components of K by one, there can be at most $11l + O(m/k)$ splits of a c-set during l update operations. ▮

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 in or incident to the cluster C in time $O(k)$ whenever we recompute $I(C)$. To compute the blocks of G, we have to determine which blocks of different cluster graphs form the same block of G.

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

If e is a dashed edge belonging to a vertex s, then both x and y represent s. If either b or b' consists of only e, then no vertex of C is biconnected with a vertex of C'. Thus no blocks of C and C' belong to the same block of G. If b and b' consist of more than one edges, let u resp. u' be a vertex of G adjacent to s belonging to b resp. b' (any such vertex can be used). According to Lemmata 2.13 and 2.15, the blocks b and b' belong to the same block of G if and only if s does not separate u and u', i.e., if and only if (the representative of) u and u' are connected in $G(s)$ or if the representative of u and the representative of u' are connected in $\tilde{G}(s)$.

This shows that using $I(C)$ and the shared graphs, we can test in $O(1)$ time for any pair of adjacent clusters whether two of their blocks should be joined. Thus, all these tests take time $O(m/k) = O(n)$.

During a depth-first traversal of the spanning tree of H_1 we compute lists of the blocks of different clusters that have to be joined. Then we mark all the edges of blocks in the same list as belonging to the same block of G. Thus the total cost is proportional to the number of tree edges in T', which is $n - 1$.

Theorem 2.40 *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 2.6. 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 $1 \leq i < p$, 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.

Note that the nodes $x^{(i)}$ and $y^{(i)}$ are the endpoints of the tree edges on the T_2-path between the node C_u in H_2 and the node C_v in H_2. To find these nodes efficiently we keep the following data structure.

(HL7) We store a least-common ancestor data structure [8] 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.

(HL8) We store at each solid intercluster tree edge, i.e., each edge of T_2, its block in G.

Both data structures are recomputed from scratch after each update in G. The computation of (HL8) proceeds in the same way as a complete block query that ignores internal edges. Both updates take time $O(m/k)$.

We first use (HL7) 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 $x^{(p)}$. We use them to test Conditions (Q1) and (Q3) in constant time as described in Section 2.3.

Testing Condition (Q2): Condition (Q2) holds for all $1 \leq i < p$ iff each pair (e_i, e_{i+1}) belongs to the same block of G, i.e., if they all belong to the same block. This holds iff e_1 and e_p belong to the same block. Testing the latter condition takes constant time.

Theorem 2.41 *The given data structure can answer a biconnectivity query in constant time.*

The total update cost is

- time $O(k)$ for restoring the relaxed partition,
- amortized item $O(k)$ to update all cluster graphs,
- amortized time $O(k \log n + \sqrt{m} \log n)$ to update all shared graphs,
- time $O(k + (m/k) \log n)$ to update all data structures for high-level graphs (HL1) – (HL8), and
- time $O((m/k) \log n)$ to update all c-structures.

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

Theorem 2.42 *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. We modify the extended topology tree data structure of [16] 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 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 [16] the topology tree data structure is extended to maintain non-tree edges of G' and additional connectivity information at each node, called *recipe*. We use the same technique to maintain dynamic 2-vertex connectivity.

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

To add non-tree edges to the topology tree data structure we define a bundle between two clusters C and C' as follows: If neither C is an ancestor of C' nor vice verse, let $e(C, C')$ be the set of all edges between C and C'. Otherwise, assume wlog that C' is the ancestor of C. We define $e(C, C')$ to be the set of all edges incident to C whose least common ancestor in the topology tree is C'. Since we are considering an embedded graph, the edges incident to a cluster C are embedded at C in a fixed circular order. A *bundle* between a cluster C and C' is a subset of $e(C, C')$ that forms a maximal continuous subsequence in the circular order at C and C'. Note that this definition is independent of the level of the clusters and planarity guarantees that there are at most three edges of H between two clusters [16]. 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 edges of H, 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 edges of H back into one.

An edge (u, v) with $u, v \in C$ is called an *internal edge* of the cluster C. Assume all dashed internal edges of C are contracted. The *projection* of an edge (x, y) onto a tree path P is the path $\pi(x, y) \cap P$. Note that, by definition, the vertices of each cluster are connected by a subtree of T'. In the following we define the projection edge of an edge, the projection path $p(C)$, the coverage graph of C which consists of small and big supernodes of C. All these definitions are independent of the level of the cluster.

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

- If C has external degree 3, it consists of only one vertex z. Both, the *projection path* $p(C)$ and the *coverage graph* consist of only this one vertex which is a *small supernode*.

- If the external degree of a cluster C is 2, there is a unique simple tree path between the tree edges that are incident to C. This path is the *projection path* $p(C)$ of C. The *projection* $p(x)$ of a vertex x in C is the vertex closest to x on the projection path. The *projection edge* of a vertex x is the edge on $\pi(x, p(x))$ incident to $p(x)$. If $x = p(x)$, the projection edge of x is undefined. The *projection edge* of an edge (x, y) with one endpoint x in C is the projection edge of x, if it is defined and it is (x, y) otherwise. The *projection edges* of an edge (x, y) with $x, y \in C$ are the projection edge of x if it is defined and (x, y) otherwise and also the projection edge of y if it is defined and (x, y) otherwise.

