Fully Dynamic Cycle-Equivalence in Graphs

Monika Rauch Henzinger
Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

Two edges e_1 and e_2 of an undirected graph are cycle-equivalent iff all cycles that contain e_1 also contain e_2, i.e., iff e_1 and e_2 are a cut-edge pair. The cycle-equivalence classes of the control-flow graph are used in optimizing compilers to speed up existing control-flow and data-flow algorithms. While the cycle-equivalence classes can be computed in linear time, we present the first fully dynamic algorithm for maintaining the cycle-equivalence relation. In an n-node graph our data structure executes an edge insertion or deletion in $O(\sqrt{n}\log n)$ time and answers the query whether two given edges are cycle-equivalent in $O(\log^2 n)$ time. We also present an algorithm for plane graphs with $O(\log n)$ update and query time and for planar graphs with $O(\log n)$ insertion time and $O(\log^2 n)$ query and deletion time. Additionally, we show a lower bound of $\Omega(\log n/\log\log n)$ for the amortized time per operation for the dynamic cycle-equivalence problem in the cell probe model.

1 Introduction

Two edges e_1 and e_2 of an undirected graph are *cycle-equivalent* iff all cycles that contain e_1 also contain e_2. Computing cycle-equivalence is central to many compilation problems, because the control-dependence equivalence relation of a program is the cycle-equivalence relation of the undirected version of the control-flow graph [12]. In particular, code-optimization algorithms, such as *static single-assignment form construction*, and data-flow analysis, such as determining the *subexpression availability*, can be sped up if the cycle-equivalence classes of the control-flow graph are known [13]. A third application of the control-dependence equivalence relation is in global scheduling of instructions for pipelined machines [11].

In [13], a static algorithm is used that computes the cycle-equivalence relation in linear time. Then the question is posed if the cycle-equivalence relation can be maintained efficiently during modifications of the control-flow graph. This problem is of practical significance, because it can speed up incremental compilers used in programming environments and text editors. In particular, fast query time is essential.

We present the first dynamic algorithm for maintaining the cycle-equivalence relation under edge insertions and deletions. Our data structure requires linear preprocessing time and space and tests if two edges are cycle-equivalent in $O(\log^2 n)$ query time, where n is the number of vertices in the graph. The data structure can be updated in $O(\sqrt{n}\log n)$ time after the insertion or deletion of an edge. Note that up to $\log n$ factors this is as efficient as the fastest known algorithm for the simpler problem of dynamically maintaining the connectivity relation, which requires $O(1)$ query time and $O(\sqrt{n})$ update time [3].

Dynamic cycle-equivalence is also interesting because of the relation to dynamic 3-edge connectivity: two edges e_1 and e_2 are cycle-equivalent iff (e_1, e_2) is a cut-edge pair (i.e., the removal of e_1 and e_2 disconnects the graph). While the best known dynamic algorithm for testing 3-edge connectivity requires $O(n^{2/3})$ query and update time [9], we can solve the *witness version* of 3-edge connectivity in $O(\log^2 n)$ query time and $O(\sqrt{n}\log n)$ update time: given two vertices u and v and two edges e_1 and e_2, is (e_1, e_2) a cut-edge pair witnessing that u and v are not 3-edge connected (i.e., does the removal of e_1 and e_2 disconnect u and v)? Again, note that up to $\log n$ factors our algorithm is as efficient as the best known dynamic algorithms for the witness versions of the simpler problems of 2-edge connectivity: checking if an edge is a bridge witnessing that two given vertices u and v are not 2-edge connected requires $O(\log n)$ query time and $O(\sqrt{n})$ update time [3].

We also present an algorithm for plane graphs with $O(\log n)$ time per operation and an algorithm for planar graphs with $O(\log^2 n)$ time per operation. Addi-

0272-5428/94 $04.00 © 1994 IEEE

tionally, we show a lower bound of $\Omega(\log n/\log\log n)$ on the amortized time per operation for the dynamic cycle-equivalence problem in Yao's cell probe model [17]. This is the most general model for lower bounds and encompasses all RAM algorithms. All three bounds match the best known bounds for the dynamic connectivity problem.

We call a binary relation of vertices (or edges) *convex* if whenever two vertices (edges) u and v are related, then there exists a path between u and v such that all vertices (edges) on the path are related. For example, connectivity, 2-edge connectivity, and 2-vertex connectivity are convex; k-edge connectivity and k-vertex connectivity for $k \geq 3$ are not convex. Convexity simplifies the design of dynamic algorithms: the best known dynamic algorithms for connectivity 2-edge connectivity require $O(\sqrt{n})$ update time and $O(1)$ resp. $O(\log n)$ query time; for 2-vertex connectivity, $O(\sqrt{m})$ update time and $O(1)$ query time [15]; for 3-edge connectivity, $\Omega(n^{2/3})$ update and query time [9]; for 3-vertex connectivity, $\Omega(n)$ update time [3]; for 4-edge connectivity and 4-vertex connectivity, $\Omega(n\alpha(n))$ update time [3]; for k-edge connectivity with constant $k > 4$, $\Omega(n\log n)$ update time [4]. Cycle-equivalence is not convex: two edges e_1 and e_2 can be cycle-equivalent without any other edge being cycle-equivalent to e_1 or e_2. Our algorithm, therefore, is the first known $O(\sqrt{n}\log n)$ update time algorithm for a nonconvex problem in general graphs.

Convexity supports a divide-and-conquer approach: (1) decompose the graph into small connected subgraphs, so-called *clusters*; (2) solve the problem in each cluster using a static algorithm; and (3) apply a variant of the dynamic algorithm recursively to the graph of clusters. The lack of convexity in the cycle-equivalence problem, on the other hand, makes it impossible to solve the problem in each cluster with a static algorithm alone. Rather, we maintain a spanning tree T of the graph and we solve three subproblems: (2a) test the cycle-equivalence between a tree edge and a non-tree edge; (2b) test the cycle-equivalence between a tree edge in the cluster and a tree edge outside of the cluster; and (2c) test the cycle equivalence between two tree edges in the cluster. (Two non-tree edge cannot be cycle-equivalent.) For testing (2a) we combine the ambivalent data structure of [7] with the recipe technique of [10]. For testing (2b), we introduce the following new technique, called *fast non-tree updates*: We give each edge outside the cluster cost 1 and each other edge cost 0 and maintain a minimum spanning tree of this graph using a data structure that implements insertions of edges and deletions of non-tree edges in time $O(\log n)$ and deletions of tree edges in time linear in its size, which is $O(\sqrt{n})$. Note that no edge outside the cluster is contained in the minimum spanning tree of the graph. Thus, the data structure of a cluster can be updated in $O(\log n)$ time when an outside edge changes. Since during an update one inside edge in at most two clusters and $O(1)$ outside edges in potentially all $O(\sqrt{n})$ clusters have to be updated, this technique allows us to update all data structures in time $O(\sqrt{n}\log n)$. Later, the technique of fast non-tree updates has also been used to analyze and design dynamic algorithms in a random input model [1]. For testing (2c), we maintain for every cluster ambivalent information and develop a variant of topology trees [6], called *lazy topology trees*: each node in the topology tree is labeled, but after each update the labels of only a dynamically changing subset of the nodes are updated, even though the label value at all nodes can change.

In Section 2 we present the algorithm for general graphs, in Section 3 we give the algorithm for plane and planar graphs and show the lower bound.

2 General graphs

We assume that the graph G is connected. If not, we connect it with $O(n)$ artificial edges that we update appropriately if the connected components change. A pair of edges (e_1, e_2) is a *cut-edge pair* if the removal of e_1 and e_2 disconnects the graph.

Lemma 2.1 *Two edges are cycle-equivalent iff they are a cut-edge pair in the graph.*

Let T be a spanning tree of G, let T' (T'') be a spanning forest of $G \setminus T$ ($G \setminus \{T \cup T'\}$), and let $G'' = T \cup T' \cup T''$. (When refering to a tree edge, we mean an edge of T.) As shown in [14] two edges are a cut-edge pair in G iff they are a cut-edge pair in G''. Lemma 2.1 immediately implies the following lemma.

Lemma 2.2 *Two edges of G are cycle-equivalent in G iff they both are contained in G'' and they are cycle-equivalent in G''.*

This implies that it suffices to test cycle-equivalence in a graph with $O(n)$ edges. Using a dynamic connectivity data structure for G, for $G \setminus T$ and for $G \setminus \{T, T'\}$ we maintain T, T', and T'' and, thus G'' dynamically. We present in the next section an algorithm for maintaining cycle-quivalence in G'' with update time $O((k + n/k + \log n)\log n)$ and query time $O(\log^2 n)$. Choosing $k = \sqrt{n}$ gives the following result.

Theorem 2.3 *The cycle-equivalence of edges in a graph can be maintained in time $O(\log^2 n)$ per query and $O(\sqrt{n}\log n)$ per update.*

Lemma 2.4 *Two edges e_1 and e_2 are a cut-edge pair witnessing that to vertices u and v are not 3-edge connected iff e_1 and e_2 are a cut-edge pair and either e_1 or e_2 lies on the tree path between u and v.*

Theorem 2.5 *Our data structure can answer a witness-query in time $O(\log^2 n)$.*

Note that 2 non-tree edges cannot be cycle-equivalent. Thus, it suffices to test the cycle-equivalence between two tree edges (Section 2.2) and the cycle-equivalence between a tree edge and a non-tree edge (Section 2.3). First (Section 2.1) we give some basic definition and data structures.

2.1 Basics

We present first the *topology trees* data structure [6]. Given a graph G with spanning tree T we expand every vertex of G with degree $d > 3$ into d vertices that are connected by a chain of $d-1$ edges. We naturally expand T to be a spanning tree of the expanded graph G'. Note that two edges are cycle-equivalent in G' iff they are cycle-equivalent in G.

A *cluster partition of order k* of T is a partition of T into $O(m/k)$ subtrees, each consisting of $O(k)$ edges and vertices. Each subtree is called a *level-0 cluster* or simply *cluster*. An edge with (exactly) 1 endpoint in C is an *incident* edge of C. The *tree degree* of a C is the number of tree edges incident to C. A *restricted partition of order k* is a cluster partition where the tree degree of all clusters is at most 3 and if the tree degree of a cluster is 3, then the cluster consists of only one vertex and this vertex is not incident to any non-tree edges.

An edge with 2 endpoints in C is an *internal* edge of C, an edge with 0 or 1 endpoint in C is an *external* edge of C. Note that a cluster with tree degree 3 does not have any internal edges. The *tree path* $TP(C)$ of C with tree degree 2 is the tree path connecting the two tree edges incident to C. If C has tree degree 1 or 3, its tree path consists of the endpoint of the tree edge(s) incident to C. We denote by $\pi(u,v)$ the tree path between u and v and by $C(u)$ the cluster containing u. A non-tree edge (x,y) *covers* a tree edge e iff e lies on the tree path $\pi(x,y)$ between x and y.

Let a *high-level graph* H of G consist of one vertex per cluster and an edge between two clusters C_1 and C_2 iff there exists an edge between a vertex of C_1 and C_2. The spanning tree of G induces a spanning tree on H. An edge incident to two clusters is a *high-level edge* or *h-edge*.

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 X is the union of two clusters X_1 and X_2 at level $i-1$, then X_1 and X_2 are the children of X. If X consists of one level-$(i-1)$ clusters X at level i, then X_1 is the only child of X in TT. We call the vertices of TT *nodes*. We say that a vertex x of G *belongs* to a node X of TT and that X *contains x if x belongs to the cluster corresponding to X*.

2.2 The cycle-equivalence of two tree edges

To test the cycle-equivalence of 2 tree edges we distinguish between testing (1) 2 h-edges, (2) an internal and an external edge of a cluster C, and (3) 2 edges internal to C. We build 3 different data structures, called *external*, *combined*, and *internal* data structure and use combinations of them to solve cases (1) - (3).

In the rest of the section let us denote the 2 query edges by (u,v) and (x,y) and assume that v and y lie on $\pi(x,u)$ (i.e., $v,y \in \pi(x,u)$). If (x,y) is a tree edge, we denote by the *subtree of x* in $T \setminus (x,y)$ the subtree of $T \setminus (x,y)$ containing x. If x is contained in a cluster C, then the *subtree of x in $C \setminus (x,y)$* is the part of the subtree of x in $T \setminus (x,y)$ that is contained in C. If $e = (u,v)$ and $e' = (x,y)$ are 2 edges of T with $v,y \in \pi(u,x)$, then the *subtree of e and e'* is the subtree of $T \setminus \{e,e'\}$ containing v and y.

We need to test 4 quite technical properties, called Type i queries for $i \leq 4$. The *external data structure* tests the 1st, the *combined data structure* tests the 2nd and 3rd, and the *internal data structure* tests the last.

Type 1 query: Let $e_1 = (x,y)$, e_2, and e_3 be h-edges of T such that the subtree T_1 of x in $T \setminus (x,y)$ and the subtree T_2 of e_2 and e_3 are vertex-disjoint. Is there a non-tree edge between T_1 and T_2?

Type 2 query: Let (x,y) be an edge on $TP(C)$ and let (u,v) be an edge with $u \notin C$ and $v,y \in \pi(u,x)$. Is there a non-tree edge between (1) the subtree of x in $C \setminus (x,y)$ and the subtree of (x,y) and (u,v) or (2) between the subtree of y in $C \setminus (x,y)$ and either the subtree of u in $T \setminus (u,v)$ or the subtree of x in $T \setminus (x,y)$?

Type 3 query: Let (x,y) be an internal edge of C not on $TP(C)$ and let (u,v) be an edge with $u \notin C$ and

$v, y \in \pi(u, x)$. Is there a non-tree edge (1) between the subtree of x in $C \setminus (x, y)$ and the subtree of (x, y) and (u, v) or (2) between the subtree of y in $C \setminus (x, y)$ and the subtree of u in $T \setminus (u, v)$?

Type 4 query: Let (x, y) be an internal edge of C no on $TP(C)$ and let (u, v) be an edge on $TP(C)$ with $y, v \in \pi(x, u)$. Let (z, w) and (z', w') be the tree edge incident to C with $w, w' \notin C$ and $u, v \in \pi(x, w')$. Is there a non-tree edge (1) between the subtree of w in $H \setminus C$ and either the subtree of x in $C \setminus (x, y)$ or the subtree of u in $C \setminus (u, v)$ or (2) between the subtree of w' in $H \setminus C$ and the subtree of (x, y) and (u, v) in C?

We first split the cycle-equivalence problem into suitable subcases and show how to solve each of them by a combination of Type i queries. Then we present 3 data structures to answer the Type i queries.