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

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

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

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

     If $p > 2$, we contract the path $u_2, \ldots, u_{p-1}$ to one vertex u, called *big supernode*, and we say $u_2, \ldots u_{p-1}$ are *replaced* by the big supernode. The vertices u_1 and u_p are called *small supernodes* and the edges (u_1, u) and (u, u_p) are called *superedges*. All edges incident to $u_2, \ldots, u_{p-1}$

are now incident to u. This splits a bundle that is incident to u_1 and/or u_p and also u_i with $1 < i < p$ into up to three *subbundles*, one incident to u and the other(s) incident to u_1 and/or u_p. If the edge (u_1, u_2) (resp. (u_{p-1}, u_p)) is dashed, then the edge (u_1, u) (resp. (u, u_p)) is dashed.

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

2. After replacing all subpaths that fulfill condition 1, let $v_1, v_2, \ldots, v_q$ be a subpath of $p(C)$ such that v_1 and v_q are two small supernodes and no vertex 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 [16] by two small and one big supernode and each bundle is split into at most three *subbundles*, one incident to each small supernode and one incident to the big supernode.

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

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

3.2 Recipes

Each node in the topology tree is enhanced by a *recipe* that describes how the coverage graph of the children of the node can be created from the coverage graph of the node. The only difference in the algorithm of [16] and this biconnectivity algorithm is in the contents of the recipes. We describe our recipes in the following. A recipe contains four kinds of instructions:

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

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

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

4. *Create a new subbundle.* Create a subbundle with a specified target and number of edges, and insert it at a specified place in a subbundle list of at most two projection edges.

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

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

Case 1: C has only one child.

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

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

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

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

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

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

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

Case 4: C has two children with external degrees 2 and 1.

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

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

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

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

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

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

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

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

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

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 [16] 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 subbundles, there are $O(\log n)$ supernodes. Because the supernodes and superedges form a tree, the number of superedges is also $O(\log n)$. Each subbundles has two projection edges. Thus the total number of projection edges is $O(\log n)$.
>
> Evaluating a recipe takes time proportional to the number of supernodes or projection edges created by the recipe plus constant 'overhead' time. Thus the total expansion time is $O(\log n)$. ▮

3.2.1 Queries

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

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

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

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

Lemma 3.2 *Let u and v be two vertices of G_2 and of G_1 and let G_2 be a graph created from G_1 by*

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

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

Proof: Consider first the case that x separates u and v in G_1. To achieve that u and v are not separated by x in G_2 a cycle has to be created that contains u, x, and v. Contracting pieces of G_1 that do not contain x or removing degree-2 vertices (other 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*

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

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

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

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

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

References

[1] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph", *J.Algorithms*, 13 (1992), 33–54.

[2] D. Eppstein, Z. Galil, G. F. Italiano, "Improved Sparsification", *Tech. Report* 93-20, Department of Information and Computer Science, University of California, Irvine, CA 92717.

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

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

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

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

[7] M. L. Fredman and M. R. Henzinger, "Lower Bounds for Fully Dynamic Connectivity Problems in Graphs", *Technical Report* No. 94-1420. Department of Computer Science, Cornell University, Ithaca, NY. To appear in *Algorithmica*

[8] H. N. Gabow, "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking", *Proc. 1st Annual Symposium on Discrete Algorithms*, 1990, 434-443.

[9] Z. Galil, G. F. Italiano, "Fully Dynamic Algorithms for Edge Connectivity Problems", *SIAM J. Comput.* 21 (1992), 1047–1069.

[10] F. Harary, "Graph Theory", Addison-Wesley, Reading, MA, 1969.

[11] M. R. Henzinger, "Fully Dynamic Biconnectivity in Graphs", *Algorithmica* 13, 1995, 503–538. Also appeared as M. Rauch, "Fully Dynamic Biconnectivity in Graphs", *Proc. 33rd Annual Symp. on Foundations of Computer Science*, 1992, pages 50–59.

[12] M. R. Henzinger and V. King. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *Proc. 27th ACM Symp. on Theory of Computing*, 1995, 519–527.

[13] M. R. Henzinger and V. King, "Maintaining Minimum Spanning Trees in Dynamic Graphs", *Proc. 24th International Colloquium on Automata, Languages, and Programming*, 1997, 594-604.

[14] M. R. Henzinger and H. La Poutré, "Certificates and Fast Algorithms for Biconnectivity in Fully-Dynamic Graphs", *Proc. 3rd Annual European Symp. on Algorithms, ESA'95*, 1995, 171-184.

[15] M. R. Henzinger and M. Thorup. Improved Sampling with Applications to Dynamic Graph Algorithms. To appear in *Proc. 23rd International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 1099, Springer-Verlag, 1996.

[16] J. Hershberger, M. Rauch, S. Suri, "Fully Dynamic 2–Edge–Connectivity in Planar Graphs", *Proc. 3rd Scandinavian Workshop on Algorithm Theory*, LNCS 621, Springer-Verlag, 1992, 233–244.

[17] D. Harel and R. E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestor", *SIAM J. Comput.* 13 (1984zzz0, 338–355.

[18] K. Mehlhorn, "Data Structures and Algorithms 1: Sorting and Searching", Springer-Verlag, 1984.

[19] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, "Complexity Models for Incremental Computation", *Theoret. Comput. Science* 130, 1994, 203-236.

[20] M. H. Rauch, "Improved Data Structures for Fully Dynamic Biconnectivity", *Proc. 26 Annual Symposium on Theory of Computing*, 1994, 686-695.

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