Two h-edges

Testing 2 h-edges requires one query in the external data structure:

Lemma 2.6 *Let (x, y) and (u, v) be two external tree edges with $y, v \in \pi(x, u)$. Then (x, y) and (u, v) are cycle-equivalent iff there is no non-tree edge between the subtree of (x, y) and (u, v) and either the subtree of x in $T \setminus (x, y)$ or the subtree of u in $T \setminus (u, v)$.*

One internal and one external edge of C

Testing an internal and an external edge of C requires tests in the external data structure (Condition 1 and 2) and in the combined data structure (Condition 3):

Lemma 2.7 *Let (x, y) be an edge internal to the cluster C, but not on $TP(C)$, and let (u, v) be an external edge of C with $y, v \in \pi(x, u)$. Let e_1 be the tree edge on $\pi(u, x)$ incident to C and let $e_2 = (z, w)$ with $w \notin C$ be the other tree edge incident to C (if it exists). Then (x, y) and (u, v) are cycle-equivalent iff*

1. *there is no non-tree edge between the subtree of u in $T \setminus (u, v)$ and the subtree of (u, v) and e_1,*
2. *there is no non-tree edge between the subtree of u in $T \setminus (u, v)$ and the subtree of w in $T \setminus e_2$, and*
3. *there is no non-tree edge between (1) the subtree of u in $T \setminus (u, v)$ and the subtree of y in $C \setminus (x, y)$ and (2) between the subtree of x in $C \setminus (x, y)$ and the subtree of (u, v) and (x, y).*

Lemma 2.8 *Let (x, y) be an edge on $TP(C)$ and let (u, v) be an external edge with $y, v \in \pi(x, u)$. Let e_1 be the tree edge on $\pi(u, x)$ incident to C and let $e_2 = (z, w)$ with $w \notin C$ be the other tree edge incident to C. Then (x, y) and (u, v) are cycle-equivalent iff*

1. *there is no non-tree edge between the subtree of u in $T \setminus (u, v)$ and the subtree of (u, v) and e_1,*
2. *there is no non-tree edge between the subtree of w and the subtree of (u, v) and e_1.*
3. *(1) there is no non-tree edge between the subtree of y in $C \setminus (x, y)$ and either the subtree of u in $T \setminus (u, v)$ or the subtree of x in $T \setminus (x, y)$ and (2) there is no edge between the subtree of x in $C \setminus (x, y)$ and the subtree of (u, v) and (x, y),*

Two internal edges of C

We distinguish the case that (1) either both query edges lie on $TP(C)$ or neither does and (2) that one query edge lies on $TP(C)$ and the other does not. Let $G'(C)$ be the graph consisting of all vertices of C and 1 vertex representing all vertices outside of C. The vertices of $G'(C)$ are connected by all edges incident to vertices of $G'(C)$. Thus, $G'(C)$ is created from G by collapsing all vertices outside C to one vertex. Lemma 2.9 shows that Case (1) can be reduced to testing cycle-equivalence in $G'(C)$, a graph with $O(k)$ edges and vertices. Cycle-equivalence in $G'(C)$ can be maintained using the static algorithm in time $O(k)$ per update and $O(1)$ per query.

Lemma 2.9 *Let (x, y) and (u, v) be two edges of C with $y, v \in \pi(x, u)$ and such that either both edges lie on $TP(C)$ or neither lies on $TP(C)$. The edges (x, y) and (u, v) are cycle-equivalent in G iff they are cycle-equivalent in $G'(C)$.*

Let $G''(C)$ be the graph induced by all vertices of C and all edges internal to C. As in the case of $G'(C)$, $G''(C)$ can be maintained in time $O(k)$ per update and $O(1)$ per query. In Case (2) we test cycle-equivalence using the external data structure for Condition 1 of the next lemma, using $G''(C)$ for Condition 2, and a Type4 query for Condition 3–5.

Lemma 2.10 *Let (x, y) be an internal edge, not on $TP(C)$ and let (u, v) be an edge on $TP(C)$ with $y, v \in \pi(x, u)$. Let $e_1 = (w, z)$ and $e_2 = (w', z')$ be the tree edges incident to C with $w, w' \notin C$ such that $(u, v) \in \pi(x, w')$. The edges (x, y) and (u, v) are cycle-equivalent in G iff*

1. *there is no non-tree edge between the subtree of w' in $T \setminus e_2$ and the subtree of w in $T \setminus e_1$.*
2. *(x, y) and (u, v) are cycle-equivalent in $G''(C)$*
3. *there is no non-tree edge between the subtree of u in $C \setminus (u, v)$ and the subtree of w in $T \setminus e_1$*
4. *there is no non-tree edge between the subtree of w' in $T \setminus e_2$ and the subtree of (x, y) and (u, v) in C,*
5. *there is no non-tree edge between the subtree of x in $C \setminus (x, y)$ and the subtree of w in $T \setminus e_1$*

2.2.1 The external data structure

Given 3 h-edges $e_1 = (x, y)$, e_2, and e_3 that are tree edge such that the subtree of x and the subtree of e_2 and e_3 are vertex-disjoint, the external data structure allows to test if there is a non-tree edge between the subtree of x in $T \setminus (x, y)$ and the subtree of e_2 and e_3.

We maintain a topology tree TT for G and keep at each node X of TT a copy $TT(X)$ of TT (called *individual topology tree* or *i-tree*) in which we store some of the edges leaving X. To be precise at the node representing the cluster Y which is not an ancestor or descendant of X in $TT(X)$ we store an edge between Y and X if such an edge exists and 0 otherwise.

All i-trees are computed bottom-up. If X is a level-0 cluster, $TT(X)$ contains at the node Y which is not an ancestor of X one edge between X and Y if such an edge exists and 0 otherwise. If X is an internal node of TT and it has one child, the labels of $TT(X)$ are identical to the label of its child. If X has two children X_1 and X_2, then a node of $TT(X)$ is labeled with an edge iff the corresponding node is labeled in $TT(X_1)$ or $TT(X_2)$. Thus, building $TT(X)$ takes time $O(k + m/k)$ in both cases. Note that updating $TT(X)$ if the structure of TT has changed takes time $O(\log n)$ given (1) the edges incident to X if X is a level-0 cluster or (2) i-tree of the children of X if X is not a level-0 cluster: First $TT(X)$ is split and joined appropriately, then the labels of the $O(1)$ new leaves of $TT(X)$ are determined from (1) the edges incident to X or (2) the i-trees of the children of X. Finally the labels of the $O(\log n)$ new non-leaf nodes of $TT(X)$ are computed bottom-up in $TT(X)$.

After an update (a, b) operation the i-trees of all clusters are rebuilt bottom-up. Since TT has depth $O(\log n)$, this takes time $O((k + m/k) \log n)$. If the update changes TT, then all $O(m/k)$ i-trees are updated, each in time $O(\log n)$. Thus the total time to update the external data structure after an update operation is $O((k + m/k) \log n)$.

To answer a query note that the subtree of x in $T \setminus (x, y)$ (the subtree of e_2 and e_3) is represented by $O(\log n)$ subtrees of TT. We call the roots of these subtrees the *topology nodes representing x (topology nodes representing the subtree of e_2 and e_3)*.

Lemma 2.11 *Let $R_1, \ldots, R_l$ be the topology nodes representing x. There is an edge between the subtree of x in $T \setminus (x, y)$ and the subtree of e_2 and e_3 iff there is an edge at a node X that is an internal node in $TT(R_i)$ for of a node R_i and represents the subtree of e_2 and e_3 in $TT(R_i)$.*

Theorem 2.12 *The external data structure can answer a query in time $O(\log^2 n)$ and can be updated in time $O((k + m/k) \log n)$.*

2.2.2 The combined data structure

Description

The combined data structure answers Type2 and Type3 queries using a data structure with fast non-tree updates. Since in this case C has an internal edge, the tree degree of C is 1 or 2.

We define a graph $G(C)$ consisting of one vertex for each vertex of C, one vertex for each edge incident to C (called *e-vertex*), and all edges incident to vertices of C with cost 0 such that an edge (a, b) with $a \in C$ and $b \notin C$ is represented by an edge between a and the e-vertex of (a, b). Since each cluster contains $O(k)$ edges, $G(C)$ has size $O(k)$. Additionally, $G(C)$ contains *artificial* edges with cost 1 that are defined using the following order, called *Eulerian tour order (ET-order)*:

We fix a tree degree-1 cluster S. Let $s \in S$ be the endpoint of the tree edge incident to S. We start an Eulerian tour of the spanning tree of H at S and create a list L of all clusters (with multiple occurrences) in the order in which they are visited. We assign to each cluster C up to 3 numbers $num_1(C)$, $num_2(C)$, and $num_3(C)$ such that $num_i(C) = j$ iff the ith visit of C is on position j of L. We also assign each vertex $x \in C$ a number $num(x)$ such that $num(x) = i$ iff x is the ith vertex visited by the following Eulerian tour of the spanning tree of C: Let e be the tree edge that is incident to a cluster C and that is used by the Eulerian tour to visit C for the first time if $C \neq S$, and let e be the tree edge incident to C if $C = S$. The tour starts at e and it visits all vertices before it crosses the other tree edges incident to C (if the tree degree is > 1). The ET-order on the vertices of G is the lexicographic combination $(num_1(C(x)), num(x))$ and is denoted by $\geq_{ET}$. If $x \geq_{ET} y$, then the first visit of x occurs after the first visit to y in the above Eulerian tour of T. The edges incident to C are in *ET-order* if they are ordered in the ET-order of their endpoints that are not in C (ambiguities are resolved in an arbitrary but fixed way).

Let w be a vertex connected to C by a tree edge. Let $L(w)$ be the list of all edges incident to C and to the subtree of w in $H \setminus C$. The e-vertices of all edges in $L(w)$ are connected in $G(C)$ by a chain of *artificial* edges such that e_1 and e_2 of $L(w)$ are connected by an artificial edge iff e_1 is the immediate successors of e_2 in the ET-order.

Assuming the vertices are in ET-order, the following lemmata show how to answer Type 2 or 3 queries.

Lemma 2.13 *Let (x,y) be an edge on $TP(C)$, let (u,v) be a tree edge with $u \notin C$ such that $v,y \in \pi(u,x)$, let G_u be the subtree of u in $T \setminus (u,v)$ and let G_{vy} be the subtree of (u,v) and (x,y).*

If $s \notin G_u$, then let

- *u' be the smallest vertex in G_u incident to C*
- *u'' be the largest vertex in G_u incident to C,*
- *v' be the largest vertex in G_{vy} incident to C or in C such that $v' <_{ET} u$,*
- *v'' be the smallest vertex in G_{vy} incident to C such that $v'' >_{ET} u$.*

If $s \in G_u$, then let

- *u' be the largest vertex in G_u incident to C such that $u' \leq_{ET} u$,*
- *u'' be the smallest vertex in G_u incident to C such that $u'' >_{ET} u$,*
- *v' be the smallest vertex in G_{vy} incident to C or in C,*
- *v'' be the largest vertex in G_{vy} incident to C.*

(It is possible that u'' and v'' do not exist.) There is an edge between (1) the subtree of x in $C \setminus (x,y)$ and the subtree of (u,v) and (x,y) or (2) between the subtree of y in $C \setminus (x,y)$ and the subtree of u in $T \setminus (u,v)$ or the subtree of x in $T \setminus (x,y)$ iff (x,y) is covered in $\{G(C) \cup (u',x)\} \setminus \{(u',v'),(u'',v'')\}$.

Lemma 2.14 *Let (x,y) be an internal edge of C not on $TP(C)$, let (u,v) be a tree edge with $u \notin C$ such that $v,y \in \pi(u,x)$. Let u',u'',v',v'' be defined as in Lemma 2.14.*

There is an edge between (1) the subtree of x in $C \setminus (x,y)$ and the subtree of (u,v) and (x,y) or (2) between the subtree of y in $C \setminus (x,y)$ and the subtree of u in $T \setminus (u,v)$ iff (x,y) is covered in $\{G(C) \cup (u',x)\} \setminus \{(u',v'),(u'',v'')\}$.

Data structure

The data structure at each cluster C consists of 2 parts: (1) We keep a balanced search tree of all edges incident to C in ET-order. (2) We keep the following *data structure for fast non-tree updates FAST(C)*: We maintain the minimum spanning tree of $G(C)$ in a dynamic tree data structure [16] and keep for each tree edge the *coverage number*, the number of non-tree edges covering it. Every insertion or deletion of an edge (a,b) increases or decreases the coverage number of all edges on $\pi(a,b)$ by one. Thus, it takes time $O(\log n)$. Since the minimum spanning tree of $G(C)$ does not contain any artificial edges, insertions or deletions of artificial edges in *FAST*(C) take only time $O(\log n)$.

Updates

We describe how to update each of the 2 parts: (1) If the ET-order inside a cluster C' changes, we delete all edges incident to C' from all balanced search trees of the other clusters and reinsert them in the new order. Since $O(k)$ edges are incident to C', this takes time $O(k \log n)$. If the ET-order of H changes, note that the change is structured as follows: Let $1,2,\ldots,p$ be the labels of the clusters before the change. Then there exist labels $i_1 < i_2$ and i_3 such that the new order is either $1,2,\ldots,i_1-1,i_2+1,i_2+2,\ldots,i_3,i_1,i_1+1,\ldots,i_2,i_3+1,i_3+2,\ldots,p$ or $1,2,\ldots i_3,i_1,i_1+1,\ldots,i_2,i_3+1,i_3+2,\ldots,i_1-1,i_2+1,i_2+2,\ldots,p$. Thus, updating a balanced search tree requires a constant number of splits and joins and takes time $O(\log n)$.

(2) Whenever an edge incident to a cluster is inserted or deleted, we rebuild its *FAST*-data structure from scratch in time $O(m/k+k)$. Since each edge is incident to at most 2 clusters, at most 2 *FAST*-data structures have to be rebuilt. If the ET-order inside a cluster C' changes, we delete all artificial edges incident to e-vertices of edges between C' and C in *FAST*(C). Then we reconnect the e-vertices with artificial edges in the new order. Updating all *FAST*-data structures takes time $O(k \log n)$. If the ET-order of H changes, then as shown in (2) a constant number of artificial edges of *FAST*(C) have to be modified (and the balanced search tree of C provides these edges). Hence, updating *FAST*(C) takes time $O(\log n)$, updating all *FAST*-data structures takes time $O(k \log n)$. Thus, updating all combined data structures takes time $O((k+m/k)\log n)$.

Queries

To answer Type 2 and Type 3 queries we determine the vertex u', u'', v', and v'' (as defined in Lemma 2.13) in time $O(\log n)$ using the balanced search tree of the edges incident to C, the num_i labels of the clusters, and the *num* label of the vertices. Then we delete (u',v') and (u'',v'') from the *FAST*-data structure of $G(C)$, insert (x,u') with cost 1, and test the coverage of (x,y). Afterwards we restore $G(C)$. Since (u',v'), (u'',v'') and (x,u') are non-tree edges of $G(C)$, this takes $O(\log n)$ time.

Theorem 2.15 *Each query in a combined data structure can be answered in time $O(\log n)$. After an update in G it takes time $O((k+m/k)\log n)$ to update all combined data structures.*

2.2.3 The internal data structure

If the query edges (x, y) and (u, v) lie in the same cluster C with (x, y) not on $TP(C)$ and (u, v) on $TP(C)$, we have to answer a Type 4 query which corresponds to testing Conditions 3–5 of Lemma 2.10. Since (x, y) is not on $TP(C)$, the tree degree of C is 2 and, thus, $s \notin C$ (see Section 2.2). A vertex x is *covered* iff the removal of x does not disconnect the graph.

Lemma 2.16 *Condition 1 of Lemma 2.10 holds iff the cluster C is not covered in H.*

We only test for Condition 3–5 if Condition 1 holds: Thus, we maintain ambivalent information to test Condition 3 and 4, and a *lazy topology tree* to test Condition 5 only for uncovered clusters. To determine at update time all uncovered clusters we maintain a dynamic biconnectivity data structure [3, 15]. It is updated in $O(m/k)$ time per insertion or deletion.

Condition 3 and 4

The non-tree edges incident to C are in *ET'-order* if they are ordered in the ET-order of their endpoint in C (ambiguities are resolved in an arbitrary but fixed way). Let the *projection* $proj(e')$ of a non-tree edge $e' = (a, b)$ with $a \in C$ and $b \notin C$ onto $TP(C)$ be the vertex on $TP(C)$ which is closest to a. Let $sub(e')$ be the subtree of a in $C \setminus proj(e')$ and let $other(e')$ be the set of all edges incident to C whose endpoint in C does not lie in $sub(e')$. Since the degree of $proj(e')$ is 3, the projection of an edge in $other(e')$ is not $proj(e')$.

Data structure

For each pair of clusters C and C' and each tree edge e incident to C we maintain ambivalent information [7] in the form of 3 non-tree edges $ambiv_i(C, C', e)$ for $i = 1, 2, 3$ (if they exist): Assuming that e lies on $\pi(C, C')$ let $ambiv_1(C, C', e)$ ($ambiv_2(C, C', e)$) be the edge e' between C and C' that (1) covers the maximum number of edges on $TP(C)$ and that (2) is the first (last) edge in the ET'-order in $sub(e')$. Let $ambiv_3(C, C', e)$ be one of the edges of $other(ambiv_1(C, C', e))$ that covers the largest number of edges on $TP(C)$, assuming again that e lies on $\pi(C, C')$. It is possible that $ambiv_2(C, C', e) = ambiv_1(C, C', e)$.

For each uncovered cluster C and each tree edge e incident to C we maintain in addition up to 3 edges $max_i(C, e)$ for $i = 1, 2, 3$ (if they exist): The edge $max_1(C, e)$ ($max_2(C, e)$) is the non-tree edge e' incident to C and covering e that (1) covers the maximum number of edges on $TP(C)$ and that (2) is the first (last) edge in the ET'-order in $sub(e')$. It is possible that $max_1(C, e) = max_2(C, e)$. Let $max_3(C, e)$ be an edge of $other(max_1(C, e))$ that (1) covers e and (2) covers the largest number of edges on $TP(C)$ of all edges in $other(max_1(C, e))$.

Updates

Since the edges $ambiv_i(C, C', e)$ depend only on the spanning tree of C and the edges between C and C', an update changes the *ambiv*-values only for clusters whose incident edges change or whose spanning tree changes. There are only $O(1)$ such clusters [6]. Computing the *ambiv*-values of a cluster takes time $O(k)$. Thus, all *ambiv*-values are updated in time $O(k)$.

Note (1) that given the num_i-labels of the cluster all $max_i(C, .)$ edges can be computed from the $ambiv_i(C, ., .)$ edges in time proportional to the number of neighboring clusters of C. Note (2) that after an update (a, b) operation the max_i-values only change for clusters on $\pi(a, b)$. Note (3) that we only maintain the max_i-values for clusters that are uncovered after the operation. The number of neighboring clusters of uncovered clusters on $\pi(a, b)$ sums to $O(m/k)$, since each cluster of H is incident to at most two of them. Thus, updating all max_i values takes time $O(m/k)$.

Queries

The max_i values are used to test Condition 3 and 4 in time $O(\log n)$ by executing a binary search on $TP(C)$ as follows:

Lemma 2.17 *Let x, y, u, v, w, w', z, and z' be defined as in Lemma 2.10.*

- *Condition 3 holds iff the endpoint of $max_1(C, e)$ does not lie in the subtree of u in $C \setminus (u, v)$.*
- *Condition 4 holds iff the endpoints of $max_1(C, e')$ and $max_2(C, e')$ are contained in the subtree of x in $C \setminus (x, y)$ or in the subtree of u in $C \setminus (u, v)$ and if the endpoint of $max_3(C, e')$ is contained in the subtree of u in $C \setminus (u, v)$.*

Condition 5

Let C be a level-0 cluster and w and w' be the vertices that are connected to C by tree edges. We denote by $edge(w)$ the set of edges incident to C whose other endpoint lies in the subtree containing w in $H \setminus C$. Given a topology tree $TT(C)$ of C let X be a node of $TT(C)$. We say X is *w-sided* if an edge of $edge(w)$ is incident to a vertex of X. A node X is *double-sided* if it is w-sided and w'-sided. If (x, y) is a tree edge in C, the subtree of x in $C \setminus (x, y)$ is represented by $O(\log n)$ subtrees of $TT(C)$. The roots of these subtrees are the *topology nodes representing x in $TT(C)$*.

Lemma 2.18 *Let x, y, u, v, w, w', z, and z' be defined as in Lemma 2.10. Condition 5 does not hold iff one of the topology nodes representing x in $TT(C)$ is w-sided.*

Proof: If a topology node R representing x is w-sided, there exists an edge e with one endpoint belonging to R and with the other endpoint in the subtree of w in $T \setminus (w,z)$. Since every vertex that belongs to R lies in the subtree of x in $C \setminus (x,y)$, Condition 5 does not hold.

If there is an edge e from the subtree of x in $C \setminus (x,y)$ to the subtree of w in $T \setminus (w,z)$, then let R be the topology node representing x that contains the endpoint of e in C. Obviously R is w-sided. ∎

Lemma 2.18 suggests a way to test for Condition 5: Test if one of the $O(\log n)$ topology nodes representing x is w-sided. In the following we present a data structure that executes this test in time $O(\log^2 n)$.

Data Structure

For each cluster C we keep a bit *Cond* 3&4 (see below) and a *lazy topology tree* $TT(C)$. At each node X of $TT(C)$ we keep (1) a list $L(C,C',X)$ for each level-0 cluster $C' \neq C$ containing all edges between the subgraph represented by X and C'; (2) a list $L(X)$ of all non-emtpy trees $L(C,.,X)$; (3) the number $num(X)$ of level-0 clusters whose vertices are incident to a vertex belonging to X; (4) the bits *marked* and *doublesided* (see below). The two bits of (4) are updated only for a dynamically changing set of nodes of $TT(C)$. The number of edges stored in all lists $L(C,C',.)$ is $O(v(C,C')\log n)$, where $v(C,C')$ is the number of edges between C and C'.

Lemma 2.19 *If $num(R) = 0$ for a node R of $TT(C)$, then R is not w-sided.*

Thus, we are left with testing if a node R with $num(R) > 0$ is w-sided: Using the num_i labels of Section 2.2 we can determine in constant time if a cluster $C' \neq C$ lies in the subtree of w or of w' in $H \setminus C$. If $L(C,C',X)$ is non-empty and C' lies in the subtree of $\tilde{w}$ with $\tilde{w} \in \{w,w'\}$, it follows that X is $\tilde{w}$-sided. Thus, using $L(X)$ we can determine in constant time a vertex $\tilde{w}$ such that X is $\tilde{w}$-sided. If we have some more information about X, namely if we know that X is not double-sided, then this test determines in constant time if X is w-sided. Note that without this information if it takes time $O(num(X))$ to determine if X is w-sided or not. Since it takes also time $O(num(X))$ to determine if X is double-sided, it is too expensive to determine for each node X in $TT(C)$ if it is double-sided. Thus, after each update we mark a dynamically changing set of $O(\log n)$ nodes in $TT(C)$ and we determine for all marked nodes and their children if they are double-sided. We keep at each node of $TT(C)$ two bits, called *marked* and *doublesided* such that the following invariant is fulfilled: *If C is not covered and either X or its parent is marked, then doublesided(X) is set to 1 iff X is double-sided.* We discuss below which nodes are marked.

We call a node of $TT(C)$ *red* if it contains a vertex of $TP(C)$ and a vertex not on $TP(C)$, and *white* otherwise. We store at each node a bit indicating its color when $TT(C)$ is built. (This increases the time to build $TT(C)$ only by a constant factor.) If Lemma 2.10 applies to C, the edge (x,y) does not lie on $TP(C)$ and, thus, the root of $TT(C)$ is red. The following lemma shows that the double-sided, red nodes in C form two paths in $TT(C)$.

Lemma 2.20 *Let x, y, u, v, w, w', z, and z' be defined as in Lemma 2.10. If Condition 3 and 4 of Lemma 2.10 hold, then every double-sided, red node contains either (x,y) or (u,v).*

Proof: Assume there exists a double-sided, red node X that does not contain (x,y) or (u,v). Then the subtree represented by X either lies completely (1) in the subtree of (x,y) and (u,v) or (2) in the subtree of u in $C \setminus (u,v)$ or (3) in the subtree of x in $C \setminus (x,y)$. Since Condition 3 and 4 hold and X is double-sided, the first and second case are not possible. The third case is not possible, since X is red and the subtree of x in $C \setminus (x,y)$ does not contain vertices from $TP(C)$. ∎

After each update we determine for ever uncovered cluster C if the red, double-sided nodes from two paths in $TT(C)$. If not, we set the bit *Cond* 3&4(C) to 1. Otherwise, we set this bit to 0 and call the nodes with lowest level on each of the paths X_1 and X_2 such that X_1 contains (x,y). Note that the definition of X_1 and X_2 does not depend on the query edges and can, thus, be computed after each update.

The *marked* nodes are all ancestors and all children of X_1 and X_2 and some of the descendants of these children such that the following invariant is maintained: *A node X is marked iff its parent is marked, X is double-sided and the sibling of X is not double-sided.* Note that $O(\log n)$ nodes are marked, since at most one child of each marked proper descendent of X_1 or X_2 is marked. We first show a technical lemma.

Lemma 2.21 *If a node X is double-sided and unmarked and its parent is marked, then (1) either the parent in an ancestor of X_1 or X_2 or (2) the parent is a proper descendant of X_1 or X_2 and the sibling of X is double-sided.*

The highest unmarked ancestor of a node R in $TT(C)$ is denoted by $HU(R)$. This is the lowest ancestor of R for which we know at query time if it is double-sided or not.

Lemma 2.22 *Let x, y, u, v, w, w', z, and z' be defined as in Lemma 2.10 and let R be a topology node representing x with $num(R) > 0$. If Condition 3 and 4 of Lemma 2.10 hold, then the following is true: If $HU(R)$ is not w-sided, then R is not w-sided. If $HU(R)$ is w-sided, then there exists a topology node R' representing x that is w-sided.*

Proof: Let $A = HU(R)$. If R is w-sided, then A is w-sided, since A is an ancestor of R.

If A is w-sided, but not w'-sided, then all edges incident to A, and, thus all edges incident to R belong to *edge*(w). Hence, R is w-sided. We are left with the case that A is double-sided. Again we distinguish two cases: (1) If A does not contain x, then A contains only vertices of the subtree of x in $C\setminus(x,y)$. Since A is double-sided, there exists a topology node R' representing x that is w-sided. (2) If A contains x, then, by Lemma 2.21, A and the parent of A are proper descendants of X_1, since A is unmarked. Since the parent of A is double-sided and X_1 is the lowest double-sided node that is red and contains x, it follows that the parent of A is white. Together with the fact that A contains x, this implies that the parent of A does not contain any vertices on $TP(C)$, especially it does not contain u. Lemma 2.21 shows that the sibling $s(A)$ of A is double-sided as well. Since $s(A)$ contains neither x nor u, all vertices of $s(A)$ are completely contained either (1) in the subtree of x in $C \setminus (x,y)$ or (2) in the subtree of u in $C\setminus(u,v)$ or (3) in the subtree of (u,v) and (x,y) of C. Since Condition 3 and 4 hold, case (2) and (3) are not possible. Thus, all vertices of $s(A)$ are contained in the subtree of x in $C \setminus (x,y)$. Since $s(A)$ is double-sided, it follows that there exists a topology node R' representing x that is w-sided. ∎

Lemmata 2.19, 2.18, and 2.22 imply the following lemma.

Lemma 2.23 *Let x, y, u, v, w, w', z, and z' be defined as in Lemma 2.10. If Condition 3 and 4 of Lemma 2.10 hold, then Condition 5 of Lemma 2.10 holds iff for all topology nodes R representing x in $TT(C)$ with $num(R) > 0$ the node $HU(R)$ is not w-sided.*

The topology tree $TT(C)$ has depth $O(\log n)$ which implies that there are $O(\log n)$ topology nodes representing x. Finding the highest unmarked ancestor of a topology node takes time $O(\log n)$. Thus, it takes time $O(\log^2 n)$ to find the highest unmarked ancestor for all topology nodes that represent x. Since we know at query time for each highest unmarked ancestor, if it is double-sided, we can determine in constant time if it is w-sided. Thus, Lemma 2.23 provides a test for condition 5 in time $O(\log^2 n)$.

Updates

After each update we rebuild the topology tree of the $O(1)$ clusters whose spanning tree has changed. The list $L(C,C',X)$ can be built in time linear in its size if the lists $L(C,C',.)$ of the children of X are given. Thus, whenever $TT(C)$ is rebuilt or an edge incident to C is inserted, we can compute all lists $L(C,.,.)$, the lists $L(X)$, and $num(X)$ from scratch in time $O(k \log n)$. Since only $O(1)$ clusters are affected, updating all trees $TT(C)$ and labels (1) - (3) takes time $O((k + m/k)\log n)$.

To update the *doublesided* and *marked* bits after an update (a,b) operation, note (1) that the *doublesided* bits only change for clusters on $\pi(a,b)$ and (2) that only the *doublesided* and *marked* bits of uncovered clusters have to be updated. We describe below how to update these bits in $TT(C)$ and the *Cond* 3&4(C) bit in time $O(v(C)\log n)$, where $v(C)$ is the number of neighboring clusters of C. Since the sum of $v(C)$ for all clusters C fulfilling (1) and (2) adds up to $O(m/k)$, the total time for updating all *marked*, *doublesided*, and *Cond* 3&4 bits is $O(m/k \log n)$.

We describe first how to find X_1 and X_2. We set a counter to 0 and start the following recursion using the root of $TT(C)$ as current node: We determine if the children of the current node are double-sided. If the current node has only one red, double-sided child, we recurse on it. If it has two red, double-sided children and the counter is 0, we set the counter to 1 and recurse on both. Otherwise, we terminate. If at termination one of the two current nodes has two red, double-sided children, we set *Cond* 3&4(C) to 0. Otherwise, we set it to 1 and we call the two current nodes X_1 and X_2. Note that this requires to determine for $O(\log n)$ nodes if they are double-sided. Each test takes time $O(v(C))$.

Next we describe how to set the *marked* and *doublesided* bits of the appropriate nodes of $TT(C)$ in $O(v(C)\log n)$ time. First we clear all *marked* bits of $TT(C)$. Then we set the *marked* bit and compute the *doublesided* bit for all ancestors of X_1 and X_2 and the children of X_1 and X_2. Then for each child X

of X_1 or X_2 we execute the following recursive algorithm starting with X as the current node Y: First we set the *marked* bit of Y and determine if Y is double-sided. If Y is not double-sided, set *doublesided*(Y) to 0 and stop. Otherwise, set Y's *doublesided* bit to 1 and determine and set the *doublesided* bit for the at most 2 children of Y. If one child of Y is double-sided and the other child is not double-sided or does not exist, we recurse on the double-sided child, otherwise we quit. This algorithm evaluates $O(\log n)$ nodes in time $O(v(C))$ each. This shows that the *doublesided* and *marked* bits of C can be updated in time $O(v(C)\log n)$.

Note that the space requirement of the presented data structure is $O(m\log n)$, since each cluster requires $O(k\log n)$ space for the labels. However, the space per cluster can be improved to $O(k)$ as follows: Except for the root of $TT(C)$ we store at X instead of $L(C,C',X)$ a *recipe* $r(C,C',X)$ that allows to reconstruct $L(C,C',X)$ if $L(C,C',p(X))$ for the parent $p(X)$ of X is known. Since $L(C,C',p(X))$ is created by concatenating the lists of the children of $p(X)$, the recipe $r(C,C',X)$ consists of 2 pointers into $L(C,C',p(X))$ at the first and last element of $L(C,C',X)$ if $L(C,C',X) \neq L(C,C',p(X))$, and is empty otherwise. The list $L(X)$ contains all non-empty recipes. When computing the bits for X in $TT(C)$, we traverse $TT(C)$ top-down and can reconstruct $L(C,C',X)$ with an overhead of a constant factor since $L(C,C',p(X))$ has been constructed before. The lists at the root of $TT(C)$ require $O(k)$ space. The space needed at each other node is proportional to the number of non-empty recipes at C. Since there are $O(k)$ non-empty recipes, the space per cluster is $O(k)$ and, hence, the total space requirement is $O(m)$.

Theorem 2.24 *The presented data structure tests condition 3–5 in time $O(\log^2 n)$ and can be updated in time $O((k+m/k)\log n)$.*

2.3 The cycle-equivalence of a tree edge and a non-tree edge

The ambivalent data structure of [7] tests if a tree edge is covered by at least one non-tree edge. We extend this data structure to test if a tree edge e_1 is covered by exactly one non-tree edge and, if so, which non-tree edge is covering e_1. This is equivalent to testing the cycle-equivalence of e and a non-tree edge:

Lemma 2.25 [13] *A tree edge e_1 and a non-tree edge e_2 are cycle-equivalent iff e_2 is the only non-tree edge covering e_1.*

Our data structure consists of (1) a topology tree TT of G augmented with recipes and pointers to search trees PP and CP, (2) a labeled 2-dimensional topology tree $2TT$ of G, and (3) for each edge that does not lie on the tree path of its cluster only non-tree edge covering it if such a non-tree edge exists. Maintaining (3) using the static algorithm takes time $O(k)$ per update operation, since it has to be computed for only $O(1)$ clusters. Next we discuss (1) and (2).

We extend the definition of a *tree path* $TP(X)$ (Section 2.1) to clusters X whose level is > 0: If X has one child, $TP(X)$ is the tree path of its child. If X has two children, none of which has tree degree 3, $TP(X)$ is the concatenation of the tree path of the children of X. Otherwise, $TP(X)$ is empty.

A 2-dimensional topology tree maintains a label $l(X,X')$ for two clusters X and X' at the same level of TT in time $O(k+t(n)m/k)$ if Conditions (1) and (2) hold: (1) All labels $l(X,.)$ can be computed in time $O(k)$ if the level of X is 0. (2) The label $l(X,X')$ can be computed in time $t(n)$ from the labels of all pairs of children of X and X' if the level of X is > 0.

Let e be a tree edge incident to X. Let $max_1(X,X',e)$ be the edge between X and X' covering the most edges on $TP(X)$ assuming that $e \in \pi(X,X')$. Assuming that $e \in \pi(X,X')$ and that $max_1(X,X',e)$ does not exist, let $max_2(X,X',e)$ be the edge between X and X' that covers the most edges on $TP(X)$. In [7] it sufficed to maintain max_1. We maintain both max- labels and additional labels as in [7] that are necessary to guarantee that both max-labels fulfill Conditions (1) and (2) with $t(n) = O(1)$. The same holds for the additional labebs. Thus, all max-labels can be maintained in time $O(k+m/k)$.

We keep for each node X of TT a pointers to 2 binary search trees $PP(X)$ and $CP(X)$. The leaves of $PP(X)$ leaves are the edges on $TP(X)$ in the order of their occurrence on the tree path. Every node of $PP(X)$ contains two fields $cover_1$ and $cover_2$. Let e' be an internal non-tree edge of e, let e' and e'' be the first and last edge on $TP(X)$ covered by e and let $r_1,\ldots,r_l$ be the $O(\log n)$ nodes in $PP(X)$ (1) that are either ancestors of e' or e'' or children of these ancestors and (2) whose subtree contains only leaves on the path between e' and e''. We say we *cover* $PP(X)$ *by* e if for every node r_i we store e in $cover_1(r_i)$ if it is empty, or in $cover_2(r_i)$ if $cover_1(r_i)$ is not empty, but $cover_2(r_i)$ is empty. If neither $cover_1(r_i)$ nor $cover_2(r_i)$ are empty, e is not stored at r_i. If X is a level-0 cluster the *cover*-fields are set as follows: Initially, all *cover* fields of nodes of $PP(X)$ are empty. If

an edge e on $TP(X)$ is covered by at least two non-tree edges, then the *cover*-fields of the corresponding leaf in $PP(X)$ store each a different non-tree edge covering e. If e is covered by only one non-tree edge, a pointer to it is stored in the $cover_1$ field and the $cover_2$ field is empty. If e is not covered, both *cover* fields are emtpy. The tree $CP(X)$ and the recipe of X is empty.

Next we discuss the case that the level of X is > 0. If X has only one child X_1, $PP(X)$ is identical to $PP(X_1)$, $CP(X)$ and the recipe of X are empty. If X has two children X_1 and X_2, none of which has tree degree 3, then $PP(X)$ is created by (1) creating a new root node r that points to the roots of $PP(X_1)$ and $PP(X_2)$ and (2) covering the resulting tree by $max_i(X_1, X_2, e)$ and $max_i(X_2, X_1, e)$ for $i = 1, 2$. We also keep at X a list of all modified *cover* fields, called the *recipe* of X. If both children have tree degree 1, then $CP(X)$ is identical to $PP(X)$, otherwise $CP(X)$ is empty. If X has a child X_1 with tree degree 1 and a child X_2 with tree degree 3, connected by a tree edge e, then $PP(X)$ is empty. Let max_1 be the edge of all edges $max_1(X_1, X_1', e)$ that covers the maximum number of edge on $TP(X_1)$ for any cluster $X_1' \neq X_1$, and let max_2 be the edge of all edges $\{max_1(X_1, X_1', e), max_2(X_1, X_1', e)$ with $X_1' \neq X_1\} \setminus \{max_1\}$ that covers the maximum number of edge on $TP(X_1)$. The tree $CP(X)$ is created from $PP(X_1)$ by (1) creating a new root node r that points to the root of $PP(X_1)$ and to a 1-node tree that represents e and (2) covering the resulting tree by max_1 and max_2. We also keep at X a list of all modified *cover* fields, called the *recipe* of X.

Assume a label of a node X of TT can be constructed (1) from the edges and nodes of X in time $t_0(n,k)$ if X is a level-0 cluster and (2) from the label of the parent of X and the recipes in time $O(t_i'(n,k))$. Assume (3) that the label and the recipe can be built from the labels of the children of X in time $O(t_i(n,k))$ if X is a level-i cluster. Then the update algorithm for TT in [6] maintains the labels of all clusters dynamically in time $O(\sum_i t_i(n,k) + t_i'(n,k))$. The trees PP and CP are labels that fulfill Condition (1), (2), and (3) with $t_0(n,k) = O(k \log n)$, $t_i(n,k) = O(n_i + \log n)$ for $i > 0$, where n_i is the number of level-i clusters and $t_i'(n,k) = O(\log n)$. Thus, all trees PP and CP can be maintained in time $O(k \log n + \log^2 n)$.

At a node X we need $O(1)$ space for pointers to the root of $PP(X)$ and of $CP(X)$ and $O(\log n)$ space for the recipe. The trees $PP(X)$ and $CP(X)$ need $O(n)$ total space since if $PP(X)$ has size $O(k)$ for a level-0 cluster, and for each cluster whose level is > 0 we allocate a constant amount of new space. Thus, the whole data structure requires $O(m+n)$ space.

If e_1 does not lie on the tree path of its cluster, we use data structure (3) to test e_1 and e_2. If it does, we determine in time $O(\log n)$ the cluster X whose tree CP contains e_1, traverse the path from the root of $CP(X)$ to e_1 and check whether e_2 is the only edge stored in a *cover*-field on any node along this path. If yes, then e_1 and e_2 are cycle-equivalent, otherwise they are not. Since the depth of TT is $O(\log n)$, the depth of $CP(X)$ is $O(\log n)$. Thus, a query can be answered in time $O(\log n)$.

Theorem 2.26 *We can test in time $O(\log n)$ whether a tree edge and a non-tree edge are cycle-equivalent. The data structure can be updated in time $O(m/k + (k + \log n) \log n)$.*

3 Algorithms for planar graphs and the lower bound

Dynamic connectivity and cycle-equivalence are connected as follows:

Lemma 3.1 *Two edges $e_1 = (x,y)$ and $e_2 = (u,v)$ are cycle-equivalent iff after the removal of e_1 and e_2 either x and y are disconnected or u and v are disconnected.*

This lemma provides the following dynamic algorithm for cycle-equivalence: We maintain a dynamic connectivity data structure. To check the cycle-equivalence of e_1 and e_2 we delete them from the graph and test if x and y disconnected or u and v are disconnected. Then we restore the graph.

The best known dynamic connectivity algorithm in plane (=planar embedded) graphs takes $O(\log n)$ time per operation [2] solving the dynamic cycle-equivalence problem in plane graphs in time $O(\log n)$ time per operation. The best known dynamic connectivity algorithm in planar graphs takes $O(\log n)$ time per insertion or query and $O(\log^2 n)$ time per deletion [5] implying a solution for the dynamic cycle-equivalence problem in planar graphs in time $O(\log n)$ per insertion and $O(\log^2 n)$ per deletion.

We also show a lower bound of $\Omega(\log n / k(\log \log n + \log b))$ on the amortized time per operation for the fully dynamic cycle-equivalence problem in plane and planar graphs where b indicates the wordsize in Yao's cell probe model [17]. (Note that this implies a bound for general graphs.) The lower bound construction is similar to [15]. We reduce the problem to the following parity prefix sum problem (*PPS problem*) for which

a lower bound of $\Omega(\log n/(\log\log n + \log b))$ on the amortized time per operation is shown in [8]:

Given an array $A[1],\ldots,A[n]$ of integers execute $Add(l)$ and $Sum(l)$ operations, where an $Add(l)$ increases $A[l]$ by 1 and a $Sum(l)$ returns $S_l := (\sum_{1\le i\le l} A[i]) \bmod 2$.

The idea of the proof is as follows: Given an instance of the PPS problem, we construct a graph consisting of $n+1$ vertices, labeled $0,\ldots,n$. Vertex l represents S_l. Let $S_0 := 0$. We connect vertex i with vertex j if j is the largest index smaller than i such that $S_j + S_i$ is even. Thus, all vertices l with odd (even) S_l are connected by an *odd* (*even*) chain. Additionally, we insert an edge between the last vertex of the odd chain and vertex 0. In this graph a $Sum(l)$ query corresponds the testing the cycle-equivalence of the edges $(0,1)$ and e_l, where e_l is the edge connecting vertex l to its predecessor on its chain. An $Add(l)$ operation corresponds to a constant number of edge insertions and deletions. With additional care the bound can be shown even in a 2-edge connected graph.

Theorem 3.2 *Fully dynamic cycle-equivalence in a plane 2-edge connected graph requires amortized time* $\Omega(\log n/(\log\log n + \log b))$ *per operation in the cell probe model with wordsize b.*

Acknowledgements

I want to thank Keshav Pingali for bringing this problem to my attention.

References

[1] D. Alberts and M. Rauch Henzinger, Average Case Analysis of Dynamic Graph Algorithms. Technical Report 1994, International Computer Science Institute, Berkeley, CA.

[2] 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.

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

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

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

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

[7] G. N. Frederickson, "Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees" *Proc. 32nd Annual IEEE Sympos. on Foundation of Comput. Sci.*, 1991, 632–641.

[8] M. L. Fredman, and M. E. Saks, "The Cell Probe Complexity of Dynamic Data Structures", *Proc. 19th Annual Symp. on Theory of Computing*, 1989, 345–354.

[9] Z. Galil, G. F. Italiano, "Fully dynamic algorithms for 3-edge connectivity" , Manuscript.

[10] J. Hershberger, M. Rauch, S. Suri, "Data structures for two-edge connectivity in Planar Graphs" *Theoret. Comput. Sci.* 130 (1994), 139–161.

[11] R. Gupta and M. L. Soffa, "Region scheduling" *Proc. 2nd International Conference on Supercomputing,* (1987), 141-148.

[12] R. Johnson and K. Pingali, "Dependence-based program analysis" *Proc. Sigplan'93 PLDI*, 78–89. Published as ACM SIGPLAN Notices 28(6).

[13] R. Johnson, D. Pearson, and K. Pingali, "Finding Regions Fast: Single Entry Single Exit and Control Regions in Linear Time." To appear in *Proc. Sigplan'94 PLDI.*

[14] H. Nagamochi and T. Ibaraki. Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica 7*, pages 583 – 596, 1992.

[15] M. H. Rauch. "Improved Data Structures for Fully Dynamic Biconnectivity" to appear in *Proc. 26 Annual Symp. on Theory of Computing*, 1994.

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

[17] A. Yao, "Should tables be sorted", *J. Assoc. Comput. Mach.*, 28(3), 1981, 615–628